Package ‘tfprobability’

May 20, 2021

Title  Interface to 'TensorFlow Probability'
Version  0.12.0.0
Description  Interface to 'TensorFlow Probability', a 'Python' library built on 'TensorFlow' that makes it easy to combine probabilistic models and deep learning on modern hardware ('TPU', 'GPU'). 'TensorFlow Probability' includes a wide selection of probability distributions and bijectors, probabilistic layers, variational inference, Markov chain Monte Carlo, and optimizers such as Nelder-Mead, BFGS, and SGLD.
License  Apache License (>= 2.0)
URL  https://github.com/rstudio/tfprobability
BugReports  https://github.com/rstudio/tfprobability/issues
SystemRequirements  TensorFlow Probability
  (https://www.tensorflow.org/probability)
Encoding  UTF-8
RoxygenNote  7.1.1
Imports  tensorflow (>= 2.4.0), reticulate, keras, magrittr
Suggests  tfdatasets, testthat (>= 2.1.0), knitr, rmarkdown
VignetteBuilder  knitr
NeedsCompilation  no
Author  Sigrid Keydana [aut, cre],
  Daniel Falbel [ctb],
  Kevin Kuo [ctb] (<https://orcid.org/0000-0001-7803-7901>),
  RStudio [cph]
Maintainer  Sigrid Keydana <sigrid@rstudio.com>
Repository  CRAN
Date/Publication  2021-05-20 15:20:06 UTC
R topics documented:

- glm_families ......................................................... 8
- glm_fit ............................................................. 9
- glm_fit.tensorflow.tensor ........................................ 9
- glm_fit_one_step .................................................. 11
- glm_fit_one_step.tensorflow.tensor .............................. 11
- initializer_blockwise ............................................. 13
- install_tfprobability ............................................. 13
- layer_autoregressive ............................................. 14
- layer_autoregressive_transform ................................. 16
- layer_categorical_mixture_of_one_hot_categorical ............ 17
- layer_conv_1d_flipout .......................................... 18
- layer_conv_1d_reparameterization ............................... 21
- layer_conv_2d_flipout .......................................... 23
- layer_conv_2d_reparameterization ............................... 26
- layer_conv_3d_flipout .......................................... 29
- layer_conv_3d_reparameterization ............................... 32
- layer_dense_flipout ............................................. 34
- layer_dense_local_reparameterization ......................... 37
- layer_dense_reparameterization ................................ 39
- layer_dense_variational ......................................... 41
- layer_distribution_lambda ....................................... 43
- layer_independent_bernoulli .................................... 44
- layer_independent_logistic ..................................... 45
- layer_independent_normal ....................................... 46
- layer_independent_poisson ...................................... 47
- layer_kl_divergence_add_loss .................................. 48
- layer_kl_divergence_regularizer ................................. 49
- layer_mixture_logistic .......................................... 50
- layer_mixture_normal .......................................... 51
- layer_mixture_same_family ..................................... 52
- layer_multivariate_normal_tri_l ............................... 54
- layer_one_hot_categorical ..................................... 55
- layer_variable ................................................... 56
- layer_variational_gaussian_process ............................ 57
- mcmc_dual_averaging_step_size_adaptation ..................... 58
- mcmc_effective_sample_size ................................... 61
- mcmc_hamiltonian_monte_carlo ................................ 62
- mcmc_metropolis_adjusted_langevin_algorithm ................. 64
- mcmc_metropolis_hastings ...................................... 65
- mcmc_no_u_turn_sampler ........................................ 67
- mcmc_potential_scale_reduction ................................ 70
- mcmc_random_walk_metropolis .................................. 71
- mcmc_replica_exchange_mc ...................................... 73
- mcmc_sample_annealed_importance_chain ......................... 74
- mcmc_sample_chain .............................................. 76
- mcmc_sample_halton_sequence ................................... 78
mcmc_simple_step_size_adaptation .................................................. 80
mcmc_slice_sampler ................................................................. 83
mcmc_transformed_transition_kernel ........................................... 84
mcmc_uncalibrated_hamiltonian_monte_carlo ................................. 86
mcmc_uncalibrated_langevin ....................................................... 87
mcmc_uncalibrated_random_walk ................................................ 88
params_size_categorical_mixture_of_one_hot_categorical ................ 89
params_size_independent_bernoulli ............................................. 90
params_size_independent_logistic ............................................. 90
params_size_independent_normal .............................................. 91
params_size_independent_poisson ............................................. 91
params_size_mixture_logistic ................................................... 92
params_size_mixture_normal .................................................... 92
params_size_mixture_same_family ............................................ 93
params_size_multivariate_normal_tri_l ..................................... 93
params_one_hot_categorical ................................................... 94
sts_additive_state_space_model ............................................. 94
sts_autoregressive ............................................................... 97
sts_autoregressive_state_space_model ..................................... 99
sts_build_factored_surrogate_posterior .................................. 101
sts_build_factored_variational_loss ........................................ 102
sts_constrained_seasonal_state_space_model ............................. 103
sts_decompose_by_component .................................................. 106
sts_decompose_forecast_by_component ..................................... 107
sts_dynamic_linear_regression ............................................... 108
sts_dynamic_linear_regression_state_space_model ....................... 109
sts_fit_with_hmc ................................................................. 111
sts_forecast ....................................................................... 114
sts_linear_regression .......................................................... 116
sts_local_level ................................................................. 117
sts_local_level_state_space_model ......................................... 118
sts_local_linear_trend .......................................................... 120
sts_local_linear_trend_state_space_model ................................ 121
sts_one_step_predictive ........................................................ 123
sts_sample_uniform_initial_state ........................................... 124
sts_seasonal ................................................................. 125
sts_seasonal_state_space_model ............................................. 127
sts_semi_local_linear_trend .................................................... 129
sts_semi_local_linear_trend_state_space_model .......................... 131
sts_smooth_seasonal ............................................................ 133
sts_smooth_seasonal_state_space_model ................................... 135
sts_sparse_linear_regression ................................................. 138
sts_sum ........................................................................... 140
tfb_absolute_value ............................................................... 141
tfb_affine ......................................................................... 142
tfb_affine_linear_operator ..................................................... 144
tfb_affine_scalar ................................................................. 145
tfb_ascending ................................................................. 146
tfb_batch_normalization .................................................. 147
 tfb_blockwise ............................................................... 149
 tfb_chain ......................................................................... 150
 tfb_cholesky_outer_product .............................................. 151
 tfb_cholesky_to_inv_cholesky ............................................ 152
 tfb_correlation_cholesky ................................................... 153
 tfb_cumsum .................................................................... 155
 tfb_discrete_cosine_transform .......................................... 156
 tfb_exp ........................................................................... 157
 tfb_expm1 ....................................................................... 158
 tfb_ffjord ....................................................................... 159
 tfb_fill_scale_tri_l .......................................................... 162
 tfb_fill_triangular ........................................................... 163
 tfb_forward ..................................................................... 164
 tfb_forward_log_det_jacobian ............................................ 165
 tfb_glow ....................................................................... 166
 tfb_gompertz_cdf .............................................................. 169
 tfb_gumbel .................................................................... 171
 tfb_gumbel_cdf ................................................................. 172
 tfb_identity ................................................................... 173
 tfb_inline ........................................................................ 174
 tfb_inverse ...................................................................... 175
 tfb_inverse_log_det_jacobian ............................................ 176
 tfb_invert ....................................................................... 177
 tfb_iterated_sigmoid_centered ......................................... 178
 tfb_kumaraswamy .............................................................. 179
 tfb_kumaraswamy_cdf ........................................................ 180
 tfb_lambert_w_tail ........................................................... 181
 tfb_masked_autoregressive_default_template .......................... 182
 tfb_masked_autoregressive_flow ........................................ 184
 tfb_masked_dense ............................................................... 187
 tfb_matrix_inverse_tri_l ................................................... 189
 tfb_matvec_lu ................................................................. 190
 tfb_normal_cdf ................................................................. 191
 tfb_ordered ..................................................................... 192
 tfb_pad ........................................................................... 193
 tfb_permute ..................................................................... 194
 tfb_power_transform .......................................................... 195
 tfb_rational_quadratic_spline ........................................... 196
 tfb_rayleigh_cdf ............................................................... 198
 tfb_real_nvp .................................................................. 199
 tfb_real_nvp_default_template ......................................... 201
 tfb_reciprocal ................................................................. 203
 tfb_reshape .................................................................... 204
 tfb_scale ........................................................................ 205
 tfb_scale_matvec_diag ..................................................... 206
 tfb_scale_matvec_linear_operator ...................................... 207
 tfb_scale_matvec_lu ........................................................... 208
tfb_scale_matvec_tri_l ........................................ 209
tfb_scale_tri_l ............................................. 211
tfb_shift ...................................................... 212
tfb_shifted_gompertz_cdf .................................... 213
tfb_sigmoid ................................................... 214
tfb_sinh ......................................................... 215
tfb_sinh_arcsinh ............................................... 216
tfb_softmax_centered ......................................... 217
tfb_softplus ................................................... 218
tfb_softsign ................................................... 220
tfb_split ........................................................ 221
tfb_square ....................................................... 222
tfb_tanh .......................................................... 223
tfb_transform_diagonal ....................................... 224
tfb_transpose .................................................. 225
tfb_weibull ...................................................... 226
tfb_weibull_cdf ............................................... 227
tfd_autoregressive ........................................... 229
tfd_batch_reshape ............................................ 231
tfd_bates ......................................................... 232
tfd_bernoulli .................................................... 234
tfd_beta .......................................................... 236
tfd_beta_binomial ............................................. 238
tfd_binomial ..................................................... 240
tfd_blockwise .................................................. 242
tfd_categorical ................................................ 243
tfd_cauchy ....................................................... 244
tfd_cdf ............................................................ 246
tfd_chi ............................................................ 247
  tfd_chi2 ....................................................... 248
tfd_cholesky_lkj ............................................... 250
tfd_continuous_bernoulli ..................................... 251
tfd_covariance ............................................... 253
tfd_cross_entropy ............................................. 254
tfd_deterministic ............................................. 255
tfd_dirichlet ................................................... 257
tfd_dirichlet_multinomial ................................... 259
tfd_doublesided_maxwell ..................................... 261
tfd_empirical .................................................. 262
tfd_entropy ...................................................... 264
tfd_exponential ................................................. 265
tfd_exp_gamma .................................................. 266
tfd_exp_inverse_gamma ...................................... 268
tfd_exp_relaxed_one_hot_categorical ...................... 270
tfd_finite_discrete ........................................... 272
tfd_gamma ........................................................ 273
tfd_gamma_gamma .............................................. 275
tfd_gaussian_process ......................................... 277
tfd_gaussian_process_regression_model ................................................................. 280
  tfd_generalized_normal .......................................................... 283
  tfd_generalized_pareto ............................................................ 285
  tfd_geometric ................................................................. 287
  tfd_gumbel ................................................................. 288
  tfd_half_cauchy .............................................................. 290
  tfd_half_normal .............................................................. 291
  tfd_hidden_markov_model ............................................................... 293
  tfd_horseshoe ................................................................. 295
  tfd_independent ................................................................. 296
  tfd_inverse_gamma ................................................................. 298
  tfd_inverse_gaussian ................................................................. 300
  tfd_johnson_s_u ................................................................. 302
  tfd_joint_distribution_named ............................................................ 304
  tfd_joint_distribution_named_auto_batched ............................................ 306
  tfd_joint_distribution_sequential ....................................................... 308
  tfd_joint_distribution_sequential_auto_batched ......................................... 310
  tfd_kl_divergence ................................................................. 313
  tfd_kumaraswamy ................................................................. 314
  tfd_laplace ................................................................. 315
  tfd_linear_gaussian_state_space_model ................................................... 317
  tfd_lkj ................................................................. 320
  tfd_logistic ................................................................. 321
  tfd_logit_normal ................................................................. 323
  tfd_log_prob ................................................................. 328
  tfd_log_survival_function ............................................................ 328
  tfd_mean ................................................................. 329
  tfd_mixture ................................................................. 330
  tfd_mixture_same_family ............................................................ 331
  tfd_mode ................................................................. 333
  tfd_multinomial ................................................................. 334
  tfd_multivariate_normal_diag ........................................................... 336
  tfd_multivariate_normal_diag_plus_low_rank ........................................... 339
  tfd_multivariate_normal_full_covariance ............................................. 341
  tfd_multivariate_normal_linear_operator ............................................... 343
  tfd_multivariate_normal_tri_l ......................................................... 345
  tfd_multivariate_student_t_linear_operator ........................................... 347
  tfd_negative_binomial .............................................................. 349
  tfd_normal ................................................................. 351
  tfd_one_hot_categorical ............................................................ 353
  tfd_pareto ................................................................. 354
  tfd_pert ................................................................. 356
  tfd_pixel_cnn ............................................................... 358
  tfd_plackett_luce ............................................................... 360
  tfd_poisson ................................................................. 362
topics documented: 7

tfd_poisson_log_normal_quadrature_compound ............................... 363

tfd_power_spherical ............................................................. 366

tfd_prob ................................................................. 368

tfd_probit_bernoulli ......................................................... 368

tfd_quantile .......................................................... 370

tfd_quantized .......................................................... 371

tfd_relaxed_bernoulli ...................................................... 373

tfd_relaxed_one_hot_categorical ....................................... 375

tfd_sample ............................................................... 377

tfd_sample_distribution .................................................. 377

tfd_sinh_arcsinh ............................................................ 379

tfd_skellam ............................................................. 381

tfd_spherical_uniform ...................................................... 383

tfd_stddev ............................................................. 385

tfd_student_t .............................................................. 386

tfd_student_t_process ..................................................... 388

tfd_survival_function .................................................... 390

tfd_transformed_distribution .................................................. 391

tfd_triangular .................................................................. 393

tfd_truncated_cauchy ......................................................... 395

tfd_truncated_normal ......................................................... 396

tfd_uniform ................................................................. 398

tfd_variance .................................................................. 400

tfd_variational_gaussian_process ........................................ 401

tfd_vector_deterministic .................................................. 406

tfd_vector_diffeomixture .................................................. 407

tfd_vector_exponential_diag ............................................... 410

tfd_vector_exponential_linear_operator .................................. 412

tfd_vector_laplace_diag .................................................. 414

tfd_vector_laplace_linear_operator ...................................... 417

tfd_vector_sinh_arcsinh_diag ............................................ 419

tfd_von_mises ............................................................. 421

tfd_von_mises_fisher .......................................................... 423

tfd_weibull ................................................................. 425

tfd_wishart ................................................................. 427

tfd_wishart_linear_operator .................................................. 429

tfd_wishart_tri_l .............................................................. 431

tfd_zipf ................................................................. 433

tfp ................................................................. 434

tfp_version .............................................................. 435

vi_amari_alpha ................................................................. 435

vi_arithmetic_geometric .......................................................... 436

vi_chi_square .............................................................. 437

vi_csiszar_vimco ............................................................... 438

vi_dual_csiszar_function .................................................... 440

vi_fit_surrogate_posterior .................................................. 441

vi_jeffreys ................................................................. 443

vi_jensen_shannon ............................................................. 444
glm_families

Description
A list of models that can be used as the model argument in `glm_fit()`:

Details
- Bernoulli: Bernoulli(probs=mean) where mean = sigmoid(matmul(X,weights))
- BernoulliNormalCDF: Bernoulli(probs=mean) where mean = Normal(0, 1).cdf(matmul(X, weights))
- GammaExp: Gamma(concentration=1, rate=1 / mean) where mean = exp(matmul(X, weights))
- GammaSoftplus: Gamma(concentration=1, rate=1 / mean) where mean = softplus(matmul(X, weights))
- LogNormal: LogNormal(loc=log(mean) - log(2) / 2, scale=sqrt(log(2))) where mean = exp(matmul(X, weights)).
- LogNormalSoftplus: LogNormal(loc=log(mean) - log(2) / 2, scale=sqrt(log(2))) where mean = softplus(matmul(X, weights))
- Normal: Normal(loc=mean, scale=1) where mean = matmul(X, weights).
- NormalReciprocal: Normal(loc=mean, scale=1) where mean = 1 / matmul(X, weights)
- Poisson: Poisson(rate=mean) where mean = exp(matmul(X, weights)).
- PoissonSoftplus: Poisson(rate=mean) where mean = softplus(matmul(X, weights)).

Value
list of models that can be used as the model argument in `glm_fit()`

See Also
Other glm_fit: `glm_fit.tensorflow.tensor()`, `glm_fit_one_step.tensorflow.tensor()`
Description

Runs multiple Fisher scoring steps

Usage

glm_fit(x, ...)

Arguments

x float-like, matrix-shaped Tensor where each row represents a sample’s features.

...
other arguments passed to specific methods.

Value

A glm_fit object with parameter estimates, number of iterations, etc.

See Also

glm_fit.tensorflow.tensor()
Arguments

- **x**: float-like, matrix-shaped Tensor where each row represents a sample's features.
- **response**: vector-shaped Tensor where each element represents a sample's observed response (to the corresponding row of features). Must have same dtype as x.
- **model**: a string naming the model (see `glm_families`) or a `tfp$glm$ExponentialFamily-like` instance which implicitly characterizes a negative log-likelihood loss by specifying the distribution's mean, gradient_mean, and variance.
- **model_coefficients_start**: Optional (batch of) vector-shaped Tensor representing the initial model coefficients, one for each column in x. Must have same dtype as model_matrix. Default value: Zeros.
- **predicted_linear_response_start**: Optional Tensor with shape, dtype matching response; represents offset shifted initial linear predictions based on model_coefficients_start. Default value: offset if model_coefficients is NULL, and `tf.linalg.matvec(x, model_coefficients_start) + offset` otherwise.
- **l2_regularizer**: Optional scalar Tensor representing L2 regularization penalty. Default: NULL ie. no regularization.
- **dispersion**: Optional (batch of) Tensor representing response dispersion.
- **offset**: Optional Tensor representing constant shift applied to predicted_linear_response.
- **convergence_criteria_fn**: callable taking: is_converged_previous, iter_, model_coefficients_previous, predicted_linear_response_previous, model_coefficients_next, predicted_linear_response_next, response, model, dispersion and returning a logical Tensor indicating that Fisher scoring has converged.
- **learning_rate**: Optional (batch of) scalar Tensor used to dampen iterative progress. Typically only needed if optimization diverges, should be no larger than 1 and typically very close to 1. Default value: NULL (i.e., 1).
- **fast_unsafe_numerics**: Optional Python bool indicating if faster, less numerically accurate methods can be employed for computing the weighted least-squares solution. Default value: TRUE (i.e., "fast but possibly diminished accuracy").
- **maximum_iterations**: Optional maximum number of iterations of Fisher scoring to run; "and-ed" with result of convergence_criteria_fn. Default value: NULL (i.e., infinity).
- **name**: used as name prefix to ops created by this function. Default value: "fit".
- **...**: other arguments passed to specific methods.
glm_fit_one_step

Value

A glm_fit object with parameter estimates, and number of required steps.

See Also

Other glm_fit: glm_families, glm_fit_one_step.tensorflow.tensor()

glm_fit_one_step

Runs one Fisher scoring step

Description

Runs one Fisher scoring step

Usage

glm_fit_one_step(x, ...)

Arguments

x float-like, matrix-shaped Tensor where each row represents a sample's features.

... other arguments passed to specific methods.

Value

A glm_fit object with parameter estimates, number of iterations, etc.

See Also

glm_fit_one_step.tensorflow.tensor()
Usage

```r
## S3 method for class 'tensorflow.tensor'
glm_fit_one_step(
  x,
  response,
  model,
  model_coefficients_start = NULL,
  predicted_linear_response_start = NULL,
  l2_regularizer = NULL,
  dispersion = NULL,
  offset = NULL,
  learning_rate = NULL,
  fast_unsafe_numerics = TRUE,
  name = NULL,
  ...
)
```

Arguments

- `x` float-like, matrix-shaped Tensor where each row represents a sample’s features.
- `response` vector-shaped Tensor where each element represents a sample’s observed response (to the corresponding row of features). Must have same `dtype` as `x`.
- `model` a string naming the model (see `glm_families`) or a `tfp$glm$ExponentialFamily-like` instance which implicitly characterizes a negative log-likelihood loss by specifying the distribution’s mean, gradient, variance.
- `model_coefficients_start` Optional (batch of) vector-shaped Tensor representing the initial model coefficients, one for each column in `x`. Must have same `dtype` as `model_matrix`. Default value: Zeros.
- `predicted_linear_response_start` Optional Tensor with shape, `dtype` matching `response`; represents offset shifted initial linear predictions based on `model_coefficients_start`. Default value: offset if `model_coefficients` is NULL, and `tf$linalg$matvec(x, model_coefficients_start)` + offset otherwise.
- `l2_regularizer` Optional scalar Tensor representing L2 regularization penalty. Default: NULL i.e. no regularization.
- `dispersion` Optional (batch of) Tensor representing response dispersion.
- `offset` Optional Tensor representing constant shift applied to `predicted_linear_response`.
- `learning_rate` Optional (batch of) scalar Tensor used to dampen iterative progress. Typically only needed if optimization diverges, should be no larger than 1 and typically very close to 1. Default value: NULL (i.e., 1).
- `fast_unsafe_numerics` Optional Python bool indicating if faster, less numerically accurate methods can be employed for computing the weighted least-squares solution. Default value: TRUE (i.e., "fast but possibly diminished accuracy").
- `name` used as name prefix to ops created by this function. Default value: "fit".
- `...` other arguments passed to specific methods.
initializer_blockwise

Value

A glm_fit object with parameter estimates, and number of required steps.

See Also

Other glm_fit: glm_families, glm_fit.tensorflow.tensor()

initializer_blockwise  Blockwise Initializer

Description

Initializer which concats other initializers

Usage

initializer_blockwise(initializers, sizes, validate_args = FALSE)

Arguments

initializers  list of Keras initializers, eg: keras::initializer_glorot_uniform() or initializer_constant().
sizes  list of integers scalars representing the number of elements associated with each initializer in initializers.
validate_args  bool indicating we should do (possibly expensive) graph-time assertions, if necessary.

@return  Initializer which concats other initializers

install_tfprobability  Installs TensorFlow Probability

Description

Installs TensorFlow Probability

Usage

install_tfprobability(
  method = c("auto", "virtualenv", "conda"),
  conda = "auto",
  version = "default",
  tensorflow = "default",
  extra_packages = NULL,
  ...
)

layer_autoregressive

Arguments

method
  Installation method ("virtualenv" or "conda")

conda
  The path to a conda executable. Use "auto" to allow reticulate to automatically find an appropriate conda binary. See Finding Conda for more details.

version
  Version of Keras to install. Specify "default" to install the latest release. Otherwise specify an alternate version (e.g. "2.2.2").

tensorflow
  TensorFlow version to install. Specify "default" to install the CPU version of the latest release. Specify "gpu" to install the GPU version of the latest release. You can also provide a full major.minor.patch specification (e.g. "1.1.0"), appending "-gpu" if you want the GPU version (e.g. "1.1.0-gpu"). Alternatively, you can provide the full URL to an installer binary (e.g. for a nightly binary).

extra_packages
  Additional PyPI packages to install along with Keras and TensorFlow.

... Other arguments passed to tensorflow::install_tensorflow().

Value

invisible

layer_autoregressive

Masked Autoencoder for Distribution Estimation

Description

layer_autoregressive takes as input a Tensor of shape [..., event_size] and returns a Tensor of shape [..., event_size, params]. The output satisfies the autoregressive property. That is, the layer is configured with some permutation ord of \{0, ..., event_size-1\} (i.e., an ordering of the input dimensions), and the output \text{output}[batch_idx,i,...] depends only on inputs \text{x}[batch_idx,j] where ord(j) < ord(i).

Usage

layer_autoregressive(
  object,
  params,
  event_shape = NULL,
  hidden_units = NULL,
  input_order = "left-to-right",
  hidden_degrees = "equal",
  activation = NULL,
  use_bias = TRUE,
  kernel_initializer = "glorot_uniform",
  validate_args = FALSE,
  ...
)
layer_autoregressive

Arguments

object  Model or layer object
params  integer specifying the number of parameters to output per input.
event_shape  list-like of positive integers (or a single int), specifying the shape of the input to this layer, which is also the event_shape of the distribution parameterized by this layer. Currently only rank-1 shapes are supported. That is, event_shape must be a single integer. If not specified, the event_shape is inferred when this layer is first called or built.
hidden_units  list-like of non-negative integers, specifying the number of units in each hidden layer.
input_order  Order of degrees to the input units: 'random', 'left-to-right', 'right-to-left', or an array of an explicit order. For example, 'left-to-right' builds an autoregressive model: p(x) = p(x1) p(x2 | x1) ... p(xD | x<D). Default: 'left-to-right'.
hidden_degrees  Method for assigning degrees to the hidden units: 'equal', 'random'. If 'equal', hidden units in each layer are allocated equally (up to a remainder term) to each degree. Default: 'equal'.
activation  An activation function. See keras::layer_dense. Default: NULL.
use_bias  Whether or not the dense layers constructed in this layer should have a bias term. See keras::layer_dense. Default: TRUE.
kernel_initializer  Initializer for the kernel weights matrix. Default: 'glorot_uniform'.
validate_args  logical, default FALSE. When TRUE, layer parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs.
...  Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details

The autoregressive property allows us to use output[batch_idx,i] to parameterize conditional distributions: p(x[batch_idx,i] | x[batch_idx, j] for ord(j) < ord(i)) which give us a tractable distribution over input x[batch_idx]:
p(x[batch_idx]) = prod_i p(x[batch_idx, ord(i)] | x[batch_idx, ord(0:i)])

For example, when params is 2, the output of the layer can parameterize the location and log-scale of an autoregressive Gaussian distribution.

Value

a Keras layer

See Also

Other layers: layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational(), layer_variable()
An autoregressive normalizing flow layer, given a layer_autoregressive.

Description
Following Papamakarios et al. (2017), given an autoregressive model \( p(x) \) with conditional distributions in the location-scale family, we can construct a normalizing flow for \( p(x) \).

Usage

\[
\text{layer_autoregressive_transform} \text{(object, made, ...)}
\]

Arguments

- **object**: Model or layer object
- **made**: A Made layer, which must output two parameters for each input.
- **...**: Additional parameters passed to Keras Layer.

Details
Specifically, suppose \( \text{made} \) is a [layer_autoregressive()] – a layer implementing a Masked Autoencoder for Distribution Estimation (MADE) – that computes location and log-scale parameters \( \text{made}(x)[i] \) for each input \( x[i] \). Then we can represent the autoregressive model \( p(x) \) as \( x = f(u) \) where \( u \) is drawn from some base distribution and where \( f \) is an invertible and differentiable function (i.e., a Bijector) and \( f^{-1}(x) \) is defined by:

\[
\text{library(tensorflow)} \\
\text{library(zeallot)} \\
f\_inverse <- \text{function}(x) \{ \\
\text{c(shift, log\_scale) <<- tf\$unstack(made(x), 2, axis = -1L)} \\
(x - shift) * tf\$math\$exp(-log\_scale) \\
\}\n\]

Given a layer_autoregressive() made, a layer_autoregressive_transform() transforms an input \( \text{tfd}_* p(u) \) to an output \( \text{tfd}_* p(x) \) where \( x = f(u) \).

Value

- a Keras layer

References

- Papamakarios et al. (2017)

See Also

- tfb_masked_autoregressive_flow() and layer_autoregressive()
A OneHotCategorical mixture Keras layer from \(k \times (1 + d)\) params.

Description

\(k\) (i.e., num_components) represents the number of component OneHotCategorical distributions and \(d\) (i.e., event_size) represents the number of categories within each OneHotCategorical distribution.

Usage

```r
layer_categorical_mixture_of_one_hot_categorical(
  object,
  event_size,
  num_components,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  sample_dtype = NULL,
  validate_args = FALSE,
  ...
)
```

Arguments

- **object**  Model or layer object
- **event_size**  Scalar integer representing the size of single draw from this distribution.
- **num_components**  Scalar integer representing the number of mixture components. Must be at least 1. (If num_components=1, it’s more efficient to use the OneHotCategorical layer.)
- **convert_to_tensor_fn**  A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- **sample_dtype**  dtype of samples produced by this distribution. Default value: NULL (i.e., previous layer’s dtype).
- **validate_args**  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **...**  Additional arguments passed to args of keras::create_layer.

Details

Typical choices for `convert_to_tensor_fn` include:

- `tfp$distributions$Distribution$sample`
- `tfp$distributions$Distribution$mean`
- `tfp$distributions$Distribution$mode`
Value

a Keras layer

See Also

For an example how to use in a Keras model, see `layer_independent_normal()`.

Other distribution_layers: `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tri_l()`, `layer_one_hot_categorical()`
Arguments

- **object**: Model or layer object
- **filters**: Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
- **kernel_size**: An integer or list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
- **padding**: One of "valid" or "same" (case-insensitive).
- **data_format**: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
- **dilation_rate**: An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
- **activation**: Activation function. Set it to None to maintain a linear activation.
- **activity_regularizer**: Regularizer function for the output.
- **trainable**: Whether the layer weights will be updated during training.
- **kernel_posterior_fn**: Function which creates `tfd$Distribution` instance representing the surrogate posterior of the kernel parameter. Default value: `default_mean_field_normal_fn()`.
- **kernel_posterior_tensor_fn**: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
- **kernel_prior_fn**: Function which creates `tfd$Distribution` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd_normal(loc = 0, scale = 1)`.
- **kernel_divergence_fn**: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution-like` instances and the sample is a Tensor.
- **bias_posterior_fn**: Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).
- **bias_posterior_tensor_fn**: Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.
bias_prior_fn  Function which creates tfd instance. See default_mean_field_normal_fn
docstring for required parameter signature. Default value: NULL (no prior, no
variational inference)

bias_divergence_fn  Function which takes the surrogate posterior distribution, prior distribution and
random variate sample(s) from the surrogate posterior and computes or approxi-
mates the KL divergence. The distributions are tfd$Distribution-like in-
stances and the sample is a Tensor.

...  Additional keyword arguments passed to the keras::layer_dense constructed
by this layer.

Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the
kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias
posteriors,

outputs = f(inputs; kernel, bias), kernel, bias ~ posterior

where f denotes the layer's calculation. It uses the Flipout estimator (Wen et al., 2018), which
performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.
Flipout uses roughly twice as many floating point operations as the reparameterization estimator
but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and
divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the di-
vergences of kernel and/or bias surrogate posteriors and their respective priors. When doing
minibatch stochastic optimization, make sure to scale this loss such that it is applied just once
per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl
/ num_examples_per_epoch to your optimizer). You can access the kernel and/or bias pos-
terior and prior distributions after the layer is built via the kernel_posterior, kernel_prior,
bias_posterior and bias_prior properties.

Value

a Keras layer

References

- Yeming Wen, Paul Vicol, Jimmy Ba, Dustin Tran, and Roger Grosse. Flipout: Efficient
  Pseudo-Independent Weight Perturbations on Mini-Batches. In International Conference on

See Also

Other layers: layer_autoregressive(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(),
layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization().
layer_conv_1d_reparameterization

1D convolution layer (e.g. temporal convolution).

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

```r
layer_conv_1d_reparameterization(
  object,
  filters,
  kernel_size,
  strides = 1,
  padding = "valid",
  data_format = "channels_last",
  dilation_rate = 1,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...
)
```

Arguments

- **object**: Model or layer object
- **filters**: Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
- **kernel_size**: An integer or list of a single integer, specifying the length of the 1D convolution window.
- **strides**: An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding: One of "valid" or "same" (case-insensitive).

data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).

dilation_rate: An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.

activation: Activation function. Set it to None to maintain a linear activation.

activity_regularizer: Regularizer function for the output.

trainable: Whether the layer weights will be updated during training.

kernel_posterior_fn: Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn: Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn: Function which creates tfd instance. See default_mean_field_normal_fn doctstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).

kernel_divergence_fn: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

bias_posterior_fn: Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn: Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn: Function which creates tfd instance. See default_mean_field_normal_fn doctstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn: Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.
Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{outputs} = f(\text{inputs}; \text{kernel}, \text{bias}), \text{kernel}, \text{bias} \sim \text{posterior}
\]

where \( f \) denotes the layer's calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior (\( q(W|x) \)), prior (\( p(W) \)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( k_l \) is the sum of losses for each element of the batch, you should pass \( k_l / \text{num\_examples\_per\_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value

a Keras layer

References


See Also

Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational(), layer_variable()
layer_conv_2d_flipout(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    ...)

Arguments

object Model or layer object
filters Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
kernel_size An integer or list of a single integer, specifying the length of the 1D convolution window.
strides An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding One of "valid" or "same" (case-insensitive).
data_format A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
dilation_rate An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernel_posterior_fn
  Function which creates tfd$Distribution instance representing the surrogate
  posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn
  Function which takes a tfd$Distribution instance and returns a representative
  value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn
  Function which creates tfd$Distribution instance. See default_mean_field_normal_fn
  docstring for required parameter signature. Default value: tfd_normal(loc =
  0, scale = 1).

kernel_divergence_fn
  Function which takes the surrogate posterior distribution, prior distribution and
  random variate sample(s) from the surrogate posterior and computes or approxi-
  mates the KL divergence. The distributions are tfd$Distribution-like in-
  stances and the sample is a Tensor.

bias_posterior_fn
  Function which creates a tfd$Distribution instance representing the surro-
  gate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular
  = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn
  Function which takes a tfd$Distribution instance and returns a representative
  value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn
  Function which creates tfd instance. See default_mean_field_normal_fn
  docstring for required parameter signature. Default value: NULL (no prior, no
  variational inference)

bias_divergence_fn
  Function which takes the surrogate posterior distribution, prior distribution and
  random variate sample(s) from the surrogate posterior and computes or approxi-
  mates the KL divergence. The distributions are tfd$Distribution-like in-
  stances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed
  by this layer.

**Details**

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the
kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias
posterior,

```
outputs = f(inputs; kernel, bias), kernel, bias ~ posterior
```

where f denotes the layer's calculation. It uses the Flipout estimator (Wen et al., 2018), which
performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.
Flipout uses roughly twice as many floating point operations as the reparameterization estimator
but has the advantage of significantly lower variance.
The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the `losses` property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g., if \( k_1 \) is the sum of losses for each element of the batch, you should pass \( k_1 / \text{num_examples_per_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the `kernel_posterior`, `kernel_prior`, `bias_posterior` and `bias_prior` properties.

**Value**

A Keras layer

**References**


**See Also**

Other layers: `layer_autoregressive()`, `layer_conv_1d_flipout()`, `layer_conv_1d_reparameterization()`, `layer_conv_2d_reparameterization()`, `layer_conv_3d_flipout()`, `layer_conv_3d_reparameterization()`, `layer_dense_flipout()`, `layer_dense_local_reparameterization()`, `layer_dense_reparameterization()`, `layer_dense_variational()`, `layer_variable()`

```
layer_conv_2d_reparameterization

2D convolution layer (e.g. spatial convolution over images)
```

**Description**

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

**Usage**

```python
layer_conv_2d_reparameterization(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
)```
layer_conv_2d_reparameterization

activation = NULL,
activity_regularizer = NULL,
 trainable = TRUE,
kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
kernelprior_fn = tfp$layers$util$default_multivariate_normal_fn,
kerneldivergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
bias_prior_fn = NULL,
bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
...
)

Arguments

object Model or layer object
filters Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).
kernel_size An integer or list of a single integer, specifying the length of the 1D convolution window.
strides An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.
padding One of "valid" or "same" (case-insensitive).
data_format A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).
dilation_rate An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernel_posterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernel_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernel_prior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernel_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

bias_posterior_fn
Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn
Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn
Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details
This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

outputs = f(inputs; kernel, bias), kernel, bias ~ posterior

where f denotes the layer's calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl / num_examples_per_epoch to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value
a Keras layer
layer_conv_3d_flipout

References


See Also

Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_reparameterization(), layer_dense_variational(), layer_variable()

layer_conv_3d_flipout 3D convolution layer (e.g. spatial convolution over volumes) with Flipout

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

layer_conv_3d_flipout(
    object,
    filters,
    kernel_size,
    strides = 1,
    padding = "valid",
    data_format = "channels_last",
    dilation_rate = 1,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    ...
)
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>filters</td>
<td>Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).</td>
</tr>
<tr>
<td>kernel_size</td>
<td>An integer or list of a single integer, specifying the length of the 1D convolution window.</td>
</tr>
<tr>
<td>strides</td>
<td>An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.</td>
</tr>
<tr>
<td>padding</td>
<td>One of &quot;valid&quot; or &quot;same&quot; (case-insensitive).</td>
</tr>
<tr>
<td>data_format</td>
<td>A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).</td>
</tr>
<tr>
<td>dilation_rate</td>
<td>An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.</td>
</tr>
<tr>
<td>activation</td>
<td>Activation function. Set it to None to maintain a linear activation.</td>
</tr>
<tr>
<td>trainable</td>
<td>Whether the layer weights will be updated during training.</td>
</tr>
<tr>
<td>kernel_posterior_fn</td>
<td>Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().</td>
</tr>
<tr>
<td>kernel_posterior_tensor_fn</td>
<td>Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %&gt;% tfd_sample().</td>
</tr>
<tr>
<td>kernel_prior_fn</td>
<td>Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).</td>
</tr>
<tr>
<td>kernel_divergence_fn</td>
<td>Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.</td>
</tr>
<tr>
<td>bias_posterior_fn</td>
<td>Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).</td>
</tr>
<tr>
<td>bias_posterior_tensor_fn</td>
<td>Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %&gt;% tfd_sample().</td>
</tr>
</tbody>
</table>
**bias_prior_fn**  
Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

**bias_divergence_fn**  
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

**Details**

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{outputs} = f(\text{inputs}; \text{kernel, bias}), \text{kernel, bias} \sim \text{posterior}
\]

where \( f \) denotes the layer's calculation. It uses the Flipout estimator (Wen et al., 2018), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior (\( q(W|x) \)), prior (\( p(W) \)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( kl \) is the sum of losses for each element of the batch, you should pass \( kl / \text{num_examples_per_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

**Value**

a Keras layer

**References**


**See Also**

- layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_reparameterization(),
layer_conv_3d_reparameterization

3D convolution layer (e.g. spatial convolution over volumes)

Description

This layer creates a convolution kernel that is convolved (actually cross-correlated) with the layer input to produce a tensor of outputs. It may also include a bias addition and activation function on the outputs. It assumes the kernel and/or bias are drawn from distributions.

Usage

layer_conv_3d_reparameterization(
  object,
  filters,
  kernel_size,
  strides = 1,
  padding = "valid",
  data_format = "channels_last",
  dilation_rate = 1,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Model or layer object</td>
</tr>
<tr>
<td>filters</td>
<td>Integer, the dimensionality of the output space (i.e. the number of filters in the convolution).</td>
</tr>
<tr>
<td>kernel_size</td>
<td>An integer or list of a single integer, specifying the length of the 1D convolution window.</td>
</tr>
<tr>
<td>strides</td>
<td>An integer or list of a single integer, specifying the stride length of the convolution. Specifying any stride value != 1 is incompatible with specifying any dilation_rate value != 1.</td>
</tr>
</tbody>
</table>
padding One of "valid" or "same" (case-insensitive).

data_format A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, length, channels) while channels_first corresponds to inputs with shape (batch, channels, length).

dilation_rate An integer or tuple/list of a single integer, specifying the dilation rate to use for dilated convolution. Currently, specifying any dilation_rate value != 1 is incompatible with specifying any strides value != 1.

activation Activation function. Set it to None to maintain a linear activation.

activity_regularizer Regularizer function for the output.

trainable Whether the layer weights will be updated during training.

kernel_posterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().

kernel_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

kernel_prior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).

kernel_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

bias_posterior_fn Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).

bias_posterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.
Details

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[ \text{outputs} = f(\text{inputs}; \text{kernel}, \text{bias}), \text{kernel}, \text{bias} \sim \text{posterior} \]

where \( f \) denotes the layer’s calculation. It uses the reparameterization estimator (Kingma and Welling, 2014), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior \( (q(W|x)) \), prior \( (p(W)) \), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the \text{losses} property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( \text{kl} \) is the sum of losses for each element of the batch, you should pass \( \text{kl} / \text{num_examples_per_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the \text{kernel_posterior}, \text{kernel_prior}, \text{bias_posterior} and \text{bias_prior} properties.

Value

a Keras layer

References


See Also

Other layers: \text{layer\_autoregressive()}, \text{layer\_conv\_1d\_flipout()}, \text{layer\_conv\_1d\_reparameterization()}, \text{layer\_conv\_2d\_flipout()}, \text{layer\_conv\_2d\_reparameterization()}, \text{layer\_conv\_3d\_flipout()}, \text{layer\_dense\_flipout()}, \text{layer\_dense\_local\_reparameterization()}, \text{layer\_dense\_reparameterization()}, \text{layer\_dense\_variational()}, \text{layer\_variable()}
Usage

layer_dense_flipout(
    object,
    units,
    activation = NULL,
    activity_regularizer = NULL,
    trainable = TRUE,
    kernel_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(),
    kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    kernel_prior_fn = tfp$layers$util$default_multivariate_normal_fn,
    kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
    bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
    bias_prior_fn = NULL,
    bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
    seed = NULL,
...
)

Arguments

object Model or layer object
units integer dimensionality of the output space
activation Activation function. Set it to None to maintain a linear activation.
activity_regularizer Regularizer function for the output.
trainable Whether the layer weights will be updated during training.
kernelposterior_fn Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernelposterior_tensor_fn Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernelprior_fn Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernel_divergence_fn Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
bias_posterior_fn Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).
layer_dense_flipout

bias_posterior_tensor_fn
Function which takes a tensorflow Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().

bias_prior_fn
Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tensorflow Distribution-like instances and the sample is a Tensor.

seed
scalar integer which initializes the random number generator. Default value: NULL (i.e., use global seed).

... Additional keyword arguments passed to the keras::layer_dense constructed by this layer.

Details

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{kernel, bias ~ posterior}
\]
\[
\text{outputs = activation(matmul(inputs, kernel) + bias)}
\]

It uses the Flipout estimator (Wen et al., 2018), which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias. Flipout uses roughly twice as many floating point operations as the reparameterization estimator but has the advantage of significantly lower variance.

The arguments permit separate specification of the surrogate posterior (q(W|x)), prior (p(W)), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if kl is the sum of losses for each element of the batch, you should pass kl / num_examples_per_epoch to your optimizer).

Value

a Keras layer

References

layer_dense_local_reparameterization

*Densely-connected layer class with local reparameterization estimator.*

**Description**

This layer implements the Bayesian variational inference analogue to a dense layer by assuming the kernel and/or the bias are drawn from distributions.

**Usage**

```r
layer_dense_local_reparameterization(
  object,
  units,
  activation = NULL,
  activity_regularizer = NULL,
  trainable = TRUE,
  kernel_posterior_fn = tfp$Layers$Util$Default_mean_field_normal_fn(),
  kernel_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  kernel_prior_fn = tfp$Layers$Util$Default_multivariate_normal_fn,
  kernel_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  bias_posterior_fn = tfp$Layers$Util$Default_mean_field_normal_fn(is_singular = TRUE),
  bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
  bias_prior_fn = NULL,
  bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
  ...)
```

**Arguments**

- **object**
  - Model or layer object
- **units**
  - integer dimensionality of the output space
- **activation**
  - Activation function. Set it to None to maintain a linear activation.
- **activity_regularizer**
  - Regularizer function for the output.
- **trainable**
  - Whether the layer weights will be updated during training.
- **kernel_posterior_fn**
  - Function which creates `tfd$Distribution` instance representing the surrogate posterior of the kernel parameter. Default value: `default_mean_field_normal_fn()`.
layer_dense_local_reparameterization

kernel_posterior_tensor_fn
Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.

kernel_prior_fn
Function which creates `tfd$Distribution` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `tfd_normal(loc = 0, scale = 1)`.

kernel_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a `Tensor`.

bias_posterior_fn
Function which creates a `tfd$Distribution` instance representing the surrogate posterior of the bias parameter. Default value: `default_mean_field_normal_fn(is_singular = TRUE)` (which creates an instance of `tfd_deterministic`).

bias_posterior_tensor_fn
Function which takes a `tfd$Distribution` instance and returns a representative value. Default value: `function(d) d %>% tfd_sample()`.

bias_prior_fn
Function which creates `tfd` instance. See `default_mean_field_normal_fn` docstring for required parameter signature. Default value: `NULL` (no prior, no variational inference)

bias_divergence_fn
Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are `tfd$Distribution`-like instances and the sample is a `Tensor`.

... Additional keyword arguments passed to the `keras::layer_dense` constructed by this layer.

Details

By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

```
kernel, bias ~ posterior
outputs = activation(matmul(inputs, kernel) + bias)
```

It uses the local reparameterization estimator (Kingma et al., 2015), which performs a Monte Carlo approximation of the distribution on the hidden units induced by the kernel and bias. The default `kernel_posterior_fn` is a normal distribution which factorizes across all elements of the weight matrix and bias vector. Unlike that paper’s multiplicative parameterization, this distribution has trainable location and scale parameters which is known as an additive noise parameterization (Molchanov et al., 2017).

The arguments permit separate specification of the surrogate posterior (`q(W|x)`), prior (`p(W)`), and divergence for both the kernel and bias distributions.
Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \( k_l \) is the sum of losses for each element of the batch, you should pass \( k_l / \text{num\_examples\_per\_epoch} \) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel\_posterior, kernel\_prior, bias\_posterior and bias\_prior properties.

Value
a Keras layer

References

See Also
Other layers: layer\_autoregressive(), layer\_conv\_1d\_flipout(), layer\_conv\_1d\_reparameterization(), layer\_conv\_2d\_flipout(), layer\_conv\_2d\_reparameterization(), layer\_conv\_3d\_flipout(), layer\_conv\_3d\_reparameterization(), layer\_dense\_flipout(), layer\_dense\_reparameterization(), layer\_dense\_variational(), layer\_variable()
bias_posterior_fn = tfp$layers$util$default_mean_field_normal_fn(is_singular = TRUE),
bias_posterior_tensor_fn = function(d) d %>% tfd_sample(),
bias_prior_fn = NULL,
bias_divergence_fn = function(q, p, ignore) tfd_kl_divergence(q, p),
...)

Arguments

object      Model or layer object
units       integer dimensionality of the output space
activation  Activation function. Set it to None to maintain a linear activation.
activity_regularizer
            Regularizer function for the output.
trainable   Whether the layer weights will be updated during training.
kernel_posterior_fn
            Function which creates tfd$Distribution instance representing the surrogate posterior of the kernel parameter. Default value: default_mean_field_normal_fn().
kernel_posterior_tensor_fn
            Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
kernel_prior_fn
            Function which creates tfd$Distribution instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: tfd_normal(loc = 0, scale = 1).
kernel_divergence_fn
            Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
bias_posterior_fn
            Function which creates a tfd$Distribution instance representing the surrogate posterior of the bias parameter. Default value: default_mean_field_normal_fn(is_singular = TRUE) (which creates an instance of tfd_deterministic).
bias_posterior_tensor_fn
            Function which takes a tfd$Distribution instance and returns a representative value. Default value: function(d) d %>% tfd_sample().
bias_prior_fn
            Function which creates tfd instance. See default_mean_field_normal_fn docstring for required parameter signature. Default value: NULL (no prior, no variational inference)
bias_divergence_fn
            Function which takes the surrogate posterior distribution, prior distribution and random variate sample(s) from the surrogate posterior and computes or approximates the KL divergence. The distributions are tfd$Distribution-like instances and the sample is a Tensor.
...

Additional keyword arguments passed to the keras::layer_dense constructed by this layer.
Details
By default, the layer implements a stochastic forward pass via sampling from the kernel and bias posteriors,

\[
\text{kernel, bias } \sim \text{ posterior}
\]

\[
\text{outputs} = \text{activation(matmul(inputs, kernel) + bias)}
\]

It uses the reparameterization estimator (Kingma and Welling, 2014) which performs a Monte Carlo approximation of the distribution integrating over the kernel and bias.

The arguments permit separate specification of the surrogate posterior \(q(W|x)\), prior \(p(W)\), and divergence for both the kernel and bias distributions.

Upon being built, this layer adds losses (accessible via the losses property) representing the divergences of kernel and/or bias surrogate posteriors and their respective priors. When doing minibatch stochastic optimization, make sure to scale this loss such that it is applied just once per epoch (e.g. if \(k_l\) is the sum of losses for each element of the batch, you should pass \(k_l / \text{num\_examples\_per\_epoch}\) to your optimizer). You can access the kernel and/or bias posterior and prior distributions after the layer is built via the kernel_posterior, kernel_prior, bias_posterior and bias_prior properties.

Value
a Keras layer

References

See Also
Other layers: layer_autoregressive(), layer_conv_1d_flipout(), layer_conv_1d_reparameterization(), layer_conv_2d_flipout(), layer_conv_2d_reparameterization(), layer_conv_3d_flipout(), layer_conv_3d_reparameterization(), layer_dense_flipout(), layer_dense_local_reparameterization(), layer_dense_variational(), layer_variable()

layer_dense_variational

Dense Variational Layer

Description
This layer uses variational inference to fit a "surrogate" posterior to the distribution over both the kernel matrix and the bias terms which are otherwise used in a manner similar to layer_dense().
This layer fits the "weights posterior" according to the following generative process:

\[
[K, b] \sim \text{Prior()}
\]

\[
M = \text{matmul}(X, K) + b
\]

\[
Y \sim \text{Likelihood}(M)
\]
Usage

layer_dense_variational(
  object,
  units,
  make_posterior_fn,
  make_prior_fn,
  kl_weight = NULL,
  kl_use_exact = FALSE,
  activation = NULL,
  use_bias = TRUE,
  ...
)

Arguments

object Model or layer object
units Positive integer, dimensionality of the output space.
make_posterior_fn function taking \( \text{tf}\$\text{size}(\text{kernel}), \text{tf}\$\text{size}(\text{bias}), \text{dtype} \) and returns another callable which takes an input and produces a \( \text{tfd}\$\text{Distribution} \) instance.
make_prior_fn function taking \( \text{tf}\$\text{size}(\text{kernel}), \text{tf}\$\text{size}(\text{bias}), \text{dtype} \) and returns another callable which takes an input and produces a \( \text{tfd}\$\text{Distribution} \) instance.
kl_weight Amount by which to scale the KL divergence loss between prior and posterior.
kl_use_exact Logical indicating that the analytical KL divergence should be used rather than a Monte Carlo approximation.
activation An activation function. See \texttt{keras::layer_dense}. Default: \texttt{NULL}.
use_bias Whether or not the dense layers constructed in this layer should have a bias term. See \texttt{keras::layer_dense}. Default: \texttt{TRUE}.

Value

a Keras layer

See Also

Other layers: \texttt{layer_autoregressive()}, \texttt{layer_conv_1d_flipout()}, \texttt{layer_conv_1d_reparameterization()}, \texttt{layer_conv_2d_flipout()}, \texttt{layer_conv_2d_reparameterization()}, \texttt{layer_conv_3d_flipout()}, \texttt{layer_conv_3d_reparameterization()}, \texttt{layer_dense_flipout()}, \texttt{layer_dense_local_reparameterization()}, \texttt{layer_dense_reparameterization()}, \texttt{layer_variable()}
layer_distribution_lambda

Keras layer enabling plumbing TFP distributions through Keras models

Description

Keras layer enabling plumbing TFP distributions through Keras models

Usage

layer_distribution_lambda(
  object,
  make_distribution_fn,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  ...
)

Arguments

object Model or layer object
make_distribution_fn A callable that takes previous layer outputs and returns a tfd$distributions$Distribution instance.
convert_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
...
... Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
An Independent-Bernoulli Keras layer from prod(event_shape) params

### Description
An Independent-Bernoulli Keras layer from prod(event_shape) params

### Usage
```r
code

layer_independent_bernoulli(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  sample_dtype = NULL,
  validate_args = FALSE,
  ...
)
```

### Arguments
- `object` Model or layer object
- `event_shape` Scalar integer representing the size of a single draw from this distribution.
- `convert_to_tensor_fn` A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- `sample_dtype` Dtype of samples produced by this distribution. Default value: NULL (i.e., previous layer’s dtype).
- `validate_args` Logical, default FALSE. When TRUE, distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE, invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
- `...` Additional arguments passed to args of `keras::create_layer`.

### Value
a Keras layer

### See Also
- For an example on how to use in a Keras model, see `layer_independent_normal()`.
- Other distribution layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tri_l()`, `layer_one_hot_categorical()`
layer_independent_logistic

An independent Logistic Keras layer.

Description

An independent Logistic Keras layer.

Usage

layer_independent_logistic(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)

Arguments

object Model or layer object

event_shape Scalar integer representing the size of single draw from this distribution.

convert_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ... Additional arguments passed to args of keras::create_layer.

...

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
layer_independent_normal

An independent Normal Keras layer.

Description

An independent Normal Keras layer.

Usage

layer_independent_normal(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)

Arguments

object Model or layer object

event_shape Scalar integer representing the size of single draw from this distribution.

class_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like
doct test. Default value: tfd$distributions$Distribution$sample.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE. @param ...
Additional arguments passed to args of keras::create_layer.

... Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

Other distribution layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(),
layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_poisson(),
layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(),
layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(),
layer_one_hot_categorical()
Examples

```r
library(keras)
input_shape <- c(28, 28, 1)
encoded_shape <- 2
n <- 2
model <- keras_model_sequential(
  list(
    layer_input(shape = input_shape),
    layer_flatten(),
    layer_dense(units = n),
    layer_dense(units = params_size_independent_normal(encoded_shape)),
    layer_independent_normal(event_shape = encoded_shape)
  )
)
```

layer_independent_poisson

_An independent Poisson Keras layer._

Description

An independent Poisson Keras layer.

Usage

```r
layer_independent_poisson(
  object,
  event_shape,
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)
```

Arguments

- **object**: Model or layer object
- **event_shape**: Scalar integer representing the size of single draw from this distribution.
- **convert_to_tensor_fn**: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
- **...**: Additional arguments passed to args of keras::create_layer.

... Additional arguments passed to args of keras::create_layer.
layer_kl_divergence_add_loss

Pass-through layer that adds a KL divergence penalty to the model loss

Description
Pass-through layer that adds a KL divergence penalty to the model loss

Usage
layer_kl_divergence_add_loss(
  object,
  distribution_b,
  use_exact_kl = FALSE,
  test_points_reduce_axis = NULL,
  test_points_fn = tf$convert_to_tensor,
  weight = NULL,
  ...
)

Arguments

object Model or layer object
distribution_b Distribution instance corresponding to $b$ as in $KL[a,b]$. The previous layer’s output is presumed to be a Distribution instance and is $a$.
use_exact_kl Logical indicating if KL divergence should be calculated exactly via tfp$distributions$kl_divergence or via Monte Carlo approximation. Default value: FALSE.
test_points_reduce_axis Integer vector or scalar representing dimensions over which to reduce_mean while calculating the Monte Carlo approximation of the KL divergence. As is with all tf$reduce_*$ ops, NULL means reduce over all dimensions; () means reduce over none of them. Default value: () (i.e., no reduction).
test_points_fn  A callable taking a tfp$distributions$Distribution instance and returning a tensor used for random test points to approximate the KL divergence. Default value: tf$convert_to_tensor.

weight  Multiplier applied to the calculated KL divergence for each Keras batch member. Default value: NULL (i.e., do not weight each batch member).

...  Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal()

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
Arguments

object  
Model or layer object

distribution_b  
Distribution instance corresponding to b as in KL[a,b]. The previous layer’s output is presumed to be a Distribution instance and is a.

use_exact_kl  
Logical indicating if KL divergence should be calculated exactly via tfp$distributions$kl_divergence or via Monte Carlo approximation. Default value: FALSE.

test_points_reduce_axis  
Integer vector or scalar representing dimensions over which to reduce_mean while calculating the Monte Carlo approximation of the KL divergence. As is with all tf$reduce_* ops, NULL means reduce over all dimensions; () means reduce over none of them. Default value: () (i.e., no reduction).

test_points_fn  
A callable taking a tfp$distributions$Distribution instance and returning a tensor used for random test points to approximate the KL divergence. Default value: tf$convert_to_tensor.

weight  
Multiplier applied to the calculated KL divergence for each Keras batch member. Default value: NULL (i.e., do not weight each batch member).

...  
Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_mixture_logistic(), layer_mixture_normal(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
Usage

```r
layer_mixture_logistic(
  object,
  num_components,
  event_shape = list(),
  convert_to_tensor_fn = tfp$distributions$Distribution$sample,
  validate_args = FALSE,
  ...
)
```

Arguments

- **object**: Model or layer object
- **num_components**: Number of component distributions in the mixture distribution.
- **event_shape**: integer vector Tensor representing the shape of single draw from this distribution.
- **convert_to_tensor_fn**: A callable that takes a tfd$Distribution$ instance and returns a tf$Tensor-like$ object. Default value: tfd$distributions$Distribution$sample$.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ... Additional arguments passed to args of keras::create_layer.
- **...**: Additional arguments passed to args of keras::create_layer.

Value

a Keras layer

See Also

For an example how to use in a Keras model, see `layer_independent_normal()`.

Other distribution layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_multivariate_normal_tril()`, `layer_one_hot_categorical()`

---

**layer_mixture_normal**

A mixture distribution Keras layer, with independent normal components.

Description

A mixture distribution Keras layer, with independent normal components.
layer_mixture_same_family

Usage

layer_mixture_normal(
    object,
    num_components,
    event_shape = list(),
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    validate_args = FALSE,
    ...
)

Arguments

- object: Model or layer object
- num_components: Number of component distributions in the mixture distribution.
- event_shape: integer vector Tensor representing the shape of single draw from this distribution.
- convert_to_tensor_fn: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...
  Additional arguments passed to args of keras::create_layer.
- ...: Additional arguments passed to args of keras::create_layer.

Value

- a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_same_family(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()

layer_mixture_same_family

A mixture (same-family) Keras layer.

Description

A mixture (same-family) Keras layer.
Usage

layer_mixture_same_family(
    object,
    num_components,
    component_layer,
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    validate_args = FALSE,
    ...
)

Arguments

object Model or layer object
num_components Number of component distributions in the mixture distribution.
component_layer Function that, given a tensor of shape batch_shape + [num_components, component_params_size], returns a tfd.Distribution-like instance that implements the component distribution (with batch shape batch_shape + [num_components]) – e.g., a TFP distribution layer.
convert_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE. @param ...

Value

a Keras layer

See Also

For an example how to use in a Keras model, see layer_independent_normal().

Other distribution_layers: layer_categorical_mixture_of_one_hot_categorical(), layer_distribution_lambda(), layer_independent_bernoulli(), layer_independent_logistic(), layer_independent_normal(), layer_independent_poisson(), layer_kl_divergence_add_loss(), layer_kl_divergence_regularizer(), layer_mixture_logistic(), layer_mixture_normal(), layer_multivariate_normal_tri_l(), layer_one_hot_categorical()
A \(d\)-variate Multivariate Normal TriL Keras layer from \(d + d\cdot(d+1)/2\) params

### Description

A \(d\)-variate Multivariate Normal TriL Keras layer from \(d + d\cdot(d+1)/2\) params

### Usage

```r
layer_multivariate_normal_tri_l(
    object,
    event_size,
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    validate_args = FALSE,
    ...
)
```

### Arguments

- **object**: Model or layer object
- **event_size**: Integer vector tensor representing the shape of single draw from this distribution.
- **convert_to_tensor_fn**: A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **...**: Additional arguments passed to args of keras::create_layer.

### Value

a Keras layer

### See Also

For an example how to use in a Keras model, see `layer_independent_normal()`

Other distribution_layers: `layer_categorical_mixture_of_one_hot_categorical()`, `layer_distribution_lambda()`, `layer_independent_bernoulli()`, `layer_independent_logistic()`, `layer_independent_normal()`, `layer_independent_poisson()`, `layer_kl_divergence_add_loss()`, `layer_kl_divergence_regularizer()`, `layer_mixture_logistic()`, `layer_mixture_normal()`, `layer_mixture_same_family()`, `layer_one_hot_categorical()`
layer_one_hot_categorical

A d-variate OneHotCategorical Keras layer from d params.

Description

Typical choices for convert_to_tensor_fn include:

- tfp$distributions$Distribution$sample
- tfp$distributions$Distribution$mean
- tfp$distributions$Distribution$mode
- tfp$distributions$OneHotCategorical$logits

Usage

layer_one_hot_categorical(
    object,
    event_size,
    convert_to_tensor_fn = tfp$distributions$Distribution$sample,
    sample_dtype = NULL,
    validate_args = FALSE,
    ...
)

Arguments

object Model or layer object
event_size Scalar integer representing the size of single draw from this distribution.
convert_to_tensor_fn A callable that takes a tfd$Distribution instance and returns a tf$Tensor-like object. Default value: tfd$distributions$Distribution$sample.
sample_dtype dtype of samples produced by this distribution. Default value: NULL (i.e., previous layer’s dtype).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

Value

a Keras layer
See Also

For an example how to use it in a Keras model, see \texttt{layer\_independent\_normal}.

Other distribution layers: \texttt{layer\_categorical\_mixture\_of\_one\_hot\_categorical}, \texttt{layer\_distribution\_lambda}, \texttt{layer\_independent\_bernoulli}, \texttt{layer\_independent\_logistic}, \texttt{layer\_independent\_normal}, \texttt{layer\_independent\_poisson}, \texttt{layer\_kl\_divergence\_add\_loss}, \texttt{layer\_kl\_divergence\_regularizer}, \texttt{layer\_mixture\_logistic}, \texttt{layer\_mixture\_normal}, \texttt{layer\_mixture\_same\_family}, \texttt{layer\_multivariate\_normal}.

---

\texttt{layer\_variable} \hspace{1cm} \textit{Variable Layer}

\textbf{Description}

Simply returns a (trainable) variable, regardless of input. This layer implements the mathematical function \( f(x) = c \) where \( c \) is a constant, i.e., unchanged for all \( x \). Like other Keras layers, the constant is trainable. This layer can also be interpreted as the special case of \texttt{layer\_dense} when the kernel is forced to be the zero matrix (\texttt{tf\$zeros}).

\textbf{Usage}

\begin{verbatim}
layer_variable(
    object, 
    shape, 
    dtype = NULL, 
    activation = NULL, 
    initializer = "zeros", 
    regularizer = NULL, 
    constraint = NULL, 
    ...
)
\end{verbatim}

\textbf{Arguments}

- \texttt{object} \hspace{1cm} Model or layer object
- \texttt{shape} \hspace{1cm} integer or integer vector specifying the shape of the output of this layer.
- \texttt{dtype} \hspace{1cm} TensorFlow dtype of the variable created by this layer.
- \texttt{activation} \hspace{1cm} An activation function. See \texttt{keras::layer\_dense}. Default: \texttt{NULL}.
- \texttt{initializer} \hspace{1cm} Initializer for the constant vector.
- \texttt{regularizer} \hspace{1cm} Regularizer function applied to the constant vector.
- \texttt{constraint} \hspace{1cm} Constraint function applied to the constant vector.
- ... \hspace{1cm} Additional keyword arguments passed to the \texttt{keras::layer\_dense} constructed by this layer.

\textbf{Value}

a Keras layer
A Variational Gaussian Process Layer.

Description

Create a Variational Gaussian Process distribution whose index_points are the inputs to the layer. Parameterized by number of inducing points and a kernel_provider, which should be a tf.keras.Layer with an @property that late-binds variable parameters to a tfp.positive_semidefinite_kernel.PositiveSemidefiniteKernel instance (this requirement has to do with the way that variables must be created in a keras model). The mean_fn is an optional argument which, if omitted, will be automatically configured to be a constant function with trainable variable output.

Usage

layer_variational_gaussian_process(
    object,
    num_inducing_points,
    kernel_provider,
    event_shape = 1,
    inducing_index_points_initializer = NULL,
    unconstrained_observation_noise_variance_initializer = NULL,
    mean_fn = NULL,
    jitter = 1e-06,
    name = NULL
)

Arguments

object Model or layer object
num_inducing_points number of inducing points in the Variational Gaussian Process distribution.
kernel_provider a Layer instance equipped with an @property, which yields a PositiveSemidefiniteKernel instance. The latter is used to parametrize the constructed Variational Gaussian Process distribution returned by calling the layer.
event_shape the shape of the output of the layer. This translates to a batch of underlying Variational Gaussian Process distributions. For example, event_shape = 3 means we are modelling a batch of 3 distributions over functions. We can think of this as a distribution over 3-dimensional vector-valued functions.
inducing_index_points_initializer
    a tf.keras.initializer.Initializer used to initialize the trainable inducing_index_points variables. Training VGP's is pretty sensitive to choice of initial inducing index point locations. A reasonable heuristic is to scatter them near the data, not too close to each other.

unconstrained_observation_noise_variance_initializer
    a tf.keras.initializer.Initializer used to initialize the unconstrained observation noise variable. The observation noise variance is computed from this variable via the tf.nn.softplus function.

mean_fn
    a callable that maps layer inputs to mean function values. Passed to the mean_fn parameter of Variational Gaussian Process distribution. If omitted, defaults to a constant function with trainable variable value.

jitter
    a small term added to the diagonal of various kernel matrices for numerical stability.

name
    name to give to this layer and the scope of ops and variables it contains.

Value
    a Keras layer

mcmc_dual_averaging_step_size_adaptation

Adapts the inner kernel’s step_size based on log_accept_prob.

Description

The dual averaging policy uses a noisy step size for exploration, while averaging over tuning steps to provide a smoothed estimate of an optimal value. It is based on section 3.2 of Hoffman and Gelman (2013), which modifies the stochastic convex optimization scheme of Nesterov (2009). The modified algorithm applies extra weight to recent iterations while keeping the convergence guarantees of Robbins-Monro, and takes care not to make the step size too small too quickly when maintaining a constant trajectory length, to avoid expensive early iterations. A good target acceptance probability depends on the inner kernel. If this kernel is HamiltonianMonteCarlo, then 0.6-0.9 is a good range to aim for. For RandomWalkMetropolis this should be closer to 0.25. See the individual kernels’ docstrings for guidance.

Usage

mcmc_dual_averaging_step_size_adaptation(
    inner_kernel,
    num_adaptation_steps,
    target_accept_prob = 0.75,
    exploration_shrinkage = 0.05,
    step_count_smoothing = 10,
    decay_rate = 0.75,
    step_size_setter_fn = NULL,
Arguments

**inner_kernel**  
TransitionKernel-like object.

**num_adaptation_steps**  
Scalar integer Tensor number of initial steps to during which to adjust the step size. This may be greater, less than, or equal to the number of burnin steps.

**target_accept_prob**  
A floating point Tensor representing desired acceptance probability. Must be a positive number less than 1. This can either be a scalar, or have shape [num_chains]. Default value: 0.75 (the center of asymptotically optimal rate for HMC).

**exploration_shrinkage**  
Floating point scalar Tensor. How strongly the exploration rate is biased towards the shrinkage target.

**step_count_smoothing**  
Int32 scalar Tensor. Number of "pseudo-steps" added to the number of steps taken to prevents noisy exploration during the early samples.

**decay_rate**  
Floating point scalar Tensor. How much to favor recent iterations over earlier ones. A value of 1 gives equal weight to all history.

**step_size_setter_fn**  
A function with the signature (kernel_results, new_step_size) -> new_kernel_results where kernel_results are the results of the inner_kernel, new_step_size is a Tensor or a nested collection of Tensors with the same structure as returned by the step_size_getter_fn, and new_kernel_results are a copy of kernel_results with the step size(s) set.

**step_size_getter_fn**  
A callable with the signature (kernel_results) -> step_size where kernel_results are the results of the inner_kernel, and step_size is a floating point Tensor or a nested collection of such Tensors.

**log_accept_prob_getter_fn**  
A callable with the signature (kernel_results) -> log_accept_prob where kernel_results are the results of the inner_kernel, and log_accept_prob is a floating point Tensor. log_accept_prob can either be a scalar, or have shape [num_chains]. If it's the latter, step_size should also have the same leading dimension.

**validate_args**  
logical. When TRUE kernel parameters are checked for validity. When FALSE invalid inputs may silently render incorrect outputs.

**name**  
name prefixed to Ops created by this function. Default value: NULL (i.e., 'dual_averaging_step_size_adaptation').
Details

In general, adaptation prevents the chain from reaching a stationary distribution, so obtaining consistent samples requires `num_adaptation_steps` be set to a value somewhat smaller than the number of burnin steps. However, it may sometimes be helpful to set `num_adaptation_steps` to a larger value during development in order to inspect the behavior of the chain during adaptation. The step size is assumed to broadcast with the chain state, potentially having leading dimensions corresponding to multiple chains. When there are fewer of those leading dimensions than there are chain dimensions, the corresponding dimensions in the `log_accept_prob` are averaged (in the direct space, rather than the log space) before being used to adjust the step size. This means that this kernel can do both cross-chain adaptation, or per-chain step size adaptation, depending on the shape of the step size. For example, if your problem has a state with shape `[S]`, your chain state has shape `[C0, C1, S]` (meaning that there are \(C0 \times C1\) total chains) and `log_accept_prob` has shape `[C0, C1]` (one acceptance probability per chain), then depending on the shape of the step size, the following will happen:

- Step size has shape `[]`, `[S]` or `[1]`, the `log_accept_prob` will be averaged across its `C0` and `C1` dimensions. This means that you will learn a shared step size based on the mean acceptance probability across all chains. This can be useful if you don’t have a lot of steps to adapt and want to average away the noise.

- Step size has shape `[C1, 1]` or `[C1, S]`, the `log_accept_prob` will be averaged across its `C0` dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the `C1` dimension. This can be useful when the `C1` dimension indexes different distributions, while `C0` indexes replicas of a single distribution, all sampled in parallel.

- Step size has shape `[C0, C1, 1]` or `[C0, C1, S]`, then no averaging will happen. This means that each chain will learn its own step size. This can be useful when all chains are sampling from different distributions. Even when all chains are for the same distribution, this can help during the initial warmup period.

- Step size has shape `[C0, 1, 1]` or `[C0, 1, S]`, the `log_accept_prob` will be averaged across its `C1` dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the `C0` dimension. This can be useful when the `C0` dimension indexes different distributions, while `C1` indexes replicas of a single distribution, all sampled in parallel.

Value

A Monte Carlo sampling kernel

References


mcmc_effective_sample_size

See Also

For an example how to use see mcmc_no_u_turn_sampler().

Other mcmc_kernels: mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

mcmc_effective_sample_size

Estimate a lower bound on effective sample size for each independent chain.

Description

Roughly speaking, "effective sample size" (ESS) is the size of an iid sample with the same variance as state.

Usage

mcmc_effective_sample_size(
  states,
  filter_threshold = 0,
  filter_beyond_lag = NULL,
  name = NULL
)

Arguments

states Tensor or list of Tensor objects. Dimension zero should index identically distributed states.

filter_threshold Tensor or list of Tensor objects. Must broadcast with state. The auto-correlation sequence is truncated after the first appearance of a term less than filter_threshold. Setting to NULL means we use no threshold filter. Since |R_k| <= 1, setting to any number less than -1 has the same effect.

filter_beyond_lag Tensor or list of Tensor objects. Must be int-like and scalar valued. The auto-correlation sequence is truncated to this length. Setting to NULL means we do not filter based on number of lags.

name name to prepend to created ops.
Details

More precisely, given a stationary sequence of possibly correlated random variables $X_1, X_2,...,X_N$, each identically distributed ESS is the number such that $\text{Variance\{ N**-1 * \sum\{X_i\} \} = ESS**-1 * \text{Variance\{ X_1 \}}}$.

If the sequence is uncorrelated, $ESS = N$. In general, one should expect $ESS \leq N$, with more highly correlated sequences having smaller ESS.

Value

Tensor or list of Tensor objects. The effective sample size of each component of states. Shape will be states$shape[1:]$.

See Also

Other mcmc_functions: mcmc_potential_scale_reduction(), mcmc_sample_annealed_importance_chain(), mcmc_sample_chain(), mcmc_sample_halton_sequence()

mcmc_hamiltonian_monte_carlo

Runs one step of Hamiltonian Monte Carlo.

Description

Hamiltonian Monte Carlo (HMC) is a Markov chain Monte Carlo (MCMC) algorithm that takes a series of gradient-informed steps to produce a Metropolis proposal. This class implements one random HMC step from a given current_state. Mathematical details and derivations can be found in Neal (2011).

Usage

mcmc_hamiltonian_monte_carlo(
  target_log_prob_fn,
  step_size,
  num_leapfrog_steps,
  state_gradients_are_stopped = FALSE,
  step_size_update_fn = NULL,
  seed = NULL,
  store_parameters_in_results = FALSE,
  name = NULL
)

Arguments

target_log_prob_fn

Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.
mcmc_hamiltonian_monte_carlo

step_size  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

num_leapfrog_steps  Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to step_size * num_leapfrog_steps.

state_gradients_are_stopped  logical indicating that the proposed new state be run through tf$stop_gradient. This is particularly useful when combining optimization over samples from the HMC chain. Default value: FALSE (i.e., do not apply stop_gradient).

step_size_update_fn  Function taking current step_size (typically a tf$Variable) and kernel_results (typically collections$namedtuple) and returns updated step_size (Tensors). Default value: NULL (i.e., do not update step_size automatically).

seed  integer to seed the random number generator.

store_parameters_in_results  If TRUE, then step_size and num_leapfrog_steps are written to and read from eponymous fields in the kernel results objects returned from one_step and bootstrap_results. This allows wrapper kernels to adjust those parameters on the fly. This is incompatible with step_size_update_fn, which must be set to NULL.

name  string prefixed to Ops created by this function. Default value: NULL (i.e., 'hmc_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(current_state). (The number of independent chains is tf$size(target_log_prob_fn(current_state)).)

Value

a Monte Carlo sampling kernel

References

mcmc_metropolis_adjusted_langevin_algorithm

RUNS ONE STEP OF METROPOLIS-ADJUSTED LANGEVIN ALGORITHM.

DESCRIPTION

Metropolis-adjusted Langevin algorithm (MALA) is a Markov chain Monte Carlo (MCMC) algorithm that takes a step of a discretised Langevin diffusion as a proposal. This class implements one step of MALA using Euler-Maruyama method for a given current_state and diagonal preconditioning volatility matrix.

USAGE

mcmc_metropolis_adjusted_langevin_algorithm(
    target_log_prob_fn,
    step_size,
    volatility_fn = NULL,
    seed = NULL,
    parallel_iterations = 10,
    name = NULL
)

ARGUMENTS

target_log_prob_fn
Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it's often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

volatility_fn
Function which takes an argument like current_state (or *current_state if it's a list) and returns volatility value at current_state. Should return a Tensor or list of Tensors that must broadcast with the shape of current_state. Defaults to the identity function.

seed
integer to seed the random number generator.
**parallel_iterations**
the number of coordinates for which the gradients of the volatility matrix \( \text{volatility}_\text{fn} \) can be computed in parallel.

**name**
String prefixed to Ops created by this function. Default value: NULL (i.e., ‘mala_kernel’).

**Details**

Mathematical details and derivations can be found in Roberts and Rosenthal (1998) and Xifara et al. (2013).

The \texttt{one_step} function can update multiple chains in parallel. It assumes that all leftmost dimensions of \texttt{current_state} index independent chain states (and are therefore updated independently).

The output of \texttt{target_log_prob_fn(current_state)} should reduce log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, \texttt{current_state[0,:]} could have a different target distribution from \texttt{current_state[1,:]}.

These semantics are governed by \texttt{target_log_prob_fn(current_state)}. (The number of independent chains is \texttt{tf.size(target_log_prob_fn(current_state))}.)

**References**


**See Also**

Other \texttt{mcmc_kernels}: \texttt{mcmc_dual_averaging_step_size_adaptation()}, \texttt{mcmc_hamiltonian_monte_carlo()}, \texttt{mcmc_metropolis_hastings()}, \texttt{mcmc_no_u_turn_sampler()}, \texttt{mcmc_random_walk_metropolis()}, \texttt{mcmc_replica_exchange_mc()}, \texttt{mcmc_simple_step_size_adaptation()}, \texttt{mcmc_slice_sampler()}, \texttt{mcmc_transformed_transition_kernel()}, \texttt{mcmc_uncalibrated_hamiltonian_monte_carlo()}, \texttt{mcmc_uncalibrated_langevin()}, \texttt{mcmc_uncalibrated_random_walk()}

---

**\texttt{mcmc_metropolis_hastings}**

*Runs one step of the Metropolis-Hastings algorithm.*

---

**Description**

The Metropolis-Hastings algorithm is a Markov chain Monte Carlo (MCMC) technique which uses a proposal distribution to eventually sample from a target distribution.

**Usage**

\texttt{mcmc_metropolis_hastings(inner_kernel, seed = NULL, name = NULL)}
Arguments

inner_kernel  TransitionKernel-like object which has collections$namedtuple kernel_results and which contains a target_log_prob member and optionally a log_acceptance_correction member.

seed  integer to seed the random number generator.

name  string prefixed to Ops created by this function. Default value: NULL (i.e., "mh_kernel").

Details

Note: inner_kernel$one_step must return kernel_results as a collections$namedtuple which must:

- have a target_log_prob field,
- optionally have a log_acceptance_correction field, and,
- have only fields which are Tensor-valued.

The Metropolis-Hastings log acceptance-probability is computed as:

\[
\log_{\text{accept}} = (\text{current\_kernel\_results.target\_log\_prob} - \text{previous\_kernel\_results.target\_log\_prob} + \text{current\_kernel\_results.log\_acceptance\_correction})
\]

If current_kernel_results$log_acceptance_correction does not exist, it is presumed 0 (i.e., that the proposal distribution is symmetric). The most common use-case for log_acceptance_correction is in the Metropolis-Hastings algorithm, i.e.,

\[
\text{accept}_\text{prob}(x' | x) = p(x') / p(x) (g(x|x') / g(x'|x))
\]

where,

\(p\) represents the target distribution,

\(g\) represents the proposal (conditional) distribution,

\(x'\) is the proposed state, and,

\(x\) is current state

The log of the parenthetical term is the log_acceptance_correction. The log_acceptance_correction may not necessarily correspond to the ratio of proposal distributions, e.g, log_acceptance_correction has a different interpretation in Hamiltonian Monte Carlo.

Value

a Monte Carlo sampling kernel

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_no_u_turn_sampler(), mcmc_random_walk_monte_carlo(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
mcmc_no_u_turn_sampler

Runs one step of the No U-Turn Sampler

Description

The No U-Turn Sampler (NUTS) is an adaptive variant of the Hamiltonian Monte Carlo (HMC) method for MCMC. NUTS adapts the distance traveled in response to the curvature of the target density. Conceptually, one proposal consists of reversibly evolving a trajectory through the sample space, continuing until that trajectory turns back on itself (hence the name, 'No U-Turn'). This class implements one random NUTS step from a given current_state. Mathematical details and derivations can be found in Hoffman & Gelman (2011).

Usage

mcmc_no_u_turn_sampler(
  target_log_prob_fn,
  step_size,
  max_tree_depth = 10,
  max_energy_diff = 1000,
  unrolled_leapfrog_steps = 1,
  seed = NULL,
  name = NULL
)

Arguments

target_log_prob_fn
  function which takes an argument like current_state and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

max_tree_depth
  Maximum depth of the tree implicitly built by NUTS. The maximum number of leapfrog steps is bounded by 2**max_tree_depth i.e. the number of nodes in a binary tree max_tree_depth nodes deep. The default setting of 10 takes up to 1024 leapfrog steps.

max_energy_diff
  Scaler threshold of energy differences at each leapfrog. divergence samples are defined as leapfrog steps that exceed this threshold. Default to 1000.

unrolled_leapfrog_steps
  The number of leapfrogs to unroll per tree expansion step. Applies a direct linear multiplier to the maximum trajectory length implied by max_tree_depth. Defaults to 1.
mcmc_no_u_turn_sampler

seed  integer to seed the random number generator.
name  name prefixed to Ops created by this function. Default value: NULL (i.e., 'nuts_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that a prefix of leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0][0,...] could have a different target distribution from current_state[0][1,...]. These semantics are governed by target_log_prob_fn(*current_state). (The number of independent chains is tf$size(target_log_prob_fn(current_state))).

Value

a Monte Carlo sampling kernel

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

Examples

predictors <- tf$cast( c(201,244, 47,287,203,58,210,202,198,158,165,201,157, 131,166,160,186,125,218,146),tf$float32)
obs <- tf$cast(c(592,401,583,402,495,173,479,504,510,416,393,442,317,311,400, 337,423,334,533,344),tf$float32)
y_sigma <- tf$cast(c(61,25,38,15,21,15,27,14,30,16,14,25,52,16,34,31,42,26, 16,22),tf$float32)

# Robust linear regression model
robust_lm <- tfd_joint_distribution_sequential(
list(
  tfd_normal(loc = 0, scale = 1, name = "b0"),
  tfd_normal(loc = 0, scale = 1, name = "b1"),
  tfd_half_normal(5, name = "df"),
  function(df, b1, b0)
    tfd_independent(
      tfd_student_t(
        # Likelihood
        df = tf$expand_dims(df, axis = -1L),
        df = tf$cast(c(592,401,583,402,495,173,479,504,510,416,393,442,317,311,400, 337,423,334,533,344),tf$float32)
      ))
    )
  ))

robust_lm <- tfd_joint_distribution_sequential(
list(
  tfd_normal(loc = 0, scale = 1, name = "b0"),
  tfd_normal(loc = 0, scale = 1, name = "b1"),
  tfd_half_normal(5, name = "df"),
  function(df, b1, b0)
    tfd_independent(
      tfd_student_t(
        # Likelihood
        df = tf$expand_dims(df, axis = -1L),
        df = tf$cast(c(592,401,583,402,495,173,479,504,510,416,393,442,317,311,400, 337,423,334,533,344),tf$float32)
      ))
    )
  ))

# Robust linear regression model
robust_lm <- tfd_joint_distribution_sequential(
list(
  tfd_normal(loc = 0, scale = 1, name = "b0"),
  tfd_normal(loc = 0, scale = 1, name = "b1"),
  tfd_half_normal(5, name = "df"),
  function(df, b1, b0)
    tfd_independent(
      tfd_student_t(
        # Likelihood
        df = tf$expand_dims(df, axis = -1L),
        df = tf$cast(c(592,401,583,402,495,173,479,504,510,416,393,442,317,311,400, 337,423,334,533,344),tf$float32)
      ))
    )
  ))
```r
mcmc_no_u_turn_sampler

loc = tf$expand_dims(b0, axis = -1L) +
    tf$expand_dims(b1, axis = -1L) * predictors[tf$newaxis, ],
scale = y_sigma,
name = "st"
}, name = "ind"), validate_args = TRUE)

log_prob <- function(b0, b1, df) {robust_lm %>%
    tfd_log_prob(list(b0, b1, df, obs))
step_size0 <- Map(function(x) tf$cast(x, tf$float32), c(1, .2, .5))

number_of_steps <- 10
burnin <- 5
nchain <- 50

run_chain <- function() {
  # random initialization of the starting position of each chain
  samples <- robust_lm %>% tfd_sample(nchain)
b0 <- samples[[1]]
b1 <- samples[[2]]
df <- samples[[3]]

  # bijector to map constrained parameters to real
  unconstraining_bijectors <- list(
    tfb_identity(), tfb_identity(), tfb_exp())

  trace_fn <- function(x, pkr) {
    list(pkr$inner_results$inner_results$step_size,
         pkr$inner_results$inner_results$log_accept_ratio)
  }

  nuts <- mcmc_no_u_turn_sampler(
    target_log_prob_fn = log_prob,
    step_size = step_size0
  ) %>%
    mcmc_transformed_transition_kernel(bijector = unconstraining_bijectors) %>%
    mcmc_dual_averaging_step_size_adaptation(
      num_adaptation_steps = burnin,
      step_size_setter_fn = function(pkr, new_step_size)
        pkr$'\_replace'
          (inner_results = pkr$inner_results$'\_replace'('step_size = new_step_size)),
      step_size_getter_fn = function(pkr) pkr$inner_results$step_size,
      log_accept_prob_getter_fn = function(pkr) pkr$inner_results$log_accept_ratio
    )

  nuts %>%
    mcmc_sample_chain(
      num_results = number_of_steps,
      num_burnin_steps = burnin,
      current_state = list(b0, b1, df),
      trace_fn = trace_fn
    )
}

run_chain <- tensorflow::tf_function(run_chain)
res <- run_chain()
```
mcmc_potential_scale_reduction

Gelman and Rubin (1992)'s potential scale reduction for chain convergence.

Description

Given \( N > 1 \) states from each of \( C > 1 \) independent chains, the potential scale reduction factor, commonly referred to as R-hat, measures convergence of the chains (to the same target) by testing for equality of means.

Usage

```r
mcmc_potential_scale_reduction(
  chains_states,
  independent_chain_ndims = 1,
  name = NULL
)
```

Arguments

- `chains_states` Tensor or list of Tensors representing the state(s) of a Markov Chain at each result step. The \( i \)th state is assumed to have shape \([Ni, Ci1, Ci2,...,CiD] + A\). Dimension 0 indexes the \( Ni > 1 \) result steps of the Markov Chain. Dimensions 1 through \( D \) index the \( Ci1 \times ... \times CiD \) independent chains to be tested for convergence to the same target. The remaining dimensions, \( A \), can have any shape (even empty).
- `independent_chain_ndims` Integer type Tensor with value \( \geq 1 \) giving the number of dimensions, from \( \text{dim} = 1 \) to \( \text{dim} = D \), holding independent chain results to be tested for convergence.
- `name` name to prepend to created tf. Default: `potential_scale_reduction`.

Details

Specifically, R-hat measures the degree to which variance (of the means) between chains exceeds what one would expect if the chains were identically distributed. See Gelman and Rubin (1992), Brooks and Gelman (1998).

Some guidelines:

- The initial state of the chains should be drawn from a distribution overdispersed with respect to the target.
- If all chains converge to the target, then as \( N \to \infty \), R-hat \( \to 1 \). Before that, R-hat > 1 (except in pathological cases, e.g. if the chain paths were identical).
The above holds for any number of chains \( C > 1 \). Increasing \( C \) improves effectiveness of the diagnostic.

- Sometimes, R-hat < 1.2 is used to indicate approximate convergence, but of course this is problem dependent. See Brooks and Gelman (1998).

- R-hat only measures non-convergence of the mean. If higher moments, or other statistics are desired, a different diagnostic should be used. See Brooks and Gelman (1998).

To see why R-hat is reasonable, let \( X \) be a random variable drawn uniformly from the combined states (combined over all chains). Then, in the limit \( N, C \rightarrow \infty \), with \( E, \text{Var} \) denoting expectation and variance, R-hat = \( ( E[\text{Var}[X \mid \text{chain}]] + \text{Var}[E[X \mid \text{chain}]] ) / E[\text{Var}[X \mid \text{chain}]] \). Using the law of total variance, the numerator is the variance of the combined states, and the denominator is the total variance minus the variance of the the individual chain means. If the chains are all drawing from the same distribution, they will have the same mean, and thus the ratio should be one.

**Value**

Tensor or list of Tensors representing the R-hat statistic for the state(s). Same dtype as state, and shape equal to state$shape[1 + independent\_chain\_ndims:]$.

**References**


**See Also**

Other mcmc_functions: `mcmc_effective_sample_size()`, `mcmc_sample_annealed_importance_chain()`, `mcmc_sample_chain()`, `mcmc_sample_halton_sequence()`

---

**Description**

Random Walk Metropolis is a gradient-free Markov chain Monte Carlo (MCMC) algorithm. The algorithm involves a proposal generating step \( \text{proposal\_state} = \text{current\_state} + \text{perturb} \) by a random perturbation, followed by Metropolis-Hastings accept/reject step. For more details see Section 2.1 of Roberts and Rosenthal (2004).
mcmc_random_walk_metropolis

Usage

mcmc_random_walk_metropolis(
  target_log_prob_fn,
  new_state_fn = NULL,
  seed = NULL,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state ((if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

new_state_fn
  Function which takes a list of state parts and a seed; returns a same-type list of Tensors, each being a perturbation of the input state parts. The perturbation distribution is assumed to be a symmetric distribution centered at the input state part. Default value: NULL which is mapped to tfp$mcmc$random_walk_normal_fn().

seed
  integer to seed the random number generator.

name
  String name prefixed to Ops created by this function. Default value: NULL (i.e., 'rwm_kernel').

Details

The current class implements RWM for normal and uniform proposals. Alternatively, the user can supply any custom proposal generating function. The function one_step can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(current_state). (The number of independent chains is tf$size(target_log_prob_fn(current_state)).)

Value

a Monte Carlo sampling kernel

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()
Replica Exchange Monte Carlo is a Markov chain Monte Carlo (MCMC) algorithm that is also known as Parallel Tempering. This algorithm performs multiple sampling with different temperatures in parallel, and exchanges those samplings according to the Metropolis-Hastings criterion. The \( K \) replicas are parameterized in terms of inverse temperature's, \((\text{beta}[0], \text{beta}[1], ..., \text{beta}[K-1])\). If the target distribution has probability density \( p(x) \), the \( k \)th replica has density \( p(x)^{\text{beta}_k} \).

### Usage

```r
mcmc_replica_exchange_mc(
  target_log_prob_fn,
  inverse_temperatures,
  make_kernel_fn,
  swap_proposal_fn = tfp$mcmc$replica_exchange_mc$default_swap_proposal_fn(1),
  state_includes_replicas = FALSE,
  seed = NULL,
  name = NULL
)
```

### Arguments

- **target_log_prob_fn**: Function which takes an argument like `current_state` (if it's a list `current_state` will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.
- **inverse_temperatures**: 1D Tensor of inverse temperatures to perform samplings with each replica. Must have statically known shape. \( \text{inverse_temperatures}[0] \) produces the states returned by samplers, and is typically == 1.
- **make_kernel_fn**: Function which takes target_log_prob_fn and seed args and returns a TransitionKernel instance.
- **swap_proposal_fn**: function which take a number of replicas, and return combinations of replicas for exchange.
- **state_includes_replicas**: Boolean indicating whether the leftmost dimension of each state sample should index replicas. If TRUE, the leftmost dimension of the `current_state` kwarg to `tfp.mcmc.sample_chain` will be interpreted as indexing replicas.
- **seed**: integer to seed the random number generator.
- **name**: string prefixed to Ops created by this function. Default value: NULL (i.e., "remc_kernel").
mcmc_sample_annealed_importance_chain

Details

Typically beta[0] = 1.0, and 1.0 > beta[1] > beta[2] > ... > 0.0.

- beta[0] == 1 ==> First replicas samples from the target density, p.
- beta[k] < 1, for k = 1, ..., K-1 ==> Other replicas sample from "flattened" versions of p (peak is less high, valley less low). These distributions are somewhat closer to a uniform on the support of p. Samples from adjacent replicas i, i + 1 are used as proposals for each other in a Metropolis step. This allows the lower beta samples, which explore less dense areas of p, to occasionally be used to help the beta == 1 chain explore new regions of the support. Samples from replica 0 are returned, and the others are discarded.

Value

list of next_state (Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as and current_state.) and kernel_results (collections$namedtuple of internal calculations used to advance the chain).

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

mcmc_sample_annealed_importance_chain

Runs annealed importance sampling (AIS) to estimate normalizing constants.

Description

This function uses an MCMC transition operator (e.g., Hamiltonian Monte Carlo) to sample from a series of distributions that slowly interpolates between an initial "proposal" distribution: \( \exp(\text{proposal\_log\_prob\_fn}(x) - \text{proposal\_log\_normalizer}) \) and the target distribution: \( \exp(\text{target\_log\_prob\_fn}(x) - \text{target\_log\_normalizer}) \), accumulating importance weights along the way. The product of these importance weights gives an unbiased estimate of the ratio of the normalizing constants of the initial distribution and the target distribution: \( \mathbb{E}[\exp(\text{ais\_weights})] = \exp(\text{target\_log\_normalizer} - \text{proposal\_log\_normalizer}) \).

Usage

mcmc_sample_annealed_importance_chain(
    num_steps,
    proposal_log_prob_fn,
    target_log_prob_fn,
    current_state,
    make_kernel_fn,
```
mcmc_sample_annealed_importance_chain

parallel_iterations = 10,
name = NULL
)

Arguments

num_steps Integer number of Markov chain updates to run. More iterations means more
expense, but smoother annealing between q and p, which in turn means exponen-
tially lower variance for the normalizing constant estimator.

proposal_log_prob_fn function that returns the log density of the initial distribution.

target_log_prob_fn function which takes an argument like current_state and returns its (possibly
unnormalized) log-density under the target distribution.

current_state Tensor or list of Tensors representing the current state(s) of the Markov
chain(s). The first r dimensions index independent chains, r = tf$rank(target_log_prob_fn(current_state))

make_kernel_fn function which returns a TransitionKernel-like object. Must take one argu-
ment representing the TransitionKernel's target_log_prob_fn. The target_log_prob_fn
argument represents the TransitionKernel's target log distribution. Note:
sample_annealed_importance_chain creates a new target_log_prob_fn which
is an interpolation between the supplied target_log_prob_fn and proposal_log_prob_fn;
it is this interpolated function which is used as an argument to make_kernel_fn.

parallel_iterations The number of iterations allowed to run in parallel. It must be a positive integer.
See tf$while_loop for more details.

name string prefixed to Ops created by this function. Default value: NULL (i.e., "sam-
ple_annealed_importance_chain").

Details

Note: When running in graph mode, proposal_log_prob_fn and target_log_prob_fn are called
exactly three times (although this may be reduced to two times in the future).

Value

list of next_state (Tensor or Python list of Tensors representing the state(s) of the Markov
chain(s) at the final iteration. Has same shape as input current_state), ais_weights (Tensor
with the estimated weight(s). Has shape matching target_log_prob_fn(current_state)), and
kernel_results (collections.namedtuple of internal calculations used to advance the chain).

See Also

For an example how to use see `mcmc_sample_chain()

Other mcmc_functions: `mcmc_effective_sample_size()`, `mcmc_potential_scale_reduction()`,
`mcmc_sample_chain()`, `mcmc_sample_halton_sequence()`
mcmc_sample_chain

Implements Markov chain Monte Carlo via repeated TransitionKernel steps.

Description

This function samples from an Markov chain at current_state and whose stationary distribution is governed by the supplied TransitionKernel instance (kernel).

Usage

```r
mcmc_sample_chain(
  kernel = NULL,
  num_results,
  current_state,
  previous_kernel_results = NULL,
  num_burnin_steps = 0,
  num_steps_between_results = 0,
  trace_fn = NULL,
  return_final_kernel_results = FALSE,
  parallel_iterations = 10,
  seed = NULL,
  name = NULL
)
```

Arguments

- **kernel**: An instance of `tfp$mcmc$TransitionKernel` which implements one step of the Markov chain.
- **num_results**: Integer number of Markov chain draws.
- **current_state**: Tensor or list of Tensors representing the current state(s) of the Markov chain(s).
- **previous_kernel_results**: A Tensor or a nested collection of Tensors representing internal calculations made within the previous call to this function (or as returned by `bootstrap_results`).
- **num_burnin_steps**: Integer number of chain steps to take before starting to collect results. Default value: 0 (i.e., no burn-in).
- **num_steps_between_results**: Integer number of chain steps between collecting a result. Only one out of every `num_steps_between_samples + 1` steps is included in the returned results. The number of returned chain states is still equal to `num_results`. Default value: 0 (i.e., no thinning).
- **trace_fn**: A function that takes in the current chain state and the previous kernel results and return a Tensor or a nested collection of Tensors that is then traced along with the chain state.
mcmc_sample_chain

return_final_kernel_results
   If TRUE, then the final kernel results are returned alongside the chain state and the trace specified by the trace_fn.

parallel_iterations
   The number of iterations allowed to run in parallel. It must be a positive integer. See tf$while_loop for more details.

seed
   Optional, a seed for reproducible sampling.

name
   string prefixed to Ops created by this function. Default value: NULL, (i.e., "mcmc_sample_chain").

Details

This function can sample from multiple chains, in parallel. (Whether or not there are multiple chains is dictated by the kernel.)

The current_state can be represented as a single Tensor or a list of Tensors which collectively represent the current state. Since MCMC states are correlated, it is sometimes desirable to produce additional intermediate states, and then discard them, ending up with a set of states with decreased autocorrelation. See Owen (2017). Such “thinning” is made possible by setting num_steps_between_results > 0. The chain then takes num_steps_between_results extra steps between the steps that make it into the results. The extra steps are never materialized (in calls to sess$run), and thus do not increase memory requirements.

Warning: when setting a seed in the kernel, ensure that sample_chain's parallel_iterations=1, otherwise results will not be reproducible. In addition to returning the chain state, this function supports tracing of auxiliary variables used by the kernel. The traced values are selected by specifying trace_fn. By default, all kernel results are traced but in the future the default will be changed to no results being traced, so plan accordingly. See below for some examples of this feature.

Value

list of:

• checkpointable_states_and_trace: if return_final_kernel_results is TRUE. The return value is an instance of CheckpointableStatesAndTrace.

• all_states: if return_final_kernel_results is FALSE and trace_fn is NULL. The return value is a Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as input current_state but with a prepended num_results-size dimension.

• states_and_trace: if return_final_kernel_results is FALSE and trace_fn is not NULL. The return value is an instance of StatesAndTrace.

References


See Also

Other mcmc_functions: mcmc_effective_sample_size(), mcmc_potential_scale_reduction(), mcmc_sample_annealed_importance_chain(), mcmc_sample_halton_sequence()
Examples

dims <- 10
true_stddev <- sqrt(seq(1, 3, length.out = dims))
likelihood <- tfd_multivariate_normal_diag(scale_diag = true_stddev)

kernel <- mcmc_hamiltonian_monte_carlo(
  target_log_prob_fn = likelihood$log_prob,
  step_size = 0.5,
  num_leapfrog_steps = 2
)

states <- kernel %>% mcmc_sample_chain(
  num_results = 1000,
  num_burnin_steps = 500,
  current_state = rep(0, dims),
  trace_fn = NULL
)

sample_mean <- tf$reduce_mean(states, axis = 0L)
sample_stddev <- tf$sqrt(
  tf$reduce_mean(tf$math$squared_difference(states, sample_mean), axis = 0L))

mcmc_sample_halton_sequence

Returns a sample from the dim dimensional Halton sequence.

Description

Warning: The sequence elements take values only between 0 and 1. Care must be taken to appropriately transform the domain of a function if it differs from the unit cube before evaluating integrals using Halton samples. It is also important to remember that quasi-random numbers without randomization are not a replacement for pseudo-random numbers in every context. Quasi random numbers are completely deterministic and typically have significant negative autocorrelation unless randomization is used.

Usage

mcmc_sample_halton_sequence(
  dim,
  num_results = NULL,
  sequence_indices = NULL,
  dtype = tf$float32,
  randomized = TRUE,
  seed = NULL,
  name = NULL
)
Arguments

- **dim** (Positive integer representing each sample's event size. Must not be greater than 1000.)
- **num_results** (Optional) Positive scalar Tensor of dtype int32. The number of samples to generate. Either this parameter or sequence_indices must be specified but not both. If this parameter is None, then the behaviour is determined by the sequence_indices. Default value: NULL.
- **sequence_indices** (Optional) Tensor of dtype int32 and rank 1. The elements of the sequence to compute specified by their position in the sequence. The entries index into the Halton sequence starting with 0 and hence, must be whole numbers. For example, sequence_indices=[0, 5, 6] will produce the first, sixth and seventh elements of the sequence. If this parameter is None, then the num_results parameter must be specified which gives the number of desired samples starting from the first sample. Default value: NULL.
- **dtype** (Optional) The dtype of the sample. One of: float16, float32 or float64. Default value: tf$float32.
- **randomized** (Optional) bool indicating whether to produce a randomized Halton sequence. If TRUE, applies the randomization described in Owen (2017). Default value: TRUE.
- **seed** (Optional) integer to seed the random number generator. Only used if randomized is TRUE. If not supplied and randomized is TRUE, no seed is set. Default value: NULL.
- **name** (Optional) string describing ops managed by this function. If not supplied the name of this function is used. Default value: "sample_halton_sequence".

Details

Computes the members of the low discrepancy Halton sequence in dimension dim. The dim-dimensional sequence takes values in the unit hypercube in dim dimensions. Currently, only dimensions up to 1000 are supported. The prime base for the k-th axes is the k-th prime starting from 2. For example, if dim = 3, then the bases will be [2, 3, 5] respectively and the first element of the non-randomized sequence will be: [0.5, 0.333, 0.2]. For a more complete description of the Halton sequences see here. For low discrepancy sequences and their applications see here.

If randomized is true, this function produces a scrambled version of the Halton sequence introduced by Owen (2017). For the advantages of randomization of low discrepancy sequences see here.

The number of samples produced is controlled by the num_results and sequence_indices parameters. The user must supply either num_results or sequence_indices but not both. The former is the number of samples to produce starting from the first element. If sequence_indices is given instead, the specified elements of the sequence are generated. For example, sequence_indices=tf$range(10) is equivalent to specifying n=10.

Value

halton_elements Elements of the Halton sequence. Tensor of supplied dtype and shape [num_results, dim] if num_results was specified or shape [s, dim] where s is the size of sequence_indices if sequence_indices were specified.
**mcmc_simple_step_size_adaptation**

Adapts the inner kernel’s step size based on log_accept_prob.

### Description

The simple policy multiplicatively increases or decreases the step size of the inner kernel based on the value of log_accept_prob. It is based on equation 19 of Andrieu and Thoms (2008). Given enough steps and small enough adaptation_rate the median of the distribution of the acceptance probability will converge to the target_accept_prob. A good target acceptance probability depends on the inner kernel. If this kernel is HamiltonianMonteCarlo, then 0.6-0.9 is a good range to aim for. For RandomWalkMetropolis this should be closer to 0.25. See the individual kernels’ docstrings for guidance.

### Usage

```r
mcmc_simple_step_size_adaptation(
  inner_kernel,
  num_adaptation_steps,
  target_accept_prob = 0.75,
  adaptation_rate = 0.01,
  step_size_setter_fn = NULL,
  step_size_getter_fn = NULL,
  log_accept_prob_getter_fn = NULL,
  validate_args = FALSE,
  name = NULL
)
```

### Arguments

- **inner_kernel** TransitionKernel-like object.
- **num_adaptation_steps** Scalar integer Tensor number of initial steps to during which to adjust the step size. This may be greater, less than, or equal to the number of burnin steps.
- **target_accept_prob** A floating point Tensor representing desired acceptance probability. Must be a positive number less than 1. This can either be a scalar, or have shape list(num_chains). Default value: `0.75` (the center of asymptotically optimal rate for HMC).

### References


### See Also

For an example how to use see `mcmc_sample_chain()`. Other `mcmc_functions`: `mcmc_effective_sample_size()`, `mcmc_potential_scale_reduction()`, `mcmc_sample_annealed_importance_chain()`, `mcmc_sample_chain()`
adaptation_rate

Tensor representing amount to scale the current step_size.

step_size_setter_fn

A function with the signature (kernel_results, new_step_size) -> new_kernel_results where kernel_results are the results of the inner_kernel, new_step_size is a Tensor or a nested collection of Tensors with the same structure as returned by the step_size_getter_fn, and new_kernel_results are a copy of kernel_results with the step size(s) set.

step_size_getter_fn

A function with the signature (kernel_results) -> step_size where kernel_results are the results of the inner_kernel, and step_size is a floating point Tensor or a nested collection of such Tensors.

log_accept_prob_getter_fn

A function with the signature (kernel_results) -> log_accept_prob where kernel_results are the results of the inner_kernel, and log_accept_prob is a floating point Tensor. log_accept_prob can either be a scalar, or have shape list(num_chains). If it's the latter, step_size should also have the same leading dimension.

validate_args Logical. When True kernel parameters are checked for validity. When False invalid inputs may silently render incorrect outputs.

name string prefixed to Ops created by this class. Default: "simple_step_size_adaptation".

Details

In general, adaptation prevents the chain from reaching a stationary distribution, so obtaining consistent samples requires num_adaptation_steps be set to a value somewhat smaller than the number of burnin steps. However, it may sometimes be helpful to set num_adaptation_steps to a larger value during development in order to inspect the behavior of the chain during adaptation.

The step size is assumed to broadcast with the chain state, potentially having leading dimensions corresponding to multiple chains. When there are fewer of those leading dimensions than there are chain dimensions, the corresponding dimensions in the log_accept_prob are averaged (in the direct space, rather than the log space) before being used to adjust the step size. This means that this kernel can do both cross-chain adaptation, or per-chain step size adaptation, depending on the shape of the step size.

For example, if your problem has a state with shape [S], your chain state has shape [C0, C1, Y] (meaning that there are C0 * C1 total chains) and log_accept_prob has shape [C0, C1] (one acceptance probability per chain), then depending on the shape of the step size, the following will happen:

• Step size has shape [], [S] or [1], the log_accept_prob will be averaged across its C0 and C1 dimensions. This means that you will learn a shared step size based on the mean acceptance probability across all chains. This can be useful if you don’t have a lot of steps to adapt and want to average away the noise.

• Step size has shape [C1, 1] or [C1, S], the log_accept_prob will be averaged across its C0 dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the C1 dimension. This can be useful when the C1 dimension indexes different distributions, while C0 indexes replicas of a single distribution, all sampled in parallel.
mcmc_simple_step_size_adaptation

- Step size has shape \([C0, C1, 1]\) or \([C0, C1, S]\), then no averaging will happen. This means that each chain will learn its own step size. This can be useful when all chains are sampling from different distributions. Even when all chains are for the same distribution, this can help during the initial warmup period.

- Step size has shape \([C0, 1, 1]\) or \([C0, 1, S]\), the \(\log\_\text{accept\_prob}\) will be averaged across its \(C1\) dimension. This means that you will learn a shared step size based on the mean acceptance probability across chains that share the coordinate across the \(C0\) dimension. This can be useful when the \(C0\) dimension indexes different distributions, while \(C1\) indexes replicas of a single distribution, all sampled in parallel.

Value

A Monte Carlo sampling kernel

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_slice_sampler(), mcmc_transformed_transformation_mcmc(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk().

Examples

```r
target_log_prob_fn <- tfd_normal(loc = 0, scale = 1)$log_prob
num_burnin_steps <- 500
num_results <- 500
num_chains <- 64L
step_size <- tf$fill(list(num_chains), 0.1)

kernel <- mcmc_hamiltonian_monte_carlo(
  target_log_prob_fn = target_log_prob_fn,
  num_leapfrog_steps = 2,
  step_size = step_size
)

mcmc_simple_step_size_adaptation(num_adaptation_steps = round(num_burnin_steps * 0.8))

res <- kernel %>% mcmc_sample_chain(
  num_results = num_results,
  num_burnin_steps = num_burnin_steps,
  current_state = rep(0, num_chains),
  trace_fn = function(x, pkr) {
    list (}
```
mcmc_slice_sampler

Runs one step of the slice sampler using a hit and run approach

Description

Slice Sampling is a Markov Chain Monte Carlo (MCMC) algorithm based, as stated by Neal (2003), on the observation that "...one can sample from a distribution by sampling uniformly from the region under the plot of its density function. A Markov chain that converges to this uniform distribution can be constructed by alternately uniform sampling in the vertical direction with uniform sampling from the horizontal slice defined by the current vertical position, or more generally, with some update that leaves the uniform distribution over this slice invariant". Mathematical details and derivations can be found in Neal (2003). The one dimensional slice sampler is extended to n-dimensions through use of a hit-and-run approach: choose a random direction in n-dimensional space and take a step, as determined by the one-dimensional slice sampling algorithm, along that direction (Belisle et al. 1993).

Usage

mcmc_slice_sampler(
  target_log_prob_fn,
  step_size,
  max_doublings,
  seed = NULL,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.
mcmc_transformed_transition_kernel

max_doublings  Scalar positive int32 tf$Tensor. The maximum number of doublings to consider.
seed           integer to seed the random number generator.
name           string prefixed to Ops created by this function. Default value: NULL (i.e., 'slice_sampler_kernel').

Details

The one_step function can update multiple chains in parallel. It assumes that all leftmost dimensions of current_state index independent chain states (and are therefore updated independently). The output of target_log_prob_fn(*current_state) should sum log-probabilities across all event dimensions. Slices along the rightmost dimensions may have different target distributions; for example, current_state[0, :] could have a different target distribution from current_state[1, :]. These semantics are governed by target_log_prob_fn(*current_state). (The number of independent chains is tf$size(target_log_prob_fn(*current_state)).)

Note that the sampler only supports states where all components have a common dtype.

Value

list of next_state (Tensor or Python list of Tensors representing the state(s) of the Markov chain(s) at each result step. Has same shape as and current_state) and kernel_results (collections$namedtuple of internal calculations used to 'advance the chain).

References


See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_adjusted_langevin_algorithm(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin(), mcmc_uncalibrated_random_walk()

mcmc_transformed_transition_kernel

Applies a bijector to the MCMC’s state space

Description

The transformed transition kernel enables fitting a bijector which serves to decorrelate the Markov chain Monte Carlo (MCMC) event dimensions thus making the chain mix faster. This is particularly useful when the geometry of the target distribution is unfavorable. In such cases it may take many evaluations of target_log_prob_fn for the chain to mix between faraway states.
mcmc_transformed_transition_kernel

Usage

mcmc_transformed_transition_kernel(inner_kernel, bijector, name = NULL)

Arguments

inner_kernel  TransitionKernel-like object which has a target_log_prob_fn argument.
bijector      bijector or list of bijectors. These bijectors use forward to map the inner_kernel state space to the state expected by inner_kernel$target_log_prob_fn.
name          string prefixed to Ops created by this function. Default value: NULL (i.e., "transformed_kernel").

Details

The idea of training an affine function to decorrelate chain event dims was presented in Parno and Marzouk (2014). Used in conjunction with the Hamiltonian Monte Carlo transition kernel, the Parno and Marzouk (2014) idea is an instance of Riemannian manifold HMC (Girolami and Calderhead, 2011).

The transformed transition kernel enables arbitrary bijective transformations of arbitrary transition kernels, e.g., one could use bijectors tfb_affine, tfb_real_nvp, etc. with transition kernels mcmc_hamiltonian_monte_carlo, mcmc_random_walk_metropolis, etc.

Value

a Monte Carlo sampling kernel

References


See Also

mcmc_uncalibrated_hamiltonian_monte_carlo

Runs one step of Uncalibrated Hamiltonian Monte Carlo

Description

Warning: this kernel will not result in a chain which converges to the target_log_prob. To get a convergent MCMC, use mcmc_hamiltonian_monte_carlo(...) or mcmc_metropolis_hastings(mcmc_uncalibrated_hamiltonian_monte_carlo(...)). For more details on UncalibratedHamiltonianMonteCarlo, see HamiltonianMonteCarlo.

Usage

mcmc_uncalibrated_hamiltonian_monte_carlo(
  target_log_prob_fn,
  step_size,
  num_leapfrog_steps,
  state_gradients_are_stopped = FALSE,
  seed = NULL,
  store_parameters_in_results = FALSE,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it's often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.

num_leapfrog_steps
  Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to step_size * num_leapfrog_steps.

state_gradients_are_stopped
  logical indicating that the proposed new state be run through tf$stop_gradient. This is particularly useful when combining optimization over samples from the HMC chain. Default value: FALSE (i.e., do not apply stop_gradient).

seed
  integer to seed the random number generator.

store_parameters_in_results
  If TRUE, then step_size and num_leapfrog_steps are written to and read from eponymous fields in the kernel results objects returned from one_step and bootstrap_results. This allows wrapper kernels to adjust those parameters on the fly. This is incompatible with step_size_update_fn, which must be set to NULL.
Runs one step of Uncalibrated Langevin discretized diffusion.

Description

The class generates a Langevin proposal using _euler_method function and also computes helper UncalibratedLangevinKernelResults for the next iteration. Warning: this kernel will not result in a chain which converges to the target_log_prob. To get a convergent MCMC, use MetropolisAdjustedLangevinAlgorithm( UncalibratedLangevin(...)) or MetropolisHastings(UncalibratedLangevin(...)).

Usage

mcmc_uncalibrated_langevin(
  target_log_prob_fn,
  step_size,
  volatility_fn = NULL,
  parallel_iterations = 10,
  compute_acceptance = TRUE,
  seed = NULL,
  name = NULL
)

Arguments

target_log_prob_fn
  Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

step_size
  Tensor or list of Tensors representing the step size for the leapfrog integrator. Must broadcast with the shape of current_state. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. When possible, it’s often helpful to match per-variable step sizes to the standard deviations of the target distribution in each variable.
mcmc_uncalibrated_random_walk

Generate proposal for the Random Walk Metropolis algorithm.

Description

Warning: this kernel will not result in a chain which converges to the target_log_prob. To get a convergent MCMC, use mcmc_random_walk_metropolis(...) or mcmc_metropolis_hastings(mcmc_uncalibrated_random_walk(...))

Usage

mcmc_uncalibrated_random_walk(
    target_log_prob_fn,
    new_state_fn = NULL,
    seed = NULL,
    name = NULL
)
params_size_categorical_mixture_of_one_hot_categorical

Arguments

- target_log_prob_fn
  Function which takes an argument like current_state (if it's a list current_state will be unpacked) and returns its (possibly unnormalized) log-density under the target distribution.

- new_state_fn
  Function which takes a list of state parts and a seed; returns a same-type list of Tensors, each being a perturbation of the input state parts. The perturbation distribution is assumed to be a symmetric distribution centered at the input state part. Default value: NULL which is mapped to tfp$mcmc$random_walk_normal_fn().

- seed
  integer to seed the random number generator.

- name
  String name prefixed to Ops created by this function. Default value: NULL (i.e., 'rwm_kernel').

Value

a Monte Carlo sampling kernel

See Also

Other mcmc_kernels: mcmc_dual_averaging_step_size_adaptation(), mcmc_hamiltonian_monte_carlo(), mcmc_metropolis_hastings(), mcmc_no_u_turn_sampler(), mcmc_random_walk_metropolis(), mcmc_replica_exchange_mc(), mcmc_simple_step_size_adaptation(), mcmc_slice_sampler(), mcmc_transformed_transition_kernel(), mcmc_uncalibrated_hamiltonian_monte_carlo(), mcmc_uncalibrated_langevin()

Description

number of params needed to create a CategoricalMixtureOfOneHotCategorical distribution

Usage

params_size_categorical_mixture_of_one_hot_categorical(
  event_size,
  num_components
)

Arguments

- event_size: event size of this distribution
- num_components: number of components in the mixture
Value

a scalar

---

```
params_size_independent_bernoulli

number of params needed to create an IndependentBernoulli distribution
```

Description

number of params needed to create an IndependentBernoulli distribution

Usage

```
params_size_independent_bernoulli(event_size)
```

Arguments

```
event_size  event size of this distribution
```

Value

a scalar

---

```
params_size_independent_logistic

number of params needed to create an IndependentLogistic distribution
```

Description

number of params needed to create an IndependentLogistic distribution

Usage

```
params_size_independent_logistic(event_size)
```

Arguments

```
event_size  event size of this distribution
```

Value

a scalar
**params_size_independent_normal**

number of params needed to create an IndependentNormal distribution

**Description**

number of params needed to create an IndependentNormal distribution

**Usage**

params_size_independent_normal(event_size)

**Arguments**

- event_size event size of this distribution

**Value**

- a scalar

---

**params_size_independent_poisson**

number of params needed to create an IndependentPoisson distribution

**Description**

number of params needed to create an IndependentPoisson distribution

**Usage**

params_size_independent_poisson(event_size)

**Arguments**

- event_size event size of this distribution

**Value**

- a scalar
**params_size_mixture_logistic**

*number of params needed to create a MixtureLogistic distribution*

**Description**

number of params needed to create a MixtureLogistic distribution

**Usage**

```
params_size_mixture_logistic(num_components, event_shape)
```

**Arguments**

- `num_components` Number of component distributions in the mixture distribution.
- `event_shape` Number of parameters needed to create a single component distribution.

**Value**

a scalar

---

**params_size_mixture_normal**

*number of params needed to create a MixtureNormal distribution*

**Description**

number of params needed to create a MixtureNormal distribution

**Usage**

```
params_size_mixture_normal(num_components, event_shape)
```

**Arguments**

- `num_components` Number of component distributions in the mixture distribution.
- `event_shape` Number of parameters needed to create a single component distribution.

**Value**

a scalar
\textit{params\_size\_mixture\_same\_family} \\
\textit{number of params needed to create a MixtureSameFamily distribution}

**Description**

number of params needed to create a MixtureSameFamily distribution

**Usage**

\texttt{params\_size\_mixture\_same\_family(num\_components, component\_params\_size)}

**Arguments**

- \texttt{num\_components} Number of component distributions in the mixture distribution.
- \texttt{component\_params\_size} Number of parameters needed to create a single component distribution.

**Value**

a scalar

\textit{params\_size\_multivariate\_normal\_tri\_l} \\
\textit{number of params needed to create a MultivariateNormalTriL distribution}

**Description**

number of params needed to create a MultivariateNormalTriL distribution

**Usage**

\texttt{params\_size\_multivariate\_normal\_tri\_l(event\_size)}

**Arguments**

- \texttt{event\_size} event size of this distribution

**Value**

a scalar
## params_size_one_hot_categorical

**number of params needed to create a OneHotCategorical distribution**

### Description

number of params needed to create a OneHotCategorical distribution

### Usage

```r
params_size_one_hot_categorical(event_size)
```

### Arguments

- `event_size` : event size of this distribution

### Value

a scalar

### sts_additive_state_space_model

**A state space model representing a sum of component state space models.**

### Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model` for details.

### Usage

```r
sts_additive_state_space_model(
  component_ssms,
  constant_offset = 0,
  observation_noise_scale = NULL,
  initial_state_prior = NULL,
  initial_step = 0,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)
```
The `sts_additive_state_space_model` represents a sum of component state space models. Each of the N components describes a random process generating a distribution on observed time series x1[t], x2[t], ..., xN[t]. The additive model represents the sum of these processes, y[t] = x1[t] + x2[t] + ... + xN[t] + eps[t], where eps[t] ~ N(0, observation_noise_scale) is an observation noise term.

**Details**

The `sts_additive_state_space_model` represents a sum of component state space models. Each of the N components describes a random process generating a distribution on observed time series x1[t], x2[t], ..., xN[t]. The additive model represents the sum of these processes, y[t] = x1[t] + x2[t] + ... + xN[t] + eps[t], where eps[t] ~ N(0, observation_noise_scale) is an observation noise term.

**Mathematical Details**

```python
component_ssms list containing one or more tfd_linear_gaussian_state_space_model instances. The components will in general implement different time-series models, with possibly different latent_size, but they must have the same dtype, event shape (num_timesteps and observation_size), and their batch shapes must broadcast to a compatible batch shape.

constant_offset scalar float tensor, or batch of scalars, specifying a constant value added to the sum of outputs from the component models. This allows the components to model the shifted series observed_time_series - constant_offset. Default value: 0.

observation_noise_scale Optional scalar float tensor indicating the standard deviation of the observation noise. May contain additional batch dimensions, which must broadcast with the batch shape of elements in component_ssms. If observation_noise_scale is specified for the sts_additive_state_space_model, the observation noise scales of component models are ignored. If NULL, the observation noise scale is derived by summing the noise variances of the component models, i.e., observation_noise_scale = sqrt(sum([ssm.observation_noise_scale**2 for ssm in component_ssms])).

initial_state_prior instance of tfd_multivariate_normal representing the prior distribution on latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model requires a rank-1 event shape).

initial_step Optional scalar integer tensor specifying the starting timestep. Default value: 0.

validate_args logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.

allow_nan_stats logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.

name string prefixed to ops created by this class. Default value: "AdditiveStateSpaceModel".
The additive model concatenates the latent states of its component models. The generative process runs each component’s dynamics in its own subspace of latent space, and then observes the sum of the observation models from the components.

Formally, the transition model is linear Gaussian:

\[ p(z[t+1] \mid z[t]) \sim \text{Normal}(\text{loc} = \text{transition_matrix}.matmul(z[t]), \text{cov} = \text{transition_cov}) \]

where each \( z[t] \) is a latent state vector concatenating the component state vectors, \( z[t] = [z1[t], z2[t], ..., zN[t]] \), so it has size \( \text{latent_size} = \text{sum}([c.latent_size \text{ for } c \text{ in components}]) \).

The transition matrix is the block-diagonal composition of transition matrices from the component processes:

\[
\begin{bmatrix}
    c0.transition_matrix & 0. & \cdots & 0. \\
    0. & c1.transition_matrix & \cdots & 0. \\
    \cdots & \cdots & \cdots \\
    0. & 0. & \cdots & cN.transition_matrix
\end{bmatrix}
\]

and the noise covariance is similarly the block-diagonal composition of component noise covariances:

\[
\begin{bmatrix}
    c0.transition_cov & 0. & \cdots & 0. \\
    0. & c1.transition_cov & \cdots & 0. \\
    \cdots & \cdots & \cdots \\
    0. & 0. & \cdots & cN.transition_cov
\end{bmatrix}
\]

The observation model is also linear Gaussian,

\[ p(y[t] \mid z[t]) \sim \text{Normal}(\text{loc} = \text{observation_matrix}.matmul(z[t]), \text{stddev} = \text{observation_noise_scale}) \]

This implementation assumes scalar observations, so \( \text{observation_matrix} \) has shape \([1, \text{latent_size}]\).

The additive observation matrix simply concatenates the observation matrices from each component:

\[
\text{observation_matrix} = \text{concat}([c0.obs_matrix, c1.obs_matrix, ..., cN.obs_matrix], \text{axis}=-1)
\]

The effect is that each component observation matrix acts on the dimensions of latent state corresponding to that component, and the overall expected observation is the sum of the expected observations from each component.

If \( \text{observation_noise_scale} \) is not explicitly specified, it is also computed by summing the noise variances of the component processes:

\[ \text{observation_noise_scale} = \sqrt{\text{sum}([c.\text{observation_noise_scale}**2 \text{ for } c \text{ in components}])} \]

**Value**

an instance of \text{LinearGaussianStateSpaceModel}. 
sts_autoregressive

Formal representation of an autoregressive model.

Description

An autoregressive (AR) model posits a latent level whose value at each step is a noisy linear combination of previous steps:

\[
\text{level}[t+1] = (\text{sum}(\text{coefficients} \times \text{levels}[t:t-\text{order}:1]) + \text{Normal}(0., \text{level_scale}))
\]

Usage

```r
sts_autoregressive(
  observed_time_series = NULL,
  order,
  coefficients_prior = NULL,
  level_scale_prior = NULL,
  initial_state_prior = NULL,
  coefficient_constraining_bijector = NULL,
  name = NULL
)
```

Arguments

- `observed_time_series`
  optional float tensor of shape `batch_shape + [T, 1]` (omitting the trailing unit dimension is also supported when `T > 1`), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: `NULL`.

- `order`
  scalar positive integer specifying the number of past timesteps to regress on.

- `coefficients_prior`
  optional Distribution instance specifying a prior on the coefficients parameter. If `NULL`, a default standard normal (`tfd_multivariate_normal_diag(scale_diag = tf$ones(list(order)))`) prior is used. Default value: `NULL`.

See Also

Other sts: `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_local_linear_trend_state_space_model()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`
level_scale_prior
optional Distribution instance specifying a prior on the level_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_state_prior
optional Distribution instance specifying a prior on the initial state, corresponding to the values of the process at a set of size order of imagined timesteps before the initial step. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

coefficient_constraining_bijector
optional Bijector instance representing a constraining mapping for the autoregressive coefficients. For example, tfb_tanh() constrains the coefficients to lie in (-1, 1), while tfb_softplus() constrains them to be positive, and tfb_identity() implies no constraint. If NULL, the default behavior constrains the coefficients to lie in (-1, 1) using a tanh bijector. Default value: NULL.

name
the name of this model component. Default value: 'Autoregressive'.

Details
The latent state is levels[t:t-order:-1]. We observe a noisy realization of the current level: \( f[t] = \text{level}[t] + \text{Normal}(0., \text{observation_noise_scale}) \) at each timestep.

If coefficients=[1.], the AR process is a simple random walk, equivalent to a LocalLevel model. However, a random walk’s variance increases with time, while many AR processes (in particular, any first-order process with abs(coefficient) < 1) are stationary, i.e., they maintain a constant variance over time. This makes AR processes useful models of uncertainty.

Value
an instance of StructuralTimeSeries.

See Also
For usage examples see sts_fit_with_hmc(), sts_forecast(), sts_decompose_by_component().

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(), sts_dynamic_linear_regression(), sts_linear_regression(), sts_local_level_state_space_model(), sts_local_level(), sts_local_linear_trend_state_space_model(), sts_local_linear_trend(), sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend_state_space_model(), sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal(), sts_sparse_linear_regression(), sts_sum()
sts_autoregressive_state_space_model

State space model for an autoregressive process.

Description
A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfd_linear_gaussian_state_space_model} for details.

Usage

```r
sts_autoregressive_state_space_model(
  num_timesteps,
  coefficients,
  level_scale,
  initial_state_prior,
  observation_noise_scale = 0,
  initial_step = 0,
  validate_args = FALSE,
  name = NULL
)
```

Arguments
- `num_timesteps`: Scalar integer tensor number of timesteps to model with this distribution.
- `coefficients`: float tensor of shape `tf$concat(batch_shape, list(order))` defining the autoregressive coefficients. The coefficients are defined backwards in time: 
  \[
  \text{coefficients}[0] \times \text{level}[t] + \text{coefficients}[1] \times \text{level}[t-1] + \ldots + \text{coefficients}[\text{order}-1] \times \text{level}[t-\text{order}+1].
  \]
- `level_scale`: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the transition noise at each step.
- `initial_state_prior`: instance of \texttt{tfd$multivariate$normal} representing the prior distribution on latent states. Must have event shape `list(order)`.
- `observation_noise_scale`: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise. Default value: 0.
- `initial_step`: Optional scalar int tensor specifying the starting timestep. Default value: 0.
- `validate_args`: logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
name

name prefixed to ops created by this class. Default value: "AutoregressiveStateSpaceModel".

Details

In an autoregressive process, the expected level at each timestep is a linear function of previous levels, with added Gaussian noise:

\[
\text{level}[t+1] = (\text{sum}(\text{coefficients} \times \text{levels}[t:t-\text{order}:-1]) + \text{Normal}(\mathbf{0}, \text{level_scale}))
\]

The process is characterized by a vector \( \text{coefficients} \) whose size determines the order of the process (how many previous values it looks at), and by \( \text{level_scale} \), the standard deviation of the noise added at each step. This is formulated as a state space model by letting the latent state encode the most recent values; see 'Mathematical Details' below.

The parameters \( \text{level_scale} \) and \( \text{observation_noise_scale} \) are each (a batch of) scalars, and \( \text{coefficients} \) is a (batch) vector of size \( \text{list(order)} \). The batch shape of this Distribution is the broadcast batch shape of these parameters and of the \( \text{initial_state_prior} \).

Mathematical Details

The autoregressive model implements a \texttt{tfd_linear_gaussian_state_space_model} with \( \text{latent_size} = \text{order} \) and \( \text{observation_size} = 1 \). The latent state vector encodes the recent history of the process, with the current value in the topmost dimension. At each timestep, the transition sums the previous values to produce the new expected value, shifts all other values down by a dimension, and adds noise to the current value. This is formally encoded by the transition model:

\[
\text{transition_matrix} = [ \text{coefs}[0], \text{coefs}[1], ..., \text{coefs}[\text{order}] \\
1., 0., ..., 0. \\
0., 1., ..., 0. \\
... \\
0., 0., ..., 1., 0. ]
\]

\[
\text{transition_noise} \sim \text{N}([\text{loc}=\mathbf{0}, \text{scale} = \text{diag}([\text{level_scale}, 0., 0., ..., 0.])])
\]

The observation model simply extracts the current (topmost) value, and optionally adds independent noise at each step:

\[
\text{observation_matrix} = [[1., 0., ..., 0.]]
\]

\[
\text{observation_noise} \sim \text{N}(\text{loc}=\mathbf{0}, \text{scale}=\text{observation_noise_scale})
\]

Models with \( \text{observation_noise_scale} = 0 \) are AR processes in the formal sense. Setting \( \text{observation_noise_scale} \) to a nonzero value corresponds to a latent AR process observed under an iid noise model.

Value

an instance of \texttt{LinearGaussianStateSpaceModel}.
See Also

Other sts:  
sts_additive_state_space_model(), sts_autoregressive(), sts_constrained_seasonal_state_space_model(),  
sts_dynamic_linear_regression_state_space_model(), sts_dynamic_linear_regression(),  
sts_linear_regression(), sts_local_level_state_space_model(), sts_local_level(), sts_local_linear_trend(),  
sts_local_trend(), sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend(),  
sts_semi_local_trend(), sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal(),  
sts_sparse_linear_regression(), sts_sum()

---

sts_build_factored_surrogate_posterior

*Build a variational posterior that factors over model parameters.*

---

Description

The surrogate posterior consists of independent Normal distributions for each parameter with trainable `loc` and `scale`, transformed using the parameter’s bijector to the appropriate support space for that parameter.

Usage

```r
sts_build_factored_surrogate_posterior(
  model,
  batch_shape = list(),
  seed = NULL,
  name = NULL
)
```

Arguments

- **model**: An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN].` *
- **batch_shape**: Batch shape (list, or integer) of initial states to optimize in parallel. Default value: `list()`. (i.e., just run a single optimization).
- **seed**: integer to seed the random number generator.
- **name**: string prefixed to ops created by this function. Default value: NULL (i.e., `build_factored_surrogate_posterior`).

Value

- **variational_posterior**: `tfd_joint_distribution_named` defining a trainable surrogate posterior over model parameters. Samples from this distribution are named lists with character parameter names as keys.
See Also

Other sts-functions: `sts_build_factored_variational_loss()`, `sts_decompose_by_component()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`, `sts_sample_uniform_initial_state()`

---

**sts_build_factored_variational_loss**

*Build a loss function for variational inference in STS models.*

**Description**

Variational inference searches for the distribution within some family of approximate posteriors that minimizes a divergence between the approximate posterior $q(z)$ and true posterior $p(z|\text{observed\_time\_series})$. By converting inference to optimization, it’s generally much faster than sampling-based inference algorithms such as HMC. The tradeoff is that the approximating family rarely contains the true posterior, so it may miss important aspects of posterior structure (in particular, dependence between variables) and should not be blindly trusted. Results may vary; it’s generally wise to compare to HMC to evaluate whether inference quality is sufficient for your task at hand.

**Usage**

```r
sts_build_factored_variational_loss(
  observed_time_series,
  model,
  init_batch_shape = list(),
  seed = NULL,
  name = NULL
)
```

**Arguments**

- **observed_time_series**
  
  Float tensor of shape `concat([sample_shape, model.batch_shape, [num_timesteps, 1]])` where `sample_shape` corresponds to i.i.d. observations, and the trailing `[1]` dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- **model**
  
  An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN]`.

- **init_batch_shape**
  
  Batch shape (list) of initial states to optimize in parallel. Default value: `list()` (i.e., just run a single optimization).

- **seed**
  
  Integer to seed the random number generator.

- **name**
  
  Name prefixed to ops created by this function. Default value: `NULL` (i.e., ‘build_factored_variational_loss’).
Details

This method constructs a loss function for variational inference using the Kullback-Liebler divergence $KL[q(z) \mid \mid p(z|\text{observed_time_series})]$, with an approximating family given by independent Normal distributions transformed to the appropriate parameter space for each parameter. Minimizing this loss (the negative ELBO) maximizes a lower bound on the log model evidence $-\text{log } p(\text{observed_time_series})$. This is equivalent to the 'mean-field' method implemented in Kucukelbir et al. (2017) and is a standard approach. The resulting posterior approximations are unimodal; they will tend to underestimate posterior uncertainty when the true posterior contains multiple modes (the $KL[q||p]$ divergence encourages choosing a single mode) or dependence between variables.

Value

- variational_loss: float Tensor of shape $\text{tf}\$concat([\text{init\_batch\_shape}, \text{model}\$\text{batch\_shape}])$, encoding a stochastic estimate of an upper bound on the negative model evidence $-\text{log } p(y)$. Minimizing this loss performs variational inference; the gap between the variational bound and the true (generally unknown) model evidence corresponds to the divergence $KL[q||p]$ between the approximate and true posterior.

- variational_distributions: a named list giving the approximate posterior for each model parameter. The keys are character parameter names in order, corresponding to $\text{param\_name}$ for param in model.parameters. The values are $\text{tfd}\$Distribution instances with batch shape $\text{tf}\$concat([\text{init\_batch\_shape}, \text{model}\$\text{batch\_shape}])$; these will typically be of the form $\text{tfd}\$TransformedDistribution($\text{tfd\_Normal(...)}, \text{bijector=param\_bijector}$).

References


See Also

Other sts-functions: `sts_build_factored_surrogate_posterior()`, `sts_decompose_by_component()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`, `sts_sample_uniform_initial_state()`
Usage

```
sts_constrained_seasonal_state_space_model(
    num_timesteps,
    num_seasons,
    drift_scale,
    initial_state_prior,
    observation_noise_scale = 1e-04,
    num_steps_per_season = 1,
    initial_step = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
)
```

Arguments

- **num_timesteps**: Scalar integer tensor number of timesteps to model with this distribution.
- **num_seasons**: Scalar integer number of seasons.
- **drift_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the change in effect between consecutive occurrences of a given season. This is assumed to be the same for all seasons.
- **initial_state_prior**: instance of `tfd_multivariate_normal` representing the prior distribution on latent states; must have event shape `[num_seasons]`.
- **observation_noise_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
- **num_steps_per_season**: integer number of steps in each season. This may be either a scalar (shape `[]`), in which case all seasons have the same length, or an array of shape `[num_seasons]`, in which seasons have different length, but remain constant around different cycles, or an array of shape `[num_cycles, num_seasons]`, in which `num_steps_per_season` for each season also varies in different cycle (e.g., a 4 years cycle with leap day). Default value: 1.
- **initial_step**: Optional scalar integer tensor specifying the starting timestep. Default value: 0.
- **validate_args**: logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
- **allow_nan_stats**: logical. If `FALSE`, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If `TRUE`, batch members with valid parameters leading to undefined statistics will return `NaN` for this statistic. Default value: `TRUE`.
- **name**: string prefixed to ops created by this class. Default value: "SeasonalStateSpaceModel".
Value

an instance of LinearGaussianStateSpaceModel.

See Also

sts_seasonal_state_space_model().

Mathematical details

The constrained model implements a reparameterization of the naive SeasonalStateSpaceModel. Instead of directly representing the seasonal effects in the latent space, the latent space of the constrained model represents the difference between each effect and the mean effect. The following discussion assumes familiarity with the mathematical details of SeasonalStateSpaceModel.

Reparameterization and constraints: let the seasonal effects at a given timestep be \( E = [e_1, ..., e_N] \). The difference between each effect \( e_i \) and the mean effect is \( z_i = e_i - \frac{\sum_i(e_i)}{N} \). By itself, this transformation is not invertible because recovering the absolute effects requires that we know the mean as well. To fix this, we'll define \( z_N = \frac{\sum_i(e_i)}{N} \) as the mean effect. It's easy to see that this is invertible: given the mean effect and the differences of the first \( N-1 \) effects from the mean, it's easy to solve for all \( N \) effects. Formally, we've defined the invertible linear reparameterization \( Z = R E \), where

\[
R = \begin{bmatrix}
1 - 1/N, & -1/N, & \dots, & -1/N \\
-1/N, & 1 - 1/N, & \dots, & -1/N \\
\vdots & \vdots & \ddots & \vdots \\
1/N, & 1/N, & \dots, & 1/N
\end{bmatrix}
\]

represents the change of basis from 'effect coordinates' \( E \) to 'residual coordinates' \( Z \). The \( Z \)s form the latent space of the ConstrainedSeasonalStateSpaceModel. To constrain the mean effect \( z_N \) to zero, we fix the prior to zero, \( p(z_N) \sim N(0, 0) \), and after the transition at each timestep we project \( z_N \) back to zero. Note that this projection is linear: to set the \( N \)th dimension to zero, we simply multiply by the identity matrix with a missing element in the bottom right, i.e., \( Z_{\text{constrained}} = P Z \), where \( P = \text{eye}(N) - \text{scatter}((N-1, N-1), 1) \).

Model: concretely, suppose a naive seasonal effect model has initial state prior \( N(m, S) \), transition matrix \( F \) and noise covariance \( Q \), and observation matrix \( H \). Then the corresponding constrained seasonal effect model has initial state prior \( N(P R m, P R S R' P') \), transition matrix \( P R F R^{-1} \) and noise covariance \( F R Q R' F' \), and observation matrix \( H R^{-1} \), where the change-of-basis matrix \( R \) and constraint projection matrix \( P \) are as defined above. This follows directly from applying the reparameterization \( Z = R E \), and then enforcing the zero-sum constraint on the prior and transition noise covariances. In practice, because the sum of effects \( z_N \) is constrained to be zero, it will never contribute a term to any linear operation on the latent space, so we can drop that dimension from the model entirely. ConstrainedSeasonalStateSpaceModel does this, so that it implements the \( N-1 \) dimension latent space \( z_1, ..., z_{[N-1]} \). Note that since we constrained the mean effect to be zero, the latent \( z_i \)s now recover their interpretation as the actual effects, \( z_i = e_i \) for \( i = 1, ..., N - 1 \), even though they were originally defined as residuals. The \( N \)th effect is represented only implicitly, as the nonzero mean of the first \( N - 1 \) effects. Although the computational representation is not symmetric across all \( N \) effects, we derived theConstrainedSeasonalStateSpaceModelby starting with a symmetric representation and imposing only a symmetric constraint (the zero-sum constraint), so the probability model remains symmetric over all \( N' \) seasonal effects.
Decompose an observed time series into contributions from each component.

### Description
This method decomposes a time series according to the posterior representation of a structural time series model. In particular, it:

- Computes the posterior marginal mean and covariances over the additive model's latent space.
- Decomposes the latent posterior into the marginal blocks for each model component.
- Maps the per-component latent posteriors back through each component's observation model, to generate the time series modeled by that component.

### Usage

```python
sts_decompose_by_component(observed_time_series, model, parameter_samples)
```

### Arguments
- **observed_time_series**: float tensor of shape `concat([sample_shape, model.batch_shape, [num_timesteps, 1]])` where `sample_shape` corresponds to i.i.d. observations, and the trailing `[1]` dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.
- **model**: An instance of `sts_sum` representing a structural time series model.
- **parameter_samples**: list of tensors representing posterior samples of model parameters, with shapes `list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) )` for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

### Value
- **component_dists**: A named list mapping component `StructuralTimeSeries` instances (elements of `model$components`) to `Distribution` instances representing the posterior marginal distributions on the process modeled by each component. Each distribution has batch shape matching that of `posterior_means/posterior_covs`, and event shape of `list(num_timesteps)`. 

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`
See Also

Other sts-functions: `sts_build_factored_surrogate_posterior()`, `sts_build_factored_variational_loss()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_one_step_predictive()`, `sts_sample_uniform_initial_state()`

Examples

```r
observed_time_series <- array(rnorm(2 * 1 * 12), dim = c(2, 1, 12))
day_of_week <- observed_time_series %>% sts_seasonal(num_seasons = 7, name = "seasonal")
local_linear_trend <- observed_time_series %>% sts_local_linear_trend(name = "local_linear")
model <- observed_time_series %>%
  sts_sum(components = list(day_of_week, local_linear_trend))
states_and_results <- observed_time_series %>%
  sts_fit_with_hmc(
    model,
    num_results = 10,
    num_warmup_steps = 5,
    num_variational_steps = 15
  )
samples <- states_and_results[[1]]

component_dists <- observed_time_series %>%
  sts_decompose_by_component(model = model, parameter_samples = samples)
```

```
sts_decompose_forecast_by_component
Decompose a forecast distribution into contributions from each component.

Description

Decompose a forecast distribution into contributions from each component.

Usage

`sts_decompose_forecast_by_component(model, forecast_dist, parameter_samples)`

Arguments

- **model**: An instance of `sts_sum` representing a structural time series model.
- **forecast_dist**: A `Distribution` instance returned by `sts_forecast()`. (Specifically, must be a `tfd.MixtureSameFamily` over a `tfd_linear_gaussian_state_space_model` parameterized by posterior samples).
parameter_samples

  list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

Value

  component_dists A named list mapping component StructuralTimeSeries instances (elements of model$components) to Distribution instances representing the marginal forecast for each component. Each distribution has batch shape matching forecast_dist (specifically, the event shape is [num_steps_forecast]).

See Also

  Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_fit_with_hmc(), sts_forecast(), sts_one_step_predictive(), sts_sample_uniform_initial_state()

sts_dynamic_linear_regression

  Formal representation of a dynamic linear regression model.

Description

  The dynamic linear regression model is a special case of a linear Gaussian SSM and a generalization of typical (static) linear regression. The model represents regression weights with a latent state which evolves via a Gaussian random walk:

Usage

  sts_dynamic_linear_regression(
    observed_time_series = NULL,
    design_matrix,
    drift_scale_prior = NULL,
    initial_weights_prior = NULL,
    name = NULL
  )

Arguments

  observed_time_series

    optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.
design_matrix: float tensor of shape \( \text{tf} \text{concat}(\text{list}(\text{batch_shape}, \text{list}(\text{num}\_\text{timesteps}, \text{num}\_\text{features}))) \). This may also optionally be an instance of \( \text{tf} \text{linalg} \text{LinearOperator} \).

drift_scale_prior: instance of \text{Distribution} specifying a prior on the drift_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed\_time\_series. Default value: NULL.

initial_weights_prior: instance of \text{tfd\_multivariate\_normal} representing the prior distribution on the latent states (the regression weights). Must have event shape \text{list}(\text{num}\_\text{features}). If NULL, a weakly-informative Normal(0, 10) prior is used. Default value: NULL.

name: the name of this component. Default value: ‘DynamicLinearRegression’.

Details

weights[t] \sim \text{Normal}(weights[t-1], \text{drift\_scale})

The latent state has dimension \text{num}\_\text{features}, while the parameters drift_scale and \text{observation\_noise\_scale} are each (a batch of) scalars. The batch shape of this distribution is the broadcast batch shape of these parameters, the initial_state_prior, and the design_matrix. \text{num}\_\text{features} is determined from the last dimension of design_matrix (equivalent to the number of columns in the design matrix in linear regression).

Value

an instance of StructuralTimeSeries.

See Also

For usage examples see \text{sts\_fit\_with\_hmc()}, \text{sts\_forecast()}, \text{sts\_decompose\_by\_component()}.

Other \text{sts}: \text{sts\_additive\_state\_space\_model()}, \text{sts\_autoregressive\_state\_space\_model()}, \text{sts\_autoregressive()}, \text{sts\_constrained\_seasonal\_state\_space\_model()}, \text{sts\_dynamic\_linear\_regression\_state\_space\_model()}, \text{sts\_linear\_regression()}, \text{sts\_local\_level\_state\_space\_model()}, \text{sts\_local\_level()}, \text{sts\_local\_linear\_trend()}, \text{sts\_local\_linear\_trend\_state\_space\_model()}, \text{sts\_seasonal\_state\_space\_model()}, \text{sts\_seasonal()}, \text{sts\_semi\_local\_linear\_trend()}, \text{sts\_semi\_local\_linear\_trend\_state\_space\_model()}, \text{sts\_smooth\_seasonal\_state\_space\_model()}, \text{sts\_smooth\_seasonal()}, \text{sts\_sparse\_linear\_regression()}, \text{sts\_sum()}
specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see `tfd_linear_gaussian_state_space_model` for details.

Usage

```r
sts_dynamic_linear_regression_state_space_model(
  num_timesteps,  
  design_matrix,  
  drift_scale,  
  initial_state_prior,  
  observation_noise_scale = 0,  
  initial_step = 0,  
  validate_args = FALSE,  
  allow_nan_stats = TRUE,  
  name = NULL
)
```

Arguments

- `num_timesteps` Scalar integer tensor, number of timesteps to model with this distribution.
- `design_matrix` float tensor of shape `tf$concat(list(batch_shape,list(num_timesteps,num_features)))`. This may also optionally be an instance of `tf$linalg$LinearOperator`.
- `drift_scale` Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the latent state transitions.
- `initial_state_prior` instance of `tfd_multivariate_normal` representing the prior distribution on latent states. Must have event shape `list(num_features)`.
- `observation_noise_scale` Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise. Default value: 0.
- `initial_step` scalar integer tensor specifying the starting timestep. Default value: 0.
- `validate_args` logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
- `allow_nan_stats` logical. If `FALSE`, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If `TRUE`, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: `TRUE`.
- `name` name prefixed to ops created by this class. Default value: 'DynamicLinearRegressionStateSpaceModel'.

Details

The dynamic linear regression model is a special case of a linear Gaussian SSM and a generalization of typical (static) linear regression. The model represents regression weights with a latent state which evolves via a Gaussian random walk: \( \text{weights}[t] \sim \text{Normal}(\text{weights}[t-1], \text{drift_scale}) \)
The latent state (the weights) has dimension num_features, while the parameters drift_scale and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters, the initial_state_prior, and the design_matrix. num_features is determined from the last dimension of design_matrix (equivalent to the number of columns in the design matrix in linear regression).

Mathematical Details
The dynamic linear regression model implements a tfd_linear_gaussian_state_space_model with latent_size = num_features and observation_size = 1 following the transition model:

transition_matrix = eye(num_features)
transition_noise ~ Normal(0, diag([drift_scale]))

which implements the evolution of weights described above. The observation model is:

observation_matrix[t] = design_matrix[t]
observation_noise ~ Normal(0, observation_noise_scale)

Value
an instance of LinearGaussianStateSpaceModel.

See Also
Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(),
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression(),
sts_linear_regression(), sts_local_level_state_space_model(), sts_local_level(), sts_local_linear_trend(),
sts_local_linear_trend(), sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend(),
sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(), sts_smooth_seasonal(),
sts_sparse_linear_regression(), sts_sum()
Usage

```r
sts_fit_with_hmc(
  observed_time_series,
  model,
  num_results = 100,
  num_warmup_steps = 50,
  num_leapfrog_steps = 15,
  initial_state = NULL,
  initial_step_size = NULL,
  chain_batch_shape = list(),
  num_variational_steps = 150,
  variational_optimizer = NULL,
  variational_sample_size = 5,
  seed = NULL,
  name = NULL
)
```

Arguments

- **observed_time_series**
  - float tensor of shape $\text{concat}([\text{sample_shape}, \text{model.batch_shape}, [\text{num_timesteps}, 1]])$ where $\text{sample_shape}$ corresponds to i.i.d. observations, and the trailing [1] dimension may (optionally) be omitted if $\text{num_timesteps} > 1$. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- **model**
  - An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape $[b_1, \ldots, b_N]$.

- **num_results**
  - Integer number of Markov chain draws. Default value: 100.

- **num_warmup_steps**
  - Integer number of steps to take before starting to collect results. The warmup steps are also used to adapt the step size towards a target acceptance rate of 0.75. Default value: 50.

- **num_leapfrog_steps**
  - Integer number of steps to run the leapfrog integrator for. Total progress per HMC step is roughly proportional to $\text{step_size} \times \text{num_leapfrog_steps}$. Default value: 15.

- **initial_state**
  - Optional Python list of Tensors, one for each model parameter, representing the initial state(s) of the Markov chain(s). These should have shape $\text{tf$\$concat(list(chain\_batch\_shape, param\_prior\_batch\_shape, param\_prior\_event\_shape)}$. If NULL, the initial state is set automatically using a sample from a variational posterior. Default value: NULL.

- **initial_step_size**
  - list of tensors, one for each model parameter, representing the step size for the leapfrog integrator. Must broadcast with the shape of `initial_state`. Larger step sizes lead to faster progress, but too-large step sizes make rejection exponentially more likely. If NULL, the step size is set automatically using the standard deviation of a variational posterior. Default value: NULL.
sts_fit_with_hmc

chain_batch_shape

Batch shape (list or int) of chains to run in parallel. Default value: list() (i.e., a single chain).

num_variational_steps

int number of steps to run the variational optimization to determine the initial state and step sizes. Default value: 150.

variational_optimizer

Optional tf$train$Optimizer instance to use in the variational optimization. If NULL, defaults to tf$train$AdamOptimizer(0.1). Default value: NULL.

variational_sample_size

integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

seed

integer to seed the random number generator.

name

name prefixed to ops created by this function. Default value: NULL (i.e., 'fit_with_hmc').

Details

This method attempts to provide a sensible default approach for fitting StructuralTimeSeries models using HMC. It first runs variational inference as a fast posterior approximation, and initializes the HMC sampler from the variational posterior, using the posterior standard deviations to set per-variable step sizes (equivalently, a diagonal mass matrix). During the warmup phase, it adapts the step size to target an acceptance rate of 0.75, which is thought to be in the desirable range for optimal mixing (Betancourt et al., 2014).

Value

list of:

- samples: list of Tensors representing posterior samples of model parameters, with shapes [concat([[num_results], chain_batch_shape, param.prior.batch_shape, param.prior.event_shape]) for param in model.parameters].
- kernel_results: A (possibly nested) list of Tensors representing internal calculations made within the HMC sampler.

References

- M.J. Betancourt, Simon Byrne, and Mark Girolami. Optimizing The Integrator Step Size for Hamiltonian Monte Carlo.

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_forecast(), sts_one_step_predictive(), sts_sample_uniform_initial_state()
Examples

```r
observed_time_series <-
  rep(c(3.5, 4.1, 4.5, 3.9, 2.4, 2.1, 1.2), 5) +
  rep(c(1.1, 1.5, 2.4, 3.1, 4.0), each = 7) %>%
  tensorflow::tf$convert_to_tensor(dtype = tensorflow::tf$float64)

day_of_week <- observed_time_series %>% sts_seasonal(num_seasons = 7)
local_linear_trend <- observed_time_series %>% sts_local_linear_trend()
model <- observed_time_series %>%
  sts_sum(components = list(day_of_week, local_linear_trend))
states_and_results <- observed_time_series %>%
  sts_fit_with_hmc(
    model,
    num_results = 10,
    num_warmup_steps = 5,
    num_variational_steps = 15)
```

---

**sts_forecast**

**Construct predictive distribution over future observations**

**Description**

Given samples from the posterior over parameters, return the predictive distribution over future observations for `num_steps_forecast` timesteps.

**Usage**

```r
sts_forecast(
  observed_time_series, 
  model, 
  parameter_samples, 
  num_steps_forecast 
)
```

**Arguments**

- **observed_time_series**
  float tensor of shape concat([sample_shape, model.batch_shape, [num_timesteps, 1]]) where sample_shape corresponds to i.i.d. observations, and the trailing [1] dimension may (optionally) be omitted if num_timesteps > 1. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

- **model**
  An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape [b1, ..., bN].
parameter_samples

- list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

num_steps_forecast

- scalar integer tensor number of steps to forecast

Value

forecast_dist a tfd_mixture_same_family instance with event shape list(num_steps_forecast,1) and batch shape tf$concat(list(sample_shape, model$batch_shape)), with num_posterior_draws mixture components.

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_fit_with_hmc(), sts_one_step_predictive(), sts_sample_uniform_initial_state()
sts_linear_regression  Formal representation of a linear regression from provided covariates.

Description
This model defines a time series given by a linear combination of covariate time series provided in a design matrix:

\[
\text{observed_time_series} \leftarrow \text{tf\$matmul(design_matrix, weights)}
\]

Usage

\[
\text{sts_linear_regression(design_matrix, weights\_prior = NULL, name = NULL)}
\]

Arguments

design_matrix  float tensor of shape \text{tf\$concat(list(batch\_shape, list(num\_timesteps, num\_features)))}. This may also optionally be an instance of \text{tf\$linalg\$LinearOperator}.

weights\_prior  Distribution representing a prior over the regression weights. Must have event shape \text{list(num\_features)} and batch shape broadcastable to the design matrix's batch\_shape. Alternately, event\_shape may be scalar (list()), in which case the prior is internally broadcast as \text{tfd\_transformed\_distribution(weights\_prior, tfb\_identity\_event\_shape = list(num\_features), batch\_shape = design\_matrix\$batch\_shape)}). If NULL, defaults to \text{tfd\_student\_t(df = 5, loc = 0, scale = 10)}, a weakly-informative prior loosely inspired by the Stan prior choice recommendations. Default value: NULL.

name  the name of this model component. Default value: 'LinearRegression'.

Details
The design matrix has shape \text{list(num\_timesteps, num\_features)}. The weights are treated as an unknown random variable of size \text{list(num\_features)} (both components also support batch shape), and are integrated over using the same approximate inference tools as other model parameters, i.e., generally HMC or variational inference.

This component does not itself include observation noise; it defines a deterministic distribution with mass at the point \text{tf\$matmul(design\_matrix, weights)}. In practice, it should be combined with observation noise from another component such as \text{sts\_sum}, as demonstrated below.

Value
an instance of StructuralTimeSeries.
See Also

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`.

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`

---

**sts_local_level**

**Formal representation of a local level model**

---

**Description**

The local level model posits a level evolving via a Gaussian random walk:

\[ \text{level}[t] = \text{level}[t-1] + \text{Normal}(0, \text{level_scale}) \]

**Usage**

```r
sts_local_level(
  observed_time_series = NULL,
  level_scale_prior = NULL,
  initial_level_prior = NULL,
  name = NULL
)
```

**Arguments**

- `observed_time_series` (optional float tensor of shape `batch_shape + [T, 1]` (omitting the trailing unit dimension is also supported when `T > 1`), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: `NULL`.

- `level_scale_prior` (optional `tfp$distribution` instance specifying a prior on the `level_scale` parameter. If `NULL`, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: `NULL`.

- `initial_level_prior` (optional `tfp$distribution` instance specifying a prior on the initial level. If `NULL`, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: `NULL`.

- `name` (the name of this model component. Default value: 'LocalLevel').
sts_local_level_state_space_model

Details

The latent state is [level]. We observe a noisy realization of the current level:

\[ f[t] = \text{level}[t] + \text{Normal}(0., \text{observation_noise_scale}) \]

at each timestep.

Value

an instance of StructuralTimeSeries.

See Also

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}.

Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_semi_local_linear_trend()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_smooth_seasonal()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

sts_local_level_state_space_model

\textit{State space model for a local level}

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \).

The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfd_linear_gaussian_state_space_model} for details. The local level model is a special case of a linear Gaussian SSM, in which the latent state posits a level evolving via a Gaussian random walk:

\[ \text{level}[t] = \text{level}[t-1] + \text{Normal}(0., \text{level_scale}) \]

Usage

\begin{verbatim}
sts_local_level_state_space_model(
    num_timesteps,
    level_scale,
    initial_state_prior,
    observation_noise_scale = 0,
    initial_step = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
)
\end{verbatim}
**Arguments**

- **num_timesteps**: Scalar integer tensor number of timesteps to model with this distribution.
- **level_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the level transitions.
- **initial_state_prior**: instance of `tfd_multivariate_normal` representing the prior distribution on latent states. Must have event shape [1] (as `tfd_linear_gaussian_state_space_model` requires a rank-1 event shape).
- **observation_noise_scale**: Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
- **initial_step**: Optional scalar integer tensor specifying the starting timestep. Default value: 0.
- **validate_args**: logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.
- **allow_nan_stats**: logical. If `FALSE`, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If `TRUE`, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: `TRUE`.
- **name**: string name prefixed to ops created by this class. Default value: "LocalLevel-StateSpaceModel".

**Details**

The latent state is [level] and [level] is observed (with noise) at each timestep.

The parameters `level_scale` and `observation_noise_scale` are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the `initial_state_prior`.

**Mathematical Details**

The local level model implements a `tfp$LinearGaussianStateSpaceModel` with `latent_size = 1` and `observation_size = 1`, following the transition model:

- `transition_matrix = [[1]]`
- `transition_noise ~ N(loc = 0, scale = diag([level_scale]))`

which implements the evolution of `level` described above, and the observation model:

- `observation_matrix = [[1]]`
- `observation_noise ~ N(loc = 0, scale = observation_noise_scale)`

**Value**

an instance of `LinearGaussianStateSpaceModel`. 

---

*sts_local_level_state_space_model* 119
See Also

Other sts: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level()`, `sts_local_linear_trend()`, `sts_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_seasonal()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`

---

**sts_local_linear_trend**

*Formal representation of a local linear trend model*

---

**Description**

The local linear trend model posits a level and slope, each evolving via a Gaussian random walk:

\[
\text{level}[t] = \text{level}[t-1] + \text{slope}[t-1] + \text{Normal}(0., \text{level_scale})
\]

\[
\text{slope}[t] = \text{slope}[t-1] + \text{Normal}(0., \text{slope_scale})
\]

**Usage**

```r
sts_local_linear_trend(
  observed_time_series = NULL,
  level_scale_prior = NULL,
  slope_scale_prior = NULL,
  initial_level_prior = NULL,
  initial_slope_prior = NULL,
  name = NULL
)
```

**Arguments**

- `observed_time_series` optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- `level_scale_prior` optional `tf$Distribution` instance specifying a prior on the level_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

- `slope_scale_prior` optional `tfd$Distribution` instance specifying a prior on the slope_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.
The latent state is the two-dimensional tuple \([\text{level}, \text{slope}]\). At each timestep we observe a noisy realization of the current level: \(f[t] = \text{level}[t] + \text{Normal}(0., \text{observation_noise_scale})\). This model is appropriate for data where the trend direction and magnitude (latent slope) is consistent within short periods but may evolve over time.

Note that this model can produce very high uncertainty forecasts, as uncertainty over the slope compounds quickly. If you expect your data to have nonzero long-term trend, i.e., that slopes tend to revert to some mean, then the SemiLocalLinearTrend model may produce sharper forecasts.

**Value**

an instance of StructuralTimeSeries.

**See Also**

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`.

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`, `sts_sum()`
sts_local_linear_trend_state_space_model

Usage

sts_local_linear_trend_state_space_model(
    num_timesteps,
    level_scale,
    slope_scale,
    initial_state_prior,
    observation_noise_scale = 0,
    initial_step = 0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
)

Arguments

num_timesteps  Scalar integer tensor number of timesteps to model with this distribution.
level_scale    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the level transitions.
slope_scale    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the slope transitions.
initial_state_prior
    instance of tfd_multivariate_normal representing the prior distribution on latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model requires a rank-1 event shape).
observation_noise_scale
    Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
initial_step    Optional scalar integer tensor specifying the starting timestep. Default value: 0.
validate_args  logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.
allow_nan_stats
    logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.
name            string prefixed to ops created by this class. Default value: "LocalLinearTrendStateSpaceModel".

Details

The local linear trend model is a special case of a linear Gaussian SSM, in which the latent state posits a level and slope, each evolving via a Gaussian random walk:

level[t] = level[t-1] + slope[t-1] + Normal(0., level_scale)
slope[t] = slope[t-1] + Normal(0., slope_scale)
The latent state is the two-dimensional tuple \([\text{level}, \text{slope}]\). The \text{level} is observed at each timestep. The parameters \text{level_scale}, \text{slope_scale}, and \text{observation_noise_scale} are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial \text{state_prior}.

Mathematical Details

The linear trend model implements a \text{tfd\_linear\_gaussian\_state\_space\_model} with \text{latent\_size} = 2 and \text{observation\_size} = 1, following the transition model:

\[
\text{transition\_matrix} = \begin{bmatrix} 1., 1. \\ 0., 1. \end{bmatrix}
\]

\[
\text{transition\_noise} \sim \text{N}(\text{loc} = 0, \text{scale} = \text{diag}([\text{level\_scale}, \text{slope\_scale}])))
\]

which implements the evolution of \([\text{level}, \text{slope}]\) described above, and the observation model:

\[
\text{observation\_matrix} = \begin{bmatrix} 1., 0. \end{bmatrix}
\]

\[
\text{observation\_noise} \sim \text{N}(\text{loc} = 0, \text{scale} = \text{observation\_noise\_scale})
\]

which picks out the first latent component, i.e., the \text{level}, as the observation at each timestep.

Value

an instance of \text{LinearGaussianStateSpaceModel}.

See Also

Other \text{sts}: \text{sts\_additive\_state\_space\_model()}, \text{sts\_autoregressive\_state\_space\_model}().
\text{sts\_autoregressive()}, \text{sts\_constrained\_seasonal\_state\_space\_model}(). \text{sts\_dynamic\_linear\_regression\_state\_space\_model}().
\text{sts\_dynamic\_linear\_regression}(), \text{sts\_linear\_regression}(), \text{sts\_local\_level\_state\_space\_model}().
\text{sts\_local\_level}(), \text{sts\_local\_linear\_trend}(), \text{sts\_seasonal\_state\_space\_model}(). \text{sts\_seasonal}().
\text{sts\_semi\_local\_linear\_trend\_state\_space\_model}(). \text{sts\_semi\_local\_linear\_trend}(). \text{sts\_smooth\_seasonal\_state\_space\_model}().
\text{sts\_smooth\_seasonal}(). \text{sts\_sparse\_linear\_regression}(), \text{sts\_sum}()

---

\text{sts\_one\_step\_predictive}

*Compute one-step-ahead predictive distributions for all timesteps*

---

\text{Description}

Given samples from the posterior over parameters, return the predictive distribution over observations at each time \(T\), given observations up through time \(T\)-1.

\text{Usage}

\text{sts\_one\_step\_predictive(\text{observed\_time\_series}, \text{model}, \text{parameter\_samples})}
## Arguments

**observed_time_series**
float tensor of shape `concat([sample_shape, model.batch_shape, [num_timesteps, 1]])`
where `sample_shape` corresponds to i.i.d. observations, and the trailing [1] dimension may (optionally) be omitted if `num_timesteps > 1`. May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations.

**model**
An instance of `StructuralTimeSeries` representing a time-series model. This represents a joint distribution over time-series and their parameters with batch shape `[b1, ..., bN]`.

**parameter_samples**
list of tensors representing posterior samples of model parameters, with shapes list(tf$concat(list(list(num_posterior_draws), param<1>$prior$batch_shape, param<1>$prior$event_shape), list(list(num_posterior_draws), param<2>$prior$batch_shape, param<2>$prior$event_shape), ... ) ) for all model parameters. This may optionally also be a named list mapping parameter names to tensor values.

## Value

**forecast_dist** a `tfd_mixture_same_family` instance with event shape `list(num_timesteps)` and batch shape `tf$concat(list(sample_shape, model$batch_shape))`, with `num_posterior_draws` mixture components. The `t`th step represents the forecast distribution $p(\text{observed_time_series}[t] | \text{observed_time_series}[0:t-1], \text{parameter_samples})$.

## See Also

Other `sts`-functions: `sts_build_factored_surrogate_posterior()`, `sts_build_factored_variational_loss()`, `sts_decompose_by_component()`, `sts_decompose_forecast_by_component()`, `sts_fit_with_hmc()`, `sts_forecast()`, `sts_sample_uniform_initial_state()`

---

**sts_sample_uniform_initial_state**

Initialize from a uniform [-2, 2] distribution in unconstrained space.

### Description

Initialize from a uniform [-2, 2] distribution in unconstrained space.

### Usage

```
sts_sample_uniform_initial_state(
  parameter,  
  return_constrained = TRUE,  
  init_sample_shape = list(),  
  seed = NULL  
)
```
sts_seasonal

Arguments

- **parameter**: sts$Parameter named tuple instance.
- **return_constrained**: if TRUE, re-applies the constraining bijector to return initializations in the original domain. Otherwise, returns initializations in the unconstrained space. Default value: TRUE.
- **init_sample_shape**: sample_shape of the sampled initializations. Default value: list().
- **seed**: integer to seed the random number generator.

Value

Uniform initializer Tensor of shape concat([init_sample_shape, parameter.prior.batch_shape, transformed_event_shape]), where transformed_event_shape is parameter.prior.event_shape, if return_constrained=TRUE, and otherwise it is parameter$bijector$inverse_event_shape(parameter$prior$event_shape).

See Also

Other sts-functions: sts_build_factored_surrogate_posterior(), sts_build_factored_variational_loss(), sts_decompose_by_component(), sts_decompose_forecast_by_component(), sts_fit_with_hmc(), sts_forecast(), sts_one_step_predictive()

---

sts_seasonal  
Formal representation of a seasonal effect model.

Description

A seasonal effect model posits a fixed set of recurring, discrete 'seasons', each of which is active for a fixed number of timesteps and, while active, contributes a different effect to the time series. These are generally not meteorological seasons, but represent regular recurring patterns such as hour-of-day or day-of-week effects. Each season lasts for a fixed number of timesteps. The effect of each season drifts from one occurrence to the next following a Gaussian random walk:

Usage

```r
sts_seasonal(
  observed_time_series = NULL,
  num_seasons,
  num_steps_per_season = 1,
  drift_scale_prior = NULL,
  initial_effect_prior = NULL,
  constrain_mean_effect_to_zero = TRUE,
  name = NULL
)
```
Arguments

observed_time_series
optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

num_seasons Scalar integer number of seasons.
num_steps_per_season integer number of steps in each season. This may be either a scalar (shape []), in which case all seasons have the same length, or an array of shape [num_seasons], in which seasons have different length, but remain constant around different cycles, or an array of shape [num_cycles, num_seasons], in which num_steps_per_season for each season also varies in different cycle (e.g., a 4 years cycle with leap day). Default value: 1.

drift_scale_prior
optional tfd$Distribution instance specifying a prior on the drift_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_effect_prior
optional tfd$Distribution instance specifying a normal prior on the initial effect of each season. This may be either a scalar tfd_normal prior, in which case it applies independently to every season, or it may be multivariate normal (e.g., tfd_multivariate_normal_diag) with event shape [num_seasons], in which case it specifies a joint prior across all seasons. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

constrain_mean_effect_to_zero
if TRUE, use a model parameterization that constrains the mean effect across all seasons to be zero. This constraint is generally helpful in identifying the contributions of different model components and can lead to more interpretable posterior decompositions. It may be undesirable if you plan to directly examine the latent space of the underlying state space model. Default value: TRUE.

name the name of this model component. Default value: ’Seasonal’.

Details

effects[season, occurrence[i]] = (    
effects[season, occurrence[i-1]] + Normal(loc=0., scale=drift_scale))

The drift_scale parameter governs the standard deviation of the random walk; for example, in a day-of-week model it governs the change in effect from this Monday to next Monday.

Value

an instance of StructuralTimeSeries.
sts_seasonal_state_space_model

State space model for a seasonal effect.

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] \mid z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] \mid z[t]) \). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see tfd_linear_gaussian_state_space_model for details.

Usage

sts_seasonal_state_space_model(
  num_timesteps,
  num_seasons,
  drift_scale,
  initial_state_prior,
  observation_noise_scale = 0,
  num_steps_per_season = 1,
  initial_step = 0,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_timesteps</td>
<td>Scalar integer tensor number of timesteps to model with this distribution.</td>
</tr>
<tr>
<td>num_seasons</td>
<td>Scalar integer number of seasons.</td>
</tr>
<tr>
<td>drift_scale</td>
<td>Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the change in effect between consecutive occurrences of a given season. This is assumed to be the same for all seasons.</td>
</tr>
</tbody>
</table>
initial_state_prior  
instance of `tfd_multivariate_normal` representing the prior distribution on latent states; must have event shape `[num_seasons]`.

observation_noise_scale  
Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.

num_steps_per_season  
integer number of steps in each season. This may be either a scalar (shape `[]`), in which case all seasons have the same length, or an array of shape `[num_seasons]`, in which seasons have different length, but remain constant around different cycles, or an array of shape `[num_cycles, num_seasons]`, in which `num_steps_per_season` for each season also varies in different cycle (e.g., a 4 years cycle with leap day). Default value: 1.

initial_step  
Optional scalar integer tensor specifying the starting timestep. Default value: 0.

validate_args  
logical. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed. Default value: `FALSE`.

allow_nan_stats  
logical. If `FALSE`, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If `TRUE`, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: `TRUE`.

name  
string prefixed to ops created by this class. Default value: "SeasonalStateSpaceModel".

details  
A seasonal effect model is a special case of a linear Gaussian SSM. The latent states represent an unknown effect from each of several 'seasons'; these are generally not meteorological seasons, but represent regular recurring patterns such as hour-of-day or day-of-week effects. The effect of each season drifts from one occurrence to the next, following a Gaussian random walk:

\[ \text{effects}_{\text{season, occurrence}[i]} = (\text{effects}_{\text{season, occurrence}[i-1]} + \text{Normal}(\text{loc}=0., \text{scale}=\text{drift}_{\text{scale}})) \]

The latent state has dimension `num_seasons`, containing one effect for each seasonal component. The parameters `drift_{scale}` and `observation_noise_{scale}` are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the `initial_state_prior`. Note: there is no requirement that the effects sum to zero.

Mathematical Details  
The seasonal effect model implements a `tfd_linear_gaussian_state_space_model` with `latent_{size} = num_seasons` and `observation_{size} = 1`. The latent state is organized so that the current seasonal effect is always in the first (zeroth) dimension. The transition model rotates the latent state to shift to a new effect at the end of each season:

\[ \text{transition_matrix}_t = (\text{permutation_matrix}([1, 2, ..., \text{num}_{\text{seasons}}-1, 0])) \]

if `season_is_{changing}(t)`
transition_noise[t] ~ Normal(loc=0., scale_diag=
[drift_scale, 0, ..., 0]
if season_is_changing(t)
else [0, 0, ..., 0])

where season_is_changing(t) is True if t mod sum(num_steps_per_season) is in the set of final days for each season, given by cumsum(num_steps_per_season) - 1. The observation model always picks out the effect for the current season, i.e., the first element of the latent state:

observation_matrix = [[1., 0., ..., 0.]]

observation_noise ~ Normal(loc=0, scale=observation_noise_scale)

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model().
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model().
sts_dynamic_linear_regression(), sts_linear_regression().
sts_local_level_state_space_model().
sts_local_level(), sts_local_linear_trend_state_space_model().
sts_local_linear_trend().
sts_seasonal().
sts_semi_local_linear_trend_state_space_model().
sts_semi_local_linear_trend().
sts_smooth_seasonal_state_space_model().
sts_smooth_seasonal().
sts_sparse_linear_regression().
sts_sum()
Arguments

observed_time_series
optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of sts_masked_time_series, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

level_scale_prior
optional tfp$distribution instance specifying a prior on the level_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

slope_mean_prior
optional tfd$Distribution instance specifying a prior on the slope_mean parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

slope_scale_prior
optional tfd$Distribution instance specifying a prior on the slope_scale parameter. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

autoregressive_coef_prior
optional tfd$Distribution instance specifying a prior on the autoregressive_coef parameter. If NULL, the default prior is a standard Normal(0, 1). Note that the prior may be implicitly truncated by constrain_ar_coef_stationary and/or constrain_ar_coef_positive. Default value: NULL.

initial_level_prior
optional tfp$distribution instance specifying a prior on the initial level. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

initial_slope_prior
optional tfd$Distribution instance specifying a prior on the initial slope. If NULL, a heuristic default prior is constructed based on the provided observed_time_series. Default value: NULL.

constrain_ar_coef_stationary
if TRUE, perform inference using a parameterization that restricts autoregressive_coef to the interval (-1, 1), or (0, 1) if force_positive_ar_coef is also TRUE, corresponding to stationary processes. This will implicitly truncate the support of autoregressive_coef_prior. Default value: TRUE.

constrain_ar_coef_positive
if TRUE, perform inference using a parameterization that restricts autoregressive_coef to be positive, or in (0, 1) if constrain_ar_coef_stationary is also TRUE. This will implicitly truncate the support of autoregressive_coef_prior. Default value: FALSE.

name
the name of this model component. Default value: 'SemiLocalLinearTrend'.

Details

level[t] = level[t-1] + slope[t-1] + Normal(0., level_scale)
The slope component in a \texttt{sts_semi_local_linear_trend} model evolves according to a first-order autoregressive (AR1) process with potentially nonzero mean:

\[
slope[t] = (slope_{mean} + autoregressive\_coef \times (slope[t-1] - slope_{mean}) + \text{Normal}(0., \text{slope\_scale}))
\]

Unlike the random walk used in \texttt{LocalLinearTrend}, a stationary AR1 process (coefficient in (-1, 1)) maintains bounded variance over time, so a \texttt{SemiLocalLinearTrend} model will often produce more reasonable uncertainties when forecasting over long timescales.

\section*{Value}

an instance of \texttt{StructuralTimeSeries}.

\section*{See Also}

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}.

Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_level()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_smooth_seasonal()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

\begin{verbatim}
sts_semi_local_linear_trend_state_space_model

State space model for a semi-local linear trend.
\end{verbatim}

\section*{Description}

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \(p(z[t+1] | z[t])\). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \(p(x[t] | z[t])\). The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfd_linear_gaussian_state_space_model} for details.

\section*{Usage}

\begin{verbatim}
sts_semi_local_linear_trend_state_space_model(
    num_timesteps,
    level_scale,
    slope_mean,
    slope_scale,
    autoregressive_coef,
    initial_state_prior,
\end{verbatim}
observation_noise_scale = 0,
initial_step = 0,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = NULL
)

Arguments

num_timesteps  Scalar integer tensor number of timesteps to model with this distribution.
level_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the level transitions.
slope_mean  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the expected long-term mean of the latent slope.
slope_scale  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the slope transitions.
autoregressive_coef
  Scalar (any additional dimensions are treated as batch dimensions) float tensor defining the AR1 process on the latent slope.
initial_state_prior
  instance of tfd_multivariate_normal representing the prior distribution on latent states. Must have event shape [1] (as tfd_linear_gaussian_state_space_model requires a rank-1 event shape).
observation_noise_scale
  Scalar (any additional dimensions are treated as batch dimensions) float tensor indicating the standard deviation of the observation noise.
initial_step
  Optional scalar integer tensor specifying the starting timestep. Default value: 0.
validate_args
  logical. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed. Default value: FALSE.
allow_nan_stats
  logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.
name
  string` prefixed to ops created by this class. Default value: "SemiLocalLinearTrend-StateSpaceModel".

Details

The semi-local linear trend model is a special case of a linear Gaussian SSM, in which the latent state posits a level and slope. The level evolves via a Gaussian random walk centered at the current slope, while the slope follows a first-order autoregressive (AR1) process with mean slope_mean:
level[t] = level[t-1] + slope[t-1] + Normal(0, level_scale)
slope[t] = (slope_mean + autoregressive_coef * (slope[t-1] - slope_mean) + Normal(0., slope_scale))

The latent state is the two-dimensional tuple [level, slope]. The level is observed at each timestep. The parameters level_scale, slope_mean, slope_scale, autoregressive_coef, and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The semi-local linear trend model implements a tfp.distributions.LinearGaussianStateSpaceModel with latent_size = 2 and observation_size = 1, following the transition model:

transition_matrix = [[1., 1.]
                     [0., autoregressive_coef]]
transition_noise ~ N(loc=slope_mean - autoregressive_coef * slope_mean, scale=diag([level_scale, slope_scale]))

which implements the evolution of [level, slope] described above, and the observation model:

observation_matrix = [[1., 0.]]
observation_noise ~ N(loc=0, scale=observation_noise_scale)

which picks out the first latent component, i.e., the level, as the observation at each timestep.

Value

an instance of LinearGaussianStateSpaceModel.

See Also

Other sts: sts_additive_state_space_model(), sts_autoregressive_state_space_model(),
sts_autoregressive(), sts_constrained_seasonal_state_space_model(), sts_dynamic_linear_regression_state_space_model(),
sts_dynamic_linear_regression(), sts_linear_regression(), sts_local_level_state_space_model(),
sts_local_level(), sts_local_linear_trend_state_space_model(), sts_local_linear_trend(),
sts_seasonal_state_space_model(), sts_seasonal(), sts_semi_local_linear_trend(), sts_smooth_seasonal_state_space_model(),
sts_smooth_seasonal(), sts_sparse_linear_regression(), sts_sum()

---

sts_smooth_seasonal  
**Formal representation of a smooth seasonal effect model**

Description

The smooth seasonal model uses a set of trigonometric terms in order to capture a recurring pattern whereby adjacent (in time) effects are similar. The model uses frequencies calculated via:
Usage

```r
sts_smooth_seasonal(
  period,
  frequency_multipliers,
  allow_drift = TRUE,
  drift_scale_prior = NULL,
  initial_state_prior = NULL,
  observed_time_series = NULL,
  name = NULL
)
```

Arguments

- **period**: positive scalar float Tensor giving the number of timesteps required for the longest cyclic effect to repeat.

- **frequency_multipliers**: One-dimensional float Tensor listing the frequencies (cyclic components) included in the model, as multipliers of the base/fundamental frequency \( 2 \times \pi / \text{period} \). Each component is specified by the number of times it repeats per period, and adds two latent dimensions to the model. A smooth seasonal model that can represent any periodic function is given by frequency_multipliers = \([1,2,\ldots,\text{floor(period / 2)}]\). However, it is often desirable to enforce a smoothness assumption (and reduce the computational burden) by dropping some of the higher frequencies.

- **allow_drift**: optional logical specifying whether the seasonal effects can drift over time. Setting this to `FALSE` removes the `drift_scale` parameter from the model. This is mathematically equivalent to `drift_scale_prior = tfd.Deterministic(0.)`, but removing drift directly is preferred because it avoids the use of a degenerate prior. Default value: `TRUE`.

- **drift_scale_prior**: optional `tfd$Distribution` instance specifying a prior on the `drift_scale` parameter. If `NULL`, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: `NULL`.

- **initial_state_prior**: instance of `tfd$MultivariateNormal` representing the prior distribution on the latent states. Must have event shape \([2 \times \text{len(frequency_multipliers)}]\). If `NULL`, a heuristic default prior is constructed based on the provided `observed_time_series`.

- **observed_time_series**: optional float Tensor of shape `batch_shape + \([T, 1]\)` (omitting the trailing unit dimension is also supported when \(T > 1\)), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `tfp$sts$MaskedTimeSeries`, which includes a mask Tensor to specify timesteps with missing observations. Default value: `NULL`.

- **name**: the name of this model component. Default value: `'LocalLinearTrend'`.

Details

- \( \text{frequencies}[j] = 2 \times \pi \times \text{frequency_multipliers}[j] / \text{period} \)
and then posits two latent states for each frequency. The two latent states associated with frequency \( j \) drift over time via:

\[
\begin{align*}
effect[t] &= (effect[t-1] \times \cos(frequencies[j]) + \\
&\quad \text{auxiliary}[t-] \times \sin(frequencies[j]) + \\
&\quad \text{Normal}(0., \text{drift_scale})
\end{align*}
\]

\[
\begin{align*}
\text{auxiliary}[t] &= (-effect[t-1] \times \sin(frequencies[j]) + \\
&\quad \text{auxiliary}[t-] \times \cos(frequencies[j]) + \\
&\quad \text{Normal}(0., \text{drift_scale})
\end{align*}
\]

where \( \text{effect} \) is the smooth seasonal effect and \( \text{auxiliary} \) only appears as a matter of construction. The interpretation of \( \text{auxiliary} \) is thus not particularly important.

Value

an instance of StructuralTimeSeries.

See Also

For usage examples see \texttt{sts_fit_with_hmc()}, \texttt{sts_forecast()}, \texttt{sts_decompose_by_component()}.

Other \texttt{sts}: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_level()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_semi_local_linear_trend()}, \texttt{sts_smooth_seasonal_state_space_model()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum()}

---

\texttt{sts_smooth_seasonal_state_space_model}

\texttt{State space model for a smooth seasonal effect}

Description

A state space model (SSM) posits a set of latent (unobserved) variables that evolve over time with dynamics specified by a probabilistic transition model \( p(z[t+1] | z[t]) \). At each timestep, we observe a value sampled from an observation model conditioned on the current state, \( p(x[t] | z[t]) \).

The special case where both the transition and observation models are Gaussians with mean specified as a linear function of the inputs, is known as a linear Gaussian state space model and supports tractable exact probabilistic calculations; see \texttt{tfp$\text{distributions$LinearGaussianStateSpaceModel} for details. A smooth seasonal effect model is a special case of a linear Gaussian SSM. It is the sum of a set of "cyclic" components, with one component for each frequency:

\[
frequencies[j] = 2. \times \pi \times frequency\_multipliers[j] / \text{period}
\]

Each cyclic component contains two latent states which we denote \( \text{effect} \) and \( \text{auxiliary} \). The two latent states for component \( j \) drift over time via:
\[ \text{effect}[t] = (\text{effect}[t-1] \ast \cos(\text{frequencies}[j]) + \\
\text{auxiliary}[t-1] \ast \sin(\text{frequencies}[j]) + \\
\text{Normal}(0., \text{drift_scale}) \]

\[ \text{auxiliary}[t] = (-\text{effect}[t-1] \ast \sin(\text{frequencies}[j]) + \\
\text{auxiliary}[t-1] \ast \cos(\text{frequencies}[j]) + \\
\text{Normal}(0., \text{drift_scale}) \]

Usage

\[
\text{sts_smooth_seasonal_state_space_model}(
\text{num_timesteps},
\text{period},
\text{frequency_multipliers},
\text{drift_scale},
\text{initial_state_prior},
\text{observation_noise_scale} = 0,
\text{initial_step} = 0,
\text{validate_args} = \text{FALSE},
\text{allow_nan_stats} = \text{TRUE},
\text{name} = \text{NULL}
\)

Arguments

- **num_timesteps**: Scalar integer Tensor number of timesteps to model with this distribution.
- **period**: positive scalar float Tensor giving the number of timesteps required for the longest cyclic effect to repeat.
- **frequency_multipliers**: One-dimensional float Tensor listing the frequencies (cyclic components) included in the model, as multipliers of the base/fundamental frequency \(2 \ast \pi / \text{period}\). Each component is specified by the number of times it repeats per period, and adds two latent dimensions to the model. A smooth seasonal model that can represent any periodic function is given by \(\text{frequency_multipliers} = [1, 2, \ldots, \text{floor} (\text{period} / 2)]\). However, it is often desirable to enforce a smoothness assumption (and reduce the computational burden) by dropping some of the higher frequencies.
- **drift_scale**: Scalar (any additional dimensions are treated as batch dimensions) float Tensor indicating the standard deviation of the latent state transitions.
- **initial_state_prior**: instance of tfd\$MultivariateNormal representing the prior distribution on latent states. Must have event shape [num_features].
- **observation_noise_scale**: Scalar (any additional dimensions are treated as batch dimensions) float Tensor indicating the standard deviation of the observation noise. Default value: 0.
- **initial_step**: scalar integer Tensor specifying the starting timestep. Default value: 0.
- **validate_args**: logical. Whether to validate input with asserts. If \text{validate_args} is \text{FALSE}, and the inputs are invalid, correct behavior is not guaranteed. Default value: \text{FALSE}.
allow_nan_stats

logical. If FALSE, raise an exception if a statistic (e.g. mean/mode/etc...) is undefined for any batch member. If TRUE, batch members with valid parameters leading to undefined statistics will return NaN for this statistic. Default value: TRUE.

name

string prefixed to ops created by this class. Default value: "LocalLinearTrend-StateSpaceModel".

Details

The auxiliary latent state only appears as a matter of construction and thus its interpretation is not particularly important. The total smooth seasonal effect is the sum of the effect values from each of the cyclic components. The parameters drift_scale and observation_noise_scale are each (a batch of) scalars. The batch shape of this Distribution is the broadcast batch shape of these parameters and of the initial_state_prior.

Mathematical Details

The smooth seasonal effect model implements a \texttt{tfp$distributions$LinearGaussianStateSpaceModel} with \texttt{latent_size} = 2 * len(frequency_multipliers) and \texttt{observation_size} = 1. The latent state is the concatenation of the cyclic latent states which themselves comprise an effect and an auxiliary state. The transition matrix is a block diagonal matrix where block j is:

\[
\text{transition_matrix}[j] = \begin{bmatrix}
\cos(\text{frequencies}[j]), & \sin(\text{frequencies}[j]) \\
-sin(\text{frequencies}[j]), & \cos(\text{frequencies}[j])
\end{bmatrix}
\]

The observation model picks out the cyclic effect values from the latent state:

\[
\text{observation_matrix} = \begin{bmatrix}
1., & 0., & 1., & 0., & \ldots, & 1., & 0.
\end{bmatrix}
\]

\[
\text{observation_noise} \sim \text{Normal}(\text{loc}=0, \text{scale} = \text{observation_noise_scale})
\]

For further mathematical details please see Harvey (1990).

Value

an instance of \texttt{LinearGaussianStateSpaceModel}.

references


See Also

Other sts: \texttt{sts_additive_state_space_model()}, \texttt{sts_autoregressive_state_space_model()}, \texttt{sts_autoregressive()}, \texttt{sts_constrained_seasonal_state_space_model()}, \texttt{sts_dynamic_linear_regression_state_space_model()}, \texttt{sts_dynamic_linear_regression()}, \texttt{sts_linear_regression()}, \texttt{sts_local_level_state_space_model()}, \texttt{sts_local_level()}, \texttt{sts_local_linear_trend_state_space_model()}, \texttt{sts_local_linear_trend()}, \texttt{sts_seasonal_state_space_model()}, \texttt{sts_seasonal()}, \texttt{sts_semi_local_linear_trend_state_space_model()}, \texttt{sts_semi_local_linear_trend()}, \texttt{sts_smooth_seasonal()}, \texttt{sts_sparse_linear_regression()}, \texttt{sts_sum}()}
sts_sparse_linear_regression

Formal representation of a sparse linear regression.

Description

This model defines a time series given by a sparse linear combination of covariate time series provided in a design matrix:

Usage

```r
sts_sparse_linear_regression(
  design_matrix,
  weights_prior_scale = 0.1,
  weights_batch_shape = NULL,
  name = NULL
)
```

Arguments

- `design_matrix`: float tensor of shape `tf$concat(list(batch_shape,list(num_timesteps,num_features)))`. This may also optionally be an instance of `tf$linalg$LinearOperator`.
- `weights_prior_scale`: float Tensor defining the scale of the Horseshoe prior on regression weights. Small values encourage the weights to be sparse. The shape must broadcast with `weights_batch_shape`. Default value: 0.1.
- `weights_batch_shape`: if `NULL`, defaults to `design_matrix.batch_shape_tensor()`. Must broadcast with the batch shape of `design_matrix`. Default value: `NULL`.
- `name`: the name of this model component. Default value: 'LinearRegression'.

Details

```r
observed_time_series <- tf$matmul(design_matrix, weights)
```

This is identical to `sts_linear_regression`, except that `sts_sparse_linear_regression` uses a parameterization of a Horseshoe prior to encode the assumption that many of the weights are zero, i.e., many of the covariate time series are irrelevant. See the mathematical details section below for further discussion. The prior parameterization used by `sts_sparse_linear_regression` is more suitable for inference than that obtained by simply passing the equivalent `tfd_horseshoe` prior to `sts_linear_regression`; when sparsity is desired, `sts_sparse_linear_regression` will likely yield better results.

This component does not itself include observation noise; it defines a deterministic distribution with mass at the point `tf$matmul(design_matrix, weights)`. In practice, it should be combined with observation noise from another component such as `sts_sum`.

Mathematical Details

The basic horseshoe prior Carvalho et al. (2009) is defined as a Cauchy-normal scale mixture:
scales[i] ~ HalfCauchy(loc=0, scale=1)
weights[i] ~ Normal(loc=0., scale=scales[i] * global_scale)

The Cauchy scale parameters put substantial mass near zero, encouraging weights to be sparse, but their heavy tails allow weights far from zero to be estimated without excessive shrinkage. The horseshoe can be thought of as a continuous relaxation of a traditional 'spike-and-slab' discrete sparsity prior, in which the latent Cauchy scale mixes between 'spike' (scales[i] \approx 0) and 'slab' (scales[i] \gg 0) regimes.

Following the recommendations in Piironen et al. (2017). \texttt{SparseLinearRegression} implements a horseshoe with the following adaptations:

- The Cauchy prior on scales[i] is represented as an InverseGamma-Normal compound.
- The global_scale parameter is integrated out following a Cauchy(0., scale=weights_prior_scale) hyperprior, which is also represented as an InverseGamma-Normal compound.
- All compound distributions are implemented using a non-centered parameterization. The compound, non-centered representation defines the same marginal prior as the original horseshoe (up to integrating out the global scale), but allows samplers to mix more efficiently through the heavy tails; for variational inference, the compound representation implicitly expands the representational power of the variational model.

Note that we do not yet implement the regularized ('Finnish') horseshoe, proposed in Piironen et al. (2017) for models with weak likelihoods, because the likelihood in STS models is typically Gaussian, where it's not clear that additional regularization is appropriate. If you need this functionality, please email tfprobability@tensorflow.org.

The full prior parameterization implemented in \texttt{SparseLinearRegression} is as follows:

Sample \texttt{global_scale} from Cauchy(0, scale=weights_prior_scale).
global_scale_variance ~ InverseGamma(alpha=0.5, beta=0.5)
global_scale_noncentered ~ HalfNormal(loc=0, scale=1)
global_scale = (global_scale_noncentered * sqrt(global_scale_variance)) * weights_prior_scale

Sample \texttt{local_scales} from Cauchy(0, 1).
local_scale_variances[i] ~ InverseGamma(alpha=0.5, beta=0.5)
local_scales_noncentered[i] ~ HalfNormal(loc=0, scale=1)
local_scales[i] = local_scales_noncentered[i] * sqrt(local_scale_variances[i])
weights[i] ~ Normal(loc=0., scale=local_scales[i] * global_scale)

Value

an instance of \texttt{StructuralTimeSeries}.

References

- Carvalho, C., Polson, N. and Scott, J. Handling Sparsity via the Horseshoe. AISTATS (2009).
See Also

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`.

Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sum()`.

---

### sts_sum

**Sum of structural time series components.**

**Description**

This class enables compositional specification of a structural time series model from basic components. Given a list of component models, it represents an additive model, i.e., a model of time series that may be decomposed into a sum of terms corresponding to the component models.

**Usage**

```r
sts_sum(
  observed_time_series = NULL,
  components,
  constant_offset = NULL,
  observation_noise_scale_prior = NULL,
  name = NULL
)
```

**Arguments**

- **observed_time_series**
  - optional float tensor of shape batch_shape + [T, 1] (omitting the trailing unit dimension is also supported when T > 1), specifying an observed time series. Any priors not explicitly set will be given default values according to the scale of the observed time series (or batch of time series). May optionally be an instance of `sts_masked_time_series`, which includes a mask tensor to specify timesteps with missing observations. Default value: NULL.

- **components**
  - list of one or more `StructuralTimeSeries` instances. These must have unique names.

- **constant_offset**
  - optional scalar float tensor, or batch of scalars, specifying a constant value added to the sum of outputs from the component models. This allows the components to model the shifted series `observed_time_series - constant_offset`. If NULL, this is set to the mean of the provided `observed_time_series`. Default value: NULL.
observation_noise_scale_prior

optional `tfd$Distribution` instance specifying a prior on `observation_noise_scale`. If `NULL`, a heuristic default prior is constructed based on the provided `observed_time_series`. Default value: `NULL`.

name

string name of this model component; used as `name_scope` for ops created by this class. Default value: `'Sum'`.

Details

Formally, the additive model represents a random process \( g[t] = f_1[t] + f_2[t] + \ldots + f_N[t] + \epsilon[t] \), where the \( f \)'s are the random processes represented by the components, and \( \epsilon[t] \sim \text{Normal}(\text{loc}=0, \text{scale}=\text{observation_noise_scale}) \) is an observation noise term. See the `AdditiveStateSpaceModel` documentation for mathematical details.

This model inherits the parameters (with priors) of its components, and adds an `observation_noise_scale` parameter governing the level of noise in the observed time series.

Value

an instance of `StructuralTimeSeries`.

See Also

For usage examples see `sts_fit_with_hmc()`, `sts_forecast()`, `sts_decompose_by_component()`. Other `sts`: `sts_additive_state_space_model()`, `sts_autoregressive_state_space_model()`, `sts_autoregressive()`, `sts_constrained_seasonal_state_space_model()`, `sts_dynamic_linear_regression_state_space_model()`, `sts_dynamic_linear_regression()`, `sts_linear_regression()`, `sts_local_level_state_space_model()`, `sts_local_level()`, `sts_local_linear_trend_state_space_model()`, `sts_local_linear_trend()`, `sts_seasonal_state_space_model()`, `sts_seasonal()`, `sts_semi_local_linear_trend_state_space_model()`, `sts_semi_local_linear_trend()`, `sts_smooth_seasonal_state_space_model()`, `sts_smooth_seasonal()`, `sts_sparse_linear_regression()`

tfb_absolute_value

Computes \( Y = g(X) = \text{Abs}(X) \), element-wise

Description

This non-injective bijector allows for transformations of scalar distributions with the absolute value function, which maps \((-\infty, \infty)\) to \([0, \infty)\).

- For \( y \in (0, \infty) \), \( \text{tfb\_absolute\_value}\$inverse(y) \) returns the set inverse \( \{ x \in (-\infty, \infty) : |x| = y \} \) as a tuple, \(-y, y\). \( \text{tfb\_absolute\_value}\$inverse(0) \) returns \(0, 0\), which is not the set inverse (the set inverse is the singleton \( \{ 0 \} \)), but "works" in conjunction with `TransformedDistribution` to produce a left semi-continuous pdf. For \( y < 0 \), \( \text{tfb\_absolute\_value}\$inverse(y) \) happily returns the wrong thing, \(-y, y\). This is done for efficiency. If `validate_args == TRUE`, \( y < 0 \) will raise an exception.
Usage

tfb_absolute_value(validate_args = FALSE, name = "absolute_value")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_ascending(),
tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(),
tfb_exp(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(),
tfb_inv(), tfb_inverted_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(),
tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(),
tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(),
tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(),
tfb_rayleigh_cdf(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(),
tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(),
tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfbTranspose(), tfb_weibull_cdf(), tfb_weibull()

tfb_affine  

Affine bijector

Description

This Bijector is initialized with shift Tensor and scale arguments, giving the forward operation: \( Y = g(X) = \text{scale} \otimes X + \text{shift} \) where the scale term is logically equivalent to: \( \text{scale} = \text{scale_identity_multiplier} \ast \text{tf.diag(tf.ones(d))} \)

Usage

tfb_affine(
    shift = NULL,
    scale_identity_multiplier = NULL,
    scale_diag = NULL,
scale_tril = NULL,
scale_perturb_factor = NULL,
scale_perturb_diag = NULL,
adjoint = FALSE,
validate_args = FALSE,
name = "affine",
dtype = NULL
)

Arguments

shift Floating-point Tensor. If this is set to NULL, no shift is applied.

scale_identity_multiplier
Floating point rank 0 Tensor representing a scaling done to the identity matrix. When
scale_identity_multiplier = scale_diag = scale_tril = NULL then
scale += IdentityMatrix. Otherwise no scaled-identity-matrix is added to scale.

scale_diag
Floating-point Tensor representing the diagonal matrix. scale.diag has shape
[N1, N2, ... k], which represents a k x k diagonal matrix. When NULL no
diagonal term is added to scale.

scale_tril
Floating-point Tensor representing the lower triangular matrix. scale_tril has
shape [N1, N2, ... k, k], which represents a k x k lower triangular matrix. When
NULL no scale_tril term is added to scale. The upper triangular elements
above the diagonal are ignored.

scale_perturb_factor
Floating-point Tensor representing factor matrix with last two dimensions of
shape (k, r) When NULL, no rank-r update is added to scale.

scale_perturb_diag
Floating-point Tensor representing the diagonal matrix. scale_perturb_diag has shape
[N1, N2, ... r], which represents an r x r diagonal matrix. When NULL
low rank updates will take the form scale_perturb_factor * scale_perturb_factor.T.

adjoint Logical indicating whether to use the scale matrix as specified or its adjoint.
Default value: FALSE.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args
is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

dtype tf$DType to prefer when converting args to Tensors. Else, we fall back to a
common dtype inferred from the args, finally falling back to float32.

Details

If NULL of scale_identity_multiplier, scale_diag, or scale_tril are specified then scale += IdentityMatrix
Otherwise specifying a scale argument has the semantics of scale += Expand(arg), i.e.,
scale_diag != NULL means scale += tf$diag(scale_diag).

Value

a bijector instance.
tfb_affine_linear_operator

Computes

\[ Y = g(X; \text{shift}, \text{scale}) = \text{scale} @ X + \text{shift} \]

Description

\( \text{shift} \) is a numeric Tensor and \( \text{scale} \) is a LinearOperator. If \( X \) is a scalar then the forward transformation is: \( \text{scale} \times X + \text{shift} \) where \( \times \) denotes broadcasted elementwise product.

Usage

```python
tfb_affine_linear_operator(
    shift = NULL,
    scale = NULL,
    adjoint = FALSE,
    validate_args = FALSE,
    name = "affine_linear_operator"
)
```

Arguments

- **shift**: Floating-point Tensor.
- **scale**: Subclass of LinearOperator. Represents the (batch) positive definite matrix \( M \) in \( \mathbb{R}^{k \times k} \).
- **adjoint**: Logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.
Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_inverse()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()``, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

tfb_affine_scalar

**AffineScalar bijector**

Description

This Bijector is initialized with shift Tensor and scale arguments, giving the forward operation: \( Y = g(X) = \text{scale} \times X + \text{shift} \) If scale is not specified, then the bijector has the semantics of scale = 1.. Similarly, if shift is not specified, then the bijector has the semantics of shift = 0..

Usage

```python
tfb_affine_scalar(
    shift = NULL,
    scale = NULL,
    validate_args = FALSE,
    name = "affine_scalar"
)
```

Arguments

- **shift**: Floating-point Tensor. If this is set to NULL, no shift is applied.
- **scale**: Floating-point Tensor. If this is set to NULL, no scale is applied.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.
tfb_ascending

Maps unconstrained R^n to R^n in ascending order.

Description
Both the domain and the codomain of the mapping is [-inf, inf]^n, however, the input of the inverse mapping must be strictly increasing. On the last dimension of the tensor, the Ascending bijector performs: y = tf$\text{cumsum}([x[0], tf\exp(x[1]), tf\exp(x[2]), ..., tf\exp(x[-1])])$

Usage
tfb_ascending(validate_args = FALSE, name = "ascending")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Value
a bijector instance.
### tfb\_batch\_normalization

Computes \( Y = g(X) \) s.t. \( X = g^{-1}(Y) = (Y - \text{mean}(Y)) / \text{std}(Y) \)

### Description

Applies Batch Normalization (Ioffe and Szegedy, 2015) to samples from a data distribution. This can be used to stabilize training of normalizing flows (Papamakarios et al., 2016; Dinh et al., 2017)

### Usage

```r
tfb_batch_normalization(
  batchnorm_layer = NULL,
  training = TRUE,
  validate_args = FALSE,
  name = "batch_normalization"
)
```

### Arguments

- `batchnorm_layer` `tf$layers$BatchNormalization` layer object. If NULL, defaults to `tf$layers$BatchNormalization(gamma_constraint=tf$nn$relu(x) + 1e-6)`. This ensures positivity of the scale variable.
- `training` If TRUE, updates running-average statistics during call to inverse().
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.
Details

When training Deep Neural Networks (DNNs), it is common practice to normalize or whiten features by shifting them to have zero mean and scaling them to have unit variance.

The `inverse()` method of the BatchNormalization bijector, which is used in the log-likelihood computation of data samples, implements the normalization procedure (shift-and-scale) using the mean and standard deviation of the current minibatch.

Conversely, the `forward()` method of the bijector de-normalizes samples (e.g. \(X \times \text{std}(Y) + \text{mean}(Y)\)) with the running-average mean and standard deviation computed at training-time. De-normalization is useful for sampling.

During training time, BatchNormalization.inverse and BatchNormalization.forward are not guaranteed to be inverses of each other because `inverse(y)` uses statistics of the current minibatch, while `forward(x)` uses running-average statistics accumulated from training. In other words, `tfb_batch_normalization().inverse(tfb_batch_normalization().forward(...))` will be identical when training=FALSE but may be different when training=TRUE.

Value

a bijector instance.

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_reshape()`, `tfb_sigmoid()`, `tfb_softplus()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_sqrt()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
**tfb_blockwise**  

*Bijector which applies a list of bijectors to blocks of a Tensor*

**Description**

More specifically, given \([F_0, F_1, \ldots, F_n]\) which are scalar or vector bijectors this bijector creates a transformation which operates on the vector \([x_0, \ldots, x_n]\) with the transformation \([F_0(x_0), F_1(x_1), \ldots, F_n(x_n)]\) where \(x_0, \ldots, x_n\) are blocks (partitions) of the vector.

**Usage**

```r
tfb_blockwise(
  bijectors,
  block_sizes = NULL,
  validate_args = FALSE,
  name = NULL
)
```

**Arguments**

- **bijectors**: A non-empty list of bijectors.
- **block_sizes**: A 1-D integer Tensor with each element signifying the length of the block of the input vector to pass to the corresponding bijector. The length of block_sizes must be equal to the length of bijectors. If left as NULL, a vector of 1’s is used.
- **validate_args**: Logical indicating whether arguments should be checked for correctness.
- **name**: String, name given to ops managed by this object. Default: E.g., `tfb_blockwise(list(tfb_exp(), tfb_softplus()))$name` == 'blockwise_of_exp_and_softplus'.

**Value**

A bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`,
tfb_chain

Bijector which applies a sequence of bijectors

Description

Bijector which applies a sequence of bijectors

Usage

```
tfb_chain(
    bijectors = NULL,
    validate_args = FALSE,
    validate_event_size = TRUE,
    parameters = NULL,
    name = NULL
)
```

Arguments

- **bijectors**: list of bijector instances. An empty list makes this bijector equivalent to the Identity bijector.
- **validate_args**: Logical indicating whether arguments should be checked for correctness.
- **validate_event_size**: Checks that bijectors are not applied to inputs with incomplete support (that is, inputs where one or more elements are a deterministic transformation of the others). For example, the following LDJ would be incorrect: `tfb_chain(list(tfb_scale(), tfb_softmax_centered()))$forward_log_det_jacobian(matrix(1:2, ncol=2))`. The jacobian contribution from `tfb_scale()` applies to a 2-dimensional input, but the output from `tfb_softmax_centered()` is a 1-dimensional input embedded in a 2-dimensional space. Setting `validate_event_size=TRUE` (default) prints warnings in these cases. When `validate_args` is also `TRUE`, the warning is promoted to an exception.
- **parameters**: Locals dict captured by subclass constructor, to be used for copy/slice re-instantiation operators.
- **name**: String, name given to ops managed by this object. Default: E.g., `tfb_chain(list(tfb_exp(), tfb_softplus()))$name == "chain_of_exp_of_softplus"`.

Value

a bijector instance.
tfb_cholesky_outer_product

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_cholesky_outer_product(),
tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform,
tfb_exp(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(),
tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inplace()
tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(),
tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(),
tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(),
tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(),
tfb_rayleigh_cdf(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(),
tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(),
tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()

tfb_cholesky_outer_product

Computes \( g(X) = X @ X.T \) where \( X \) is lower-triangular, positive-diagonal matrix

Description

Note: the upper-triangular part of \( X \) is ignored (whether or not its zero).

Usage

tfb_cholesky_outer_product(
  validate_args = FALSE,
  name = "cholesky_outer_product"
)

Arguments

- validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args
  is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- name name prefixed to Ops created by this class.

Details

The surjectivity of \( g \) as a map from the set of \( n \times n \) positive-diagonal lower-triangular matrices to
the set of SPD matrices follows immediately from executing the Cholesky factorization algorithm
on an SPD matrix \( A \) to produce a positive-diagonal lower-triangular matrix \( L \) such that \( A = L @ L.T \).
To prove the injectivity of $g$, suppose that $L_1$ and $L_2$ are lower-triangular with positive diagonals and satisfy $A = L_1 \circ L_1.T = L_2 \circ L_2.T$. Then $\text{inv}(L_1) \circ A \circ \text{inv}(L_1).T = [\text{inv}(L_1) \circ L_2] \circ [\text{inv}(L_1) \circ L_2].T = I$.

Setting $L_3 := \text{inv}(L_1) \circ L_2$, that $L_3$ is a positive-diagonal lower-triangular matrix follows from $\text{inv}(L_1)$ being positive-diagonal lower-triangular (which follows from the diagonal of a triangular matrix being its spectrum), and that the product of two positive-diagonal lower-triangular matrices is another positive-diagonal lower-triangular matrix. A simple inductive argument (proceeding one column of $L_3$ at a time) shows that, if $I = L_3 \circ L_3.T$, with $L_3$ being lower-triangular with positive-diagonal, then $L_3 = I$. Thus, $L_1 = L_2$, proving injectivity of $g$.

**Value**

A bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_ascending()`, `tfb_affine()`, `tfb_affine_linear_operator()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

**tfb_cholesky_to_inv_cholesky**

Maps the Cholesky factor of $M$ to the Cholesky factor of $M^{-1}$

**Description**

The forward and inverse calculations are conceptually identical to: forward <- function(x) tf$cholesky(tf$linalg$inv(tf$matmul(x,x,adjoint_b=TRUE))) inverse = forward However, the actual calculations exploit the triangular structure of the matrices.

**Usage**

```r
tfb_cholesky_to_inv_cholesky(
  validate_args = FALSE,
  name = "cholesky_to_inv_cholesky"
)
```
tfb_correlation_cholesky

Maps unconstrained reals to Cholesky-space correlation matrices.

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inverse()`, `tfb_inverted_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`,
`tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`,
`tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`,
`tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`,
`tfb_reshape()`,
`tfb_scale_matvec_diag()`,
`tfb_scale_matvec_linear_operator()`,
`tfb_scale_matvec_lu()`,
`tfb_scale_matvec_tri_l()`,
`tfb_scale_tri_l()`,
`tfb_shifted_gompertz_cdf()`,
`tfb_shift()`,
`tfb_sigmoid()`,
`tfb_sinh_arcsinh()`,
`tfb_sinh()`,
`tfb_softmax_arcsinh()`,
`tfb_softplus()`,
`tfb_softsign()`,
`tfb_split()`,
`tfb_square()`,
`tfb_tanh()`,
`tfb_transform_diagonal()`,
`tfb_transpose()`,
`tfb_weibull_cdf()`,
`tfb_weibull()`

Description

This bijector is a mapping between R^n and the n-dimensional manifold of Cholesky-space correlation matrices embedded in R^m, where n is the (m - 1)th triangular number; i.e. n = 1 + 2 + ... + (m - 1).

Usage

tfb_correlation_cholesky(validate_args = FALSE, name = "correlation_cholesky")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.
Details

Mathematical Details

The image of unconstrained reals under the `CorrelationCholesky` bijector is the set of correlation matrices which are positive definite. A correlation matrix can be characterized as a symmetric positive semidefinite matrix with 1s on the main diagonal. However, the correlation matrix is positive definite if no component can be expressed as a linear combination of the other components. For a lower triangular matrix $L$ to be a valid Cholesky-factor of a positive definite correlation matrix, it is necessary and sufficient that each row of $L$ have unit Euclidean norm. To see this, observe that if $L_{i}$ is the $i$th row of the Cholesky factor corresponding to the correlation matrix $R$, then the $i$th diagonal entry of $R$ satisfies:

$$1 = R_{i,i} = L_{i} \cdot L_{i} = ||L_{i}||^2$$

where '$\cdot$' is the dot product of vectors and $||...||$ denotes the Euclidean norm. Furthermore, observe that $R_{i,j}$ lies in the interval $[-1, 1]$. By the Cauchy-Schwarz inequality:

$$|R_{i,j}| = |L_{i} \cdot L_{j}| \leq ||L_{i}|| ||L_{j}|| = 1$$

This is a consequence of the fact that $R$ is symmetric positive definite with 1s on the main diagonal. The LKJ distribution with `input_output_cholesky=True` generates samples from (and computes log-densities on) the set of Cholesky factors of positive definite correlation matrices. The `CorrelationCholesky` bijector provides a bijective mapping from unconstrained reals to the support of the LKJ distribution.

Value

a bijector instance.

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_squared_expm1()`, `tfb_squared_root()`, `tfb_tanh()`, `tfb_norm()`, `tfb_exp()`, `tfb_exp1()`.
tfb_cumsum

Computes the cumulative sum of a tensor along a specified axis.

**Description**

Computes the cumulative sum of a tensor along a specified axis.

**Usage**

```r
tfb_cumsum(axis = -1, validate_args = FALSE, name = "cumsum")
```

**Arguments**

- `axis` int indicating the axis along which to compute the cumulative sum. Note that positive (and zero) values are not supported.
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value(), tfb_ascending(), tfb_blockwise(), tfb_chain(), tfb_discrete_cosine_transform(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gumbel_cdf(), tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_batch_normalization(), tfb_blockwise()`.
tfb\_discrete\_cosine\_transform

Computes $Y = g(X) = DCT(X)$, where DCT type is indicated by the type arg

Description

The discrete cosine transform efficiently applies a unitary DCT operator. This can be useful for mixing and decorrelating across the innermost event dimension. The inverse $X = g^{-1}(Y) = IDCT(Y)$, where IDCT is DCT-III for type==2. This bijector can be interleaved with Affine bijectors to build a cascade of structured efficient linear layers as in Moczulski et al., 2016. Note that the operator applied is orthonormal (i.e. norm='ortho').

Usage

```r
tfb\_discrete\_cosine\_transform(
  validate\_args = FALSE,
  dct\_type = 2,
  name = "dct"
)
```

Arguments

- `validate\_args` Logical, default FALSE. Whether to validate input with asserts. If validate\_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `dct\_type` integer, the DCT type performed by the forward transformation. Currently, only 2 and 3 are supported.
- `name` name prefixed to Ops created by this class.

Value

a bijector instance.

References


See Also

For usage examples see `tfb\_forward()`, `tfb\_inverse()`, `tfb\_inverse\_log\_det\_jacobian()`. Other bijectors: `tfb\_absolute\_value()`, `tfb\_affine\_linear\_operator()`, `tfb\_affine\_scalar()`, `tfb\_affine()`, `tfb\_ascending()`, `tfb\_batch\_normalization()`, `tfb\_blockwise()`, `tfb\_chain()`, `tfb\_cholesky\_outer\_product()`, `tfb\_cholesky\_to\_inv\_cholesky()`, `tfb\_correlation\_cholesky()`, `tfb\_cumsom()`, `tfb\_expm1()`, `tfb\_exp()`, `tfb\_ffjord()`, `tfb\_fill\_scale\_tri\_l()`, `tfb\_fill\_triangular()`, `tfb\_glow()`, `tfb\_gompertz\_cdf()`, `tfb\_gumbel\_cdf()`, `tfb\_gumbel()`, `tfb\_identity()`, `tfb\_inline()`, `tfb\_inverse\_jacobian\_log\_det()`.
tfb_exp \text{ Computes } Y = g(X) = \exp(X)

Description
Computes \(Y = g(X) = \exp(X)\)

Usage
\[tfb\textunderscore exp(validate\_args = FALSE, name = "exp")\]

Arguments

\begin{itemize}
  \item \text{validate\_args} \quad \text{Logical, default FALSE. Whether to validate input with asserts. If validate\_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.}
  \item \text{name} \quad \text{name prefixed to Ops created by this class.}
\end{itemize}

Value
a bijector instance.

See Also
For usage examples see \texttt{tfb\_forward()}, \texttt{tfb\_inverse()}, \texttt{tfb\_inverse\_log\_det\_jacobian()}.

Other bijectors: \texttt{tfb\_absolute\_value()}, \texttt{tfb\_affine\_linear\_operator()}, \texttt{tfb\_affine\_scalar()}, \texttt{tfb\_affine()}, \texttt{tfb\_ascending()}, \texttt{tfb\_batch\_normalization()}, \texttt{tfb\_blockwise()}, \texttt{tfb\_chain()}, \texttt{tfb\_cholesky\_outer\_product()}, \texttt{tfb\_cholesky\_to\_inv\_cholesky()}, \texttt{tfb\_correlation\_cholesky()}, \texttt{tfb\_cumsum()}, \texttt{tfb\_discrete\_cosine\_transform()}, \texttt{tfb\_expm1()}, \texttt{tfb\_ffjord()}, \texttt{tfb\_fill\_scale\_tri\_l()}, \texttt{tfb\_fill\_triangular()}, \texttt{tfb\_glat()}, \texttt{tfb\_gompertz\_cdf()}, \texttt{tfb\_gumbel\_cdf()}, \texttt{tfb\_gumbel()}, \texttt{tfb\_identity()}, \texttt{tfb\_inverse\_log\_det\_jacobian()}, \texttt{tfb\_invert()}, \texttt{tfb\_iterated\_sigmoid\_centered()}, \texttt{tfb\_kumaraswamy\_cdf()}, \texttt{tfb\_kumaraswamy}, \texttt{tfb\_lambert\_w\_tail()}, \texttt{tfb\_masked\_autoregressive\_default\_template()}, \texttt{tfb\_masked\_autoregressive\_flow()}, \texttt{tfb\_masked\_dense()}, \texttt{tfb\_matrix\_inverse\_tri\_l()}, \texttt{tfb\_matvec\_lu()}, \texttt{tfb\_normal\_cdf()}, \texttt{tfb\_ordered()}, \texttt{tfb\_pad()}, \texttt{tfb\_permute()}, \texttt{tfb\_power\_transform()}, \texttt{tfb\_rational\_quadratic\_spline()}, \texttt{tfb\_rayleigh\_cdf()}, \texttt{tfb\_real\_nvp\_default\_template()}, \texttt{tfb\_real\_nvp()}, \texttt{tfb\_reciprocal()}, \texttt{tfb\_reshape()}, \texttt{tfb\_scale\_matvec\_diag()}, \texttt{tfb\_scale\_matvec\_linear\_operator()}, \texttt{tfb\_scale\_matvec\_lu()}, \texttt{tfb\_scale\_matvec\_tri\_l()}, \texttt{tfb\_scale\_tri\_l()}, \texttt{tfb\_scale\_l()}, \texttt{tfb\_shifted\_gompertz\_cdf()}, \texttt{tfb\_shifted\_real\_nvp()}, \texttt{tfb\_sigmoid()}, \texttt{tfb\_sinh\_arcsinh()}, \texttt{tfb\_sinh()}, \texttt{tfb\_softmax\_centered()}, \texttt{tfb\_softplus()}, \texttt{tfb\_softsign()}, \texttt{tfb\_split()}, \texttt{tfb\_square()}, \texttt{tfb\_tanh()}, \texttt{tfb\_transform\_diagonal()}, \texttt{tfb\_transpose()}, \texttt{tfb\_weibull\_cdf()}, \texttt{tfb\_weibull}.
tfb_expm1

Computes \( Y = g(X) = \exp(X) - 1 \)

**Description**

This Bijector is no different from `tfb_chain(list(tfb_affine_scalar(shift=-1), tfb_exp()))`. However, this makes use of the more numerically stable routines `tf.math.expm1` and `tf.math.log1p`.

**Usage**

```r
tfb_expm1(validate_args = FALSE, name = "expm1")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.

- `name` name prefixed to Ops created by this class.

**Details**

Note: the `expm1(.)` is applied element-wise but the Jacobian is a reduction over the event space.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_identity()`, `tfb_inverse()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_shift()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_ffjord

Implements a continuous normalizing flow X→Y defined via an ODE.

Description

This bijector implements a continuous dynamics transformation parameterized by a differential equation, where initial and terminal conditions correspond to domain (X) and image (Y) i.e.

Usage

tfb_ffjord(
    state_time_derivative_fn,
    ode_solve_fn = NULL,
    trace_augmentation_fn = tfp$bijectors$ffjord$trace_jacobian_hutchinson,
    initial_time = 0,
    final_time = 1,
    validate_args = FALSE,
    dtype = tf$float32,
    name = "ffjord"
)

Arguments

state_time_derivative_fn
    function taking arguments time (a scalar representing time) and state (a Tensor representing the state at given time) returning the time derivative of the state at given time.

ode_solve_fn
    function taking arguments ode_fn (same as state_time_derivative_fn above), initial_time (a scalar representing the initial time of integration), initial_state (a Tensor of floating dtype represents the initial state) and solution_times (1D Tensor of floating dtype representing time at which to obtain the solution) returning a Tensor of shape [time_axis, initial_state$shape]. Will take [final_time] as the solution_times argument and state_time_derivative_fn as ode_fn argument. If NULL a DormandPrince solver from tfp$math$ode is used. Default value: NULL

trace_augmentation_fn
    function taking arguments ode_fn (function same as state_time_derivative_fn above), state_shape (TensorShape of a the state), dtype (same as dtype of the state) and returning a function taking arguments time (a scalar representing the time at which the function is evaluated), state (a Tensor representing the state at given time) that computes a tuple (ode_fn(time,state), jacobian_trace_estimation). jacobian_trace_estimation should represent trace of the jacobian of ode_fn with respect to state. state_time_derivative_fn will be passed as ode_fn argument. Default value: tfp$bijectors$ffjord$trace_jacobian_hutchinson

initial_time
    Scalar float representing time to which the x value of the bijector corresponds to. Passed as initial_time to ode_solve_fn. For default solver can be float or floating scalar Tensor. Default value: 0.
final_time  Scalar float representing time to which the \( y \) value of the bijector corresponds to. Passed as `solution_times` to `ode_solve_fn`. For default solver can be float or floating scalar Tensor. Default value: 1.

validate_args  Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

dtype  `tf$DType` to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.

name  name prefixed to Ops created by this class.

Details

\[
\frac{d}{dt}[\text{state}(t)] = \text{state_time_derivative_fn}(t, \text{state}(t))
\]

\[
\text{state}(\text{initial\_time}) = X
\]

\[
\text{state}(\text{final\_time}) = Y
\]

For this transformation the value of `log\_det\_jacobian` follows another differential equation, reducing it to computation of the trace of the jacobian along the trajectory

\[
\text{state\_time\_derivative} = \text{state\_time\_derivative_fn}(t, \text{state}(t))
\]

\[
\frac{d}{dt}[\text{log\_det\_jac}(t)] = \text{Tr}(\text{jacobian}(\text{state\_time\_derivative}, \text{state}(t)))
\]

FFJORD constructor takes two functions `ode\_solve\_fn` and `trace\_augmentation\_fn` arguments that customize integration of the differential equation and trace estimation.

Differential equation integration is performed by a call to `ode\_solve\_fn`.

Custom `ode\_solve\_fn` must accept the following arguments:

- `ode\_fn(time, state)`: Differential equation to be solved.
- `initial\_time`: Scalar float or floating Tensor representing the initial time.
- `initial\_state`: Floating Tensor representing the initial state.
- `solution\_times`: 1D floating Tensor of solution times.

And return a Tensor of shape `[solution\_times$shape, initial\_state$shape]` representing state values evaluated at `solution\_times`. In addition `ode\_solve\_fn` must support nested structures. For more details see the interface of `tfp$math$ode$Solver$solve()`.

Trace estimation is computed simultaneously with `state\_time\_derivative` using `augmented\_state\_time\_derivative\_fn` that is generated by `trace\_augmentation\_fn`. `trace\_augmentation\_fn` takes `state\_time\_derivative\_fn`, `state\_shape` and `state\_dtype` arguments and returns a `augmented\_state\_time\_derivative\_fn` callable that computes both `state\_time\_derivative` and unreduced trace estimation.

Custom `ode\_solve\_fn` and `trace\_augmentation\_fn` examples:

```r
# custom_solver\_fn: 'function(f, t\_initial, t\_solutions, y\_initial, ...)' # ... : Additional arguments to pass to custom_solver\_fn.
ode\_solve\_fn <- function(ode\_fn, initial\_time, initial\_state, solution\_times) {
  custom_solver\_fn(ode\_fn, initial\_time, solution\_times, initial\_state, ...)
}
ffjord <- tfb\_ffjord(state\_time\_derivative\_fn, ode\_solve\_fn = ode\_solve\_fn)
```
For more details on FFJORD and continuous normalizing flows see Chen et al. (2018), Grathwohl et al. (2018).

Value

A bijector instance.

References

differential equations. In Advances in neural information processing systems (pp. 6571-6583)

Free-form continuous dynamics for scalable reversible generative models. arXiv preprint

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_exp(), tfb_fill_scale_tril(),
tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(),
tfb_identity(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(),
tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tril(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tril(), tfb_scale_tril(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfb_transpose(), tfb_weibull_cdf(), tfb_weibull
tfb_fill_scale_tri_l  Transforms unconstrained vectors to TriL matrices with positive diagonal

Description
This is implemented as a simple tfb_chain of tfb_fill_triangular followed by tfb_transform_diagonal, and provided mostly as a convenience. The default setup is somewhat opinionated, using a Softplus transformation followed by a small shift (1e-5) which attempts to avoid numerical issues from zeros on the diagonal.

Usage
```
tfb_fill_scale_tri_l(
    diag_bijector = NULL,
    diag_shift = 1e-05,
    validate_args = FALSE,
    name = "fill_scale_tril"
)
```

Arguments
- `diag_bijector`  Bijector instance, used to transform the output diagonal to be positive. Default value: NULL (i.e., tfb_softplus()).
- `diag_shift`  Float value broadcastable and added to all diagonal entries after applying the diag_bijector. Setting a positive value forces the output diagonal entries to be positive, but prevents inverting the transformation for matrices with diagonal entries less than this value. Default value: 1e-5.
- `validate_args`  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`  name prefixed to Ops created by this class.

Value
a bijector instance.

See Also
For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().
Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfbascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfbcumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfbffjord(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_independent(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_fill_triangular  Transforms vectors to triangular

Description

Triangular matrix elements are filled in a clockwise spiral. Given input with shape batch_shape + [d], produces output with shape batch_shape + [n, n], where $n = \frac{-1 + \sqrt{1 + 8 \times d}}{2}$. This follows by solving the quadratic equation $d = 1 + 2 + \ldots + n = n \times (n + 1)/2$.

Usage

```python
tfb_fill_triangular(
    upper = FALSE,
    validate_args = FALSE,
    name = "fill_triangular"
)
```

Arguments

- **upper**: Logical representing whether output matrix should be upper triangular (TRUE) or lower triangular (FALSE, default).
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`.
tfb_forward

Returns the forward Bijector evaluation, i.e., $X = g(Y)$.

Description

Returns the forward Bijector evaluation, i.e., $X = g(Y)$.

Usage

```r
tfb_forward(bijector, x, name = "forward")
```

Arguments

- `bijector` The bijector to apply
- `x` Tensor. The input to the "forward" evaluation.
- `name` name of the operation

Value

a tensor

See Also

Other bijector methods: `tfb_forward_log_det_jacobian()`, `tfb_inverse_log_det_jacobian()`, `tfb_inverse()`

Examples

```r
b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
b %>% tfb_forward(x)
```
tfb_forward_log_det_jacobian

Returns the result of the forward evaluation of the log determinant of the Jacobian

Description

Returns the result of the forward evaluation of the log determinant of the Jacobian

Usage

\[
\text{tfb_forward_log_det_jacobian}(\text{bijector}, \text{x}, \text{event_ndims}, \text{name} = \text{"forward_log_det_jacobian"})
\]

Arguments

- **bijector**: The bijector to apply
- **x**: Tensor. The input to the "forward" Jacobian determinant evaluation.
- **event_ndims**: Number of dimensions in the probabilistic events being transformed. Must be greater than or equal to bijector$\text{forward_min_event_ndims}$. The result is summed over the final dimensions to produce a scalar Jacobian determinant for each event, i.e. it has shape x$\text{shape}\text{ndims} - \text{event_ndims}$ dimensions.
- **name**: name of the operation

Value

a tensor

See Also

Other bijector_methods: \text{tfb_forward()}, \text{tfb_inverse_log_det_jacobian()}, \text{tfb_inverse}()

Examples

\[
b \leftarrow \text{tfb_affine_scalar}(\text{shift} = 1, \text{scale} = 2)
\]
\[
x \leftarrow 10
\]
\[
b \text{%%} \text{tfb_forward_log_det_jacobian}(x, \text{event_ndims} = 0)
\]
tfb_glow

Implements the Glow Bijector from Kingma & Dhariwal (2018).

Description

Overview: Glow is a chain of bijectors which transforms a rank-1 tensor (vector) into a rank-3 tensor (e.g. an RGB image). Glow does this by chaining together an alternating series of "Blocks," "Squeezes," and "Exits" which are each themselves special chains of other bijectors. The intended use of Glow is as part of a tfd_transformed_distribution, in which the base distribution over the vector space is used to generate samples in the image space. In the paper, an Independent Normal distribution is used as the base distribution.

Usage

```r
tfb_glow(
  output_shape = c(32, 32, 3),
  num_glow_blocks = 3,
  num_steps_per_block = 32,
  coupling_bijector_fn = NULL,
  exit_bijector_fn = NULL,
  grab_after_block = NULL,
  use_actnorm = TRUE,
  seed = NULL,
  validate_args = FALSE,
  name = "glow"
)
```

Arguments

- **output_shape**: A list of integers, specifying the event shape of the output, of the bijectors forward pass (the image). Specified as [H, W, C]. Default Value: (32, 32, 3)
- **num_glow_blocks**: An integer, specifying how many downsampling levels to include in the model. This must divide equally into both H and W, otherwise the bijector would not be invertible. Default Value: 3
- **num_steps_per_block**: An integer specifying how many Affine Coupling and 1x1 convolution layers to include at each level of the spatial hierarchy. Default Value: 32 (i.e. the value used in the original glow paper).
- **coupling_bijector_fn**: A function which takes the argument `input_shape` and returns a callable neural network (e.g. a keras_model_sequential()). The network should either return a tensor with the same event shape as `input_shape` (this will employ additive coupling), a tensor with the same height and width as `input_shape` but twice the number of channels (this will employ affine coupling), or a bijector which takes in a tensor with event shape `input_shape`, and returns a tensor with shape `input_shape`. The intended use of Glow is as part of a tfd_transformed_distribution, in which the base distribution over the vector space is used to generate samples in the image space. In the paper, an Independent Normal distribution is used as the base distribution.
**exit_bijector_fn**

Similar to coupling_bijector_fn, exit_bijector_fn is a function which takes the argument input_shape and output_chan and returns a callable neural network. The neural network it returns should take a tensor of shape input_shape as the input, and return one of three options: A tensor with output_chan channels, a tensor with 2 * output_chan channels, or a bijector. Additional details can be found in the documentation for ExitBijector.

**grab_after_block**

A tuple of floats, specifying what fraction of the remaining channels to remove following each glow block. Glow will take the integer floor of this number multiplied by the remaining number of channels. The default is half at each spatial hierarchy. Default value: None (this will take out half of the channels after each block).

**use_actnorm**

A boolean deciding whether or not to use actnorm. Data-dependent initialization is used to initialize this layer. Default value: FALSE

**seed**

A seed to control randomness in the 1x1 convolution initialization. Default value: NULL (i.e., non-reproducible sampling).

**validate_args**

Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

**name**

name prefixed to Ops created by this class.

**Details**

A "Block" (implemented as the GlowBlock Bijector) performs much of the transformations which allow glow to produce sophisticated and complex mappings between the image space and the latent space and therefore achieve rich image generation performance. A Block is composed of num_steps_per_block steps, which are each implemented as a Chain containing an ActivationNormalization (ActNorm) bijector, followed by an (invertible) OneByOneConv bijector, and finally a coupling bijector. The coupling bijector is an instance of a RealNVP bijector, and uses the coupling_bijector_fn function to instantiate the coupling bijector function which is given to the RealNVP. This function returns a bijector which defines the coupling (e.g. Shift(Scale) for affine coupling or Shift for additive coupling).

A "Squeeze" converts spatial features into channel features. It is implemented using the Expand bijector. The difference in names is due to the fact that the forward function from glow is meant to ultimately correspond to sampling from a tfp$util$TransformedDistribution object, which would use Expand (Squeeze is just Invert(Expand)). The Expand bijector takes a tensor with shape [H, W, C] and returns a tensor with shape [2H, 2W, C / 4], such that each 2x2x1 spatial tile in the output is composed from a single 1x1x4 tile in the input tensor, as depicted in the figure below.

Forward pass (Expand)

```
\__1__ \----> \ 1 \ 2 \\
\__2__ \ \ \ \ 3 \ 4 \\
\__3__ \ \ \ \ 5 \ 6 \\
\__4__ \ \ \ \ 7 \ 8 \\
```
168

tfb_glow
Inverse pass (Squeeze) This is implemented using a chain of Reshape -> Transpose -> Reshape
bijectors. Note that on an inverse pass through the bijector, each Squeeze will cause the width/height
of the image to decrease by a factor of 2. Therefore, the input image must be evenly divisible by 2
at least num_glow_blocks times, since it will pass through a Squeeze step that many times.
An "Exit" is simply a junction at which some of the tensor "exits" from the glow bijector and therefore avoids any further alteration. Each exit is implemented as a Blockwise bijector, where some
channels are given to the rest of the glow model, and the rest are given to a bypass implemented using the Identity bijector. The fraction of channels to be removed at each exit is determined by the
grab_after_block arg, indicates the fraction of remaining channels which join the identity bypass.
The fraction is converted to an integer number of channels by multiplying by the remaining number
of channels and rounding. Additionally, at each exit, glow couples the tensor exiting the highway
to the tensor continuing onward. This makes small scale features in the image dependent on larger
scale features, since the larger scale features dictate the mean and scale of the distribution over the
smaller scale features. This coupling is done similarly to the Coupling bijector in each step of the
flow (i.e. using a RealNVP bijector). However for the exit bijector, the coupling is instantiated using exit_bijector_fn rather than coupling bijector fn, allowing for different behaviors between
standard coupling and exit coupling. Also note that because the exit utilizes a coupling bijector,
there are two special cases (all channels exiting and no channels exiting). The full Glow bijector
consists of num_glow_blocks Blocks each of which contains num_steps_per_block steps. Each
step implements a coupling using bijector_coupling_fn. Between blocks, glow converts between spatial pixels and channels using the Expand Bijector, and splits channels out of the bijector
using the Exit Bijector. The channels which have exited continue onward through Identity bijectors
and those which have not exited are given to the next block. After passing through all Blocks, the
tensor is reshaped to a rank-1 tensor with the same number of elements. This is where the distribution will be defined. A schematic diagram of Glow is shown below. The forward function of the
bijector starts from the bottom and goes upward, while the inverse function starts from the top and
proceeds downward.

Value
a bijector instance.
#’ “‘
Glow Schematic Diagram Input Image ######################## shape = [H, W, C] \ /<- Expand Bijector turns spatial \ / dimensions into channels. | XXXXXXXXXXXXXXXXXXXX |
XXXXXXXXXXXXXXXXXXXX | XXXXXXXXXXXXXXXXXXXX A single step of the flow
consists Glow Block - | XXXXXXXXXXXXXXXXXXXX <- of ActNorm -> 1x1Conv -> Coupling. | XXXXXXXXXXXXXXXXXXXX there are num_steps_per_block | XXXXXXXXXXXXXXXXXXXX
steps of the flow in each block. |_ XXXXXXXXXXXXXXXXXXXX \ / <– Expand bijectors follow each glow \ / block XXXXXXXX\\\\ <– Exit Bijector removes channels _ _ from additional
alteration. | XXXXXXXX ! | ! | XXXXXXXX ! | ! | XXXXXXXX ! | ! After exiting, channels are
passed Glow Block - | XXXXXXXX ! | ! <— downward using the Blockwise and | XXXXXXXX
! | ! Identify bijectors. | XXXXXXXX ! | ! |_ XXXXXXXX ! | ! \ / <—- Expand Bijector \ / XXX\\
| ! <—- Exit Bijector _ | XXX ! | | ! | XXX ! | | ! | XXX ! | | ! low Block - | XXX ! | | ! | XXX ! | |
! | XXX ! | | ! |_ XXX ! | | ! XX\ ! | | ! <—– (Optional) Exit Bijector | | | v v v Output Distribution
########## shape = [H * W * C]
Legend


tfb_gompertz_cdf

\[ Y = g(X) = 1 - \exp(-c \cdot (\exp(rate \cdot X) - 1) \]

**Description**

This bijector maps inputs from \([-\infty, \infty]\) to \([0, \infty]\). The inverse of the bijector applied to a uniform random variable \(X \sim U(0, 1)\) gives back a random variable with the Gompertz distribution:

\[ Y \sim \text{GompertzCDF}(\text{concentration}, \text{rate}) \]

\[ \text{pdf}(y; c, r) = r \cdot c \cdot \exp(r \cdot y + c - c \cdot \exp(-c \cdot \exp(r \cdot y))) \]

Note: Because the Gompertz distribution concentrates its mass close to zero, for larger rates or larger concentrations, `bijector.forward` will quickly saturate to 1.

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_inverse()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
Usage

```plaintext
tfb_gompertz_cdf(
    concentration,
    rate,
    validate_args = FALSE,
    name = "gompertz_cdf"
)
```

Arguments

- **concentration**: Positive Float-like Tensor that is the same dtype and is broadcastable with concentration. This is $c$ in $Y = g(X) = 1 - \exp(-c \cdot (\exp(rate \cdot X) - 1)$.

- **rate**: Positive Float-like Tensor that is the same dtype and is broadcastable with concentration. This is $rate$ in $Y = g(X) = 1 - \exp(-c \cdot (\exp(rate \cdot X) - 1)$.

- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_gumbel

Computes $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$

**Description**

This bijector maps inputs from $[-\infty, \infty]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Gumbel distribution:

**Usage**

```r
tfb_gumbel(loc = 0, scale = 1, validate_args = FALSE, name = "gumbel")
```

**Arguments**

- `loc`: Float-like Tensor that is the same dtype and is broadcastable with `scale`. This is $\text{loc}$ in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- `scale`: Positive Float-like Tensor that is the same dtype and is broadcastable with `loc`. This is scale in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- `validate_args`: Logical, default `FALSE`. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

**Details**

$Y \sim \text{Gumbel}(\text{loc}, \text{scale})$

$$\text{pdf}(y; \text{loc}, \text{scale}) = \exp(-( (y - \text{loc}) / \text{scale} + \exp(- (y - \text{loc}) / \text{scale}) ) ) / \text{scale}$$

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_ascending()`, `tfb_ascending()`, `tfb_block_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`,
tfb_gumbel_cdf

Compute $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$, the Gumbel CDF.

Description

This bijector maps inputs from $[-\infty, \infty]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0,1)$ gives back a random variable with the Gumbel distribution:

Usage

tfb_gumbel_cdf(loc = 0, scale = 1, validate_args = FALSE, name = "gumbel_cdf")

Arguments

- **loc**: Float-like Tensor that is the same dtype and is broadcastable with scale. This is loc in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- **scale**: Positive Float-like Tensor that is the same dtype and is broadcastable with loc. This is scale in $Y = g(X) = \exp(-\exp(-(X - \text{loc}) / \text{scale}))$.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Details

$Y \sim \text{GumbelCDF}(\text{loc}, \text{scale})$

$$\text{pdf}(y; \text{loc}, \text{scale}) = \exp(-( (y - \text{loc}) / \text{scale} + \exp(- (y - \text{loc}) / \text{scale} ))) / \text{scale}$$

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian(). Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel(), tfb_identity(), tfb_independent(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_shifted_box_cox(), tfb_sigmoid_centered(), tfb_sigmoid(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull().
tfb_identity

Computes \( Y = g(X) = X \)

**Description**
Computes \( Y = g(X) = X \)

**Usage**
```
tfb_identity(validate_args = FALSE, name = "identity")
```

**Arguments**

- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name** name prefixed to Ops created by this class.

**Value**
a bijector instance.

**See Also**
For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expm1()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_gompef()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_instantiated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp()`.

```r

```
tfb_inline

_Bijector constructed from custom functions_

**Description**

Bijector constructed from custom functions

**Usage**

```r
tfb_inline(
  forward_fn = NULL,
  inverse_fn = NULL,
  inverse_log_det_jacobian_fn = NULL,
  forward_log_det_jacobian_fn = NULL,
  forward_event_shape_fn = NULL,
  forward_event_shape_tensor_fn = NULL,
  inverse_event_shape_fn = NULL,
  inverse_event_shape_tensor_fn = NULL,
  is_constant_jacobian = NULL,
  validate_args = FALSE,
  forward_min_event_ndims = NULL,
  inverse_min_event_ndims = NULL,
  name = "inline"
)
```

**Arguments**

- `forward_fn` Function implementing the forward transformation.
- `inverse_fn` Function implementing the inverse transformation.
- `inverse_log_det_jacobian_fn` Function implementing the log_det_jacobian of the forward transformation.
- `forward_log_det_jacobian_fn` Function implementing the log_det_jacobian of the inverse transformation.
- `forward_event_shape_fn` Function implementing non-identical static event shape changes. Default: shape is assumed unchanged.
- `forward_event_shape_tensor_fn` Function implementing non-identical event shape changes. Default: shape is assumed unchanged.
- `inverse_event_shape_fn` Function implementing non-identical static event shape changes. Default: shape is assumed unchanged.
- `inverse_event_shape_tensor_fn` Function implementing non-identical event shape changes. Default: shape is assumed unchanged.
tfb_inverse

\[ X = g^{-1}(Y) \]

**Description**

Returns the inverse Bijector evaluation, i.e., \( X = g^{-1}(Y) \).

**Usage**

```python
tfb_inverse(bijector, y, name = "inverse")
```
tfb_inverse_log_det_jacobian

Arguments

- bijector: The bijector to apply
- y: Tensor. The input to the "inverse" evaluation.
- name: name of the operation

Value

a tensor

See Also

Other bijector methods: tfb_forward_log_det_jacobian(), tfb_forward(), tfb_inverse_log_det_jacobian()

Examples

```r
b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
y <- b %>% tfb_forward(x)
b %>% tfb_inverse(y)
```

Description

Returns the result of the inverse evaluation of the log determinant of the Jacobian

Usage

```r
tfb_inverse_log_det_jacobian(
  bijector,
  y,
  event_ndims,
  name = "inverse_log_det_jacobian"
)
```

Arguments

- bijector: The bijector to apply
- y: Tensor. The input to the "inverse" Jacobian determinant evaluation.
- event_ndims: Number of dimensions in the probabilistic events being transformed. Must be greater than or equal to bijector$inverse_min_event_ndims. The result is summed over the final dimensions to produce a scalar Jacobian determinant for each event, i.e. it has shape x$shape$ndims - event_ndims dimensions.
- name: name of the operation
tfb_invert

Value

a tensor

See Also

Other bijector_methods: tfb_forward_log_det_jacobian(), tfb_forward(), tfb_inverse()

Examples

b <- tfb_affine_scalar(shift = 1, scale = 2)
x <- 10
y <- b %>% tfb_forward(x)
b %>% tfb_inverse_log_det_jacobian(y, event_ndims = 0)

---

tfb_invert

Bijector which inverts another Bijector

Description

Creates a Bijector which swaps the meaning of inverse and forward. Note: An inverted bijector’s inverse_log_det_jacobian is often more efficient if the base bijector implements _forward_log_det_jacobian. If _forward_log_det_jacobian is not implemented then the following code is used: y = b$inverse(x) -b$inverse_log_det_jacobian(y)

Usage

tfb_invert(bijector, validate_args = FALSE, name = NULL)

Arguments

bijector Bijector instance.
validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
name name prefixed to Ops created by this class.

Value

a bijector instance.
tfb_iterated_sigmoid_centered

Bijector which applies a Stick Breaking procedure.

Description

Bijector which applies a Stick Breaking procedure.

Usage

tfb_iterated_sigmoid_centered(validate_args = FALSE, name = "iterated_sigmoid")

Arguments

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_ffjord(), tfb_fill_triangular(), tfb_glow(), tfb_gumbel_cdf(), tfb_identity(), tfb_inverse_log_det_jacobian(), tfb_inverse(), tfb_lambert_w_tail(), tfb_matrix_inverse_tri_l(), tfb_permute(), tfb_power_transform(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_shifted_gompertz_cdf(), tfb_shift(), tfb_softplus(), tfb_stack(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
tfb_kumaraswamy Computes \( Y = g(X) = (1 - (1 - X)^{(1/b)})^{(1/a)} \), with \( X \) in [0, 1]

Description

This bijector maps inputs from [0, 1] to [0, 1]. The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Kumaraswamy distribution: \( Y \sim \text{Kumaraswamy}(a,b) \) 

\[
pdf(Y; a, b, 0 \leq y \leq 1) = a \cdot b \cdot y^{(a - 1)} \cdot (1 - y^a)^{(b - 1)}
\]

Usage

```python
tfb_kumaraswamy(
    concentration1 = NULL,
    concentration0 = NULL,
    validate_args = FALSE,
    name = "kumaraswamy"
)
```

Arguments

- **concentration1**: float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where \( a \) is concentration1.
- **concentration0**: float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where \( b \) is concentration0.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

- a bijector instance.
tfb_kumaraswamy_cdf

Computes \( Y = g(X) = \left( 1 - (1 - X)^{1/b} \right)^{1/a} \), with \( X \) in \([0, 1]\).

### Description

This bijector maps inputs from \([0, 1]\) to \([0, 1]\). The inverse of the bijector applied to a uniform random variable \( X \sim U(0, 1) \) gives back a random variable with the Kumaraswamy distribution: \( Y \sim \text{Kumaraswamy}(a, b) \)

\[
\text{pdf}(y; a, b, 0 \leq y \leq 1) = a \cdot b \cdot y^{(a - 1)} \cdot (1 - y^a)^{(b - 1)}
\]

### Usage

```python
tfb_kumaraswamy_cdf(
    concentration1 = 1,
    concentration0 = 1,
    validate_args = FALSE,
    name = "kumaraswamy_cdf"
)
```

### Arguments

- **concentration1**: float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where \( a \) is concentration1.
- **concentration0**: float scalar indicating the transform power, i.e., \( Y = g(X) = (1 - (1 - X)^{1/b})^{1/a} \) where \( b \) is concentration0.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

### See Also

For usage examples see `tfb_forward`, `tfb_inverse`, `tfb_inverse_log_det_jacobian`.

Other bijectors: `tfb_absolute_value`, `tfb_affine_linear_operator`, `tfb_affine_scalar`, `tfb_ascending`, `tfb_batch_normalization`, `tfb_blockwise`, `tfb_chain`, `tfb_cholesky_outer_product`, `tfb_cholesky_to_inv_cholesky`, `tfb_correlation_cholesky`, `tfb_cumsum`, `tfb_discrete_cosine_transform`, `tfb_exp`, `tfb_expm1`, `tfb_ffjord`, `tfb_fill_scale_tri_l`, `tfb_fill_triangular`, `tfb_glow`, `tfb_gompertz_cdf`, `tfb_gumbel_cdf`, `tfb_gumbel`, `tfb_identity`, `tfb_inline`, `tfb_invert`, `tfb_iterated_sigmoid_centered`, `tfb_kumaraswamy_cdf`, `tfb_lambert_w_tail`, `tfb_masked_autoregressive_default_template`, `tfb_masked_autoregressive_flow`, `tfb_masked_dense`, `tfb_matrix_inverse_tri_l`, `tfb_matvec_lu`, `tfb_normal_cdf`, `tfb_ordered`, `tfb_pad`, `tfb_permute`, `tfb_power_transform`, `tfb_rational_quadratic_spline`, `tfb_rayleigh_cdf`, `tfb_real_nvp_default_template`, `tfb_real_nvp`, `tfb_reciprocal`, `tfb_reshape`, `tfb_scale_matvec_diag`, `tfb_scale_matvec_linear_operator`, `tfb_scale_matvec_lu`, `tfb_scale_matvec_tri_l`, `tfb_scale_tri_l`, `tfb_shift`, `tfb_sigmoid`, `tfb_sinh_arcsinh`, `tfb_sinh`, `tfb_softmax_centered`, `tfb_softplus`, `tfb_softsign`, `tfb_split`, `tfb_square`, `tfb_tanh`, `tfb_transform_diagonal`, `tfb_transpose`, `tfb_weibull_cdf`, `tfb_weibull`
tfb_lambert_w_tail

Value

A bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(),
tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()

Description

A random variable $Y$ has a Lambert $W \times F$ distribution if $W_{\tau}(Y) = X$ has distribution $F$, where
$\tau = (\text{shift}, \text{scale}, \text{tail})$ parameterizes the inverse transformation.

Usage

tfb_lambert_w_tail(  
    shift = NULL,  
    scale = NULL,  
    tailweight = NULL,  
    validate_args = FALSE,  
    name = "lambertw_tail"  
)

Arguments

- **shift**: Floating point tensor; the shift for centering (uncentering) the input (output) random variable(s).
- **scale**: Floating point tensor; the scaling (unscaling) of the input (output) random variable(s). Must contain only positive values.
tailweight  Floating point tensor; the tail behaviors of the output random variable(s). Must contain only non-negative values.
validate_args  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
name  name prefixed to Ops created by this class.

Details
This bijector defines the transformation underlying Lambert \( W \times F \) distributions that transform an input random variable to an output random variable with heavier tails. It is defined as

\[ Y = (U \times \exp(0.5 \times \text{tail} \times U^2)) \times \text{scale} + \text{shift}, \quad \text{tail} \geq 0 \]

where \( U = (X - \text{shift}) / \text{scale} \) is a shifted/scaled input random variable, and \( \text{tail} \geq 0 \) is the tail parameter.

Attributes:
- shift: shift to center (uncenter) the input data.
- scale: scale to normalize (de-normalize) the input data.
- tailweight: Tail parameter \( \Delta \) of heavy-tail transformation; must be \( \geq 0 \).

Value
a bijector instance.

See Also
For usage examples see \( \text{tfb\_forward()} \), \( \text{tfb\_inverse()} \), \( \text{tfb\_inverse\_log\_det\_jacobian()} \).

Other bijectors:
- \( \text{tfb\_absolute\_value()} \)
- \( \text{tfb\_affine\_linear\_operator()} \)
- \( \text{tfb\_affine\_scalar()} \)
- \( \text{tfb\_ascending()} \)
- \( \text{tfb\_batch\_normalization()} \)
- \( \text{tfb\_blockwise()} \)
- \( \text{tfb\_chain()} \)
- \( \text{tfb\_cholesky\_outer\_product()} \)
- \( \text{tfb\_cholesky\_to\_inv\_cholesky()} \)
- \( \text{tfb\_correlation\_cholesky()} \)
- \( \text{tfb\_cumsum()} \)
- \( \text{tfb\_discrete\_cosine\_transform()} \)
- \( \text{tfb\_expm1()} \)
- \( \text{tfb\_exp()} \)
- \( \text{tfb\_ffjord()} \)
- \( \text{tfb\_fill\_scale\_tri\_l()} \)
- \( \text{tfb\_fill\_triangular()} \)
- \( \text{tfb\_glow()} \)
- \( \text{tfb\_gompertz\_cdf()} \)
- \( \text{tfb\_gumbel\_cdf()} \)
- \( \text{tfb\_kumaraswamy\_cdf()} \)
- \( \text{tfb\_kumaraswamy()} \)
- \( \text{tfb\_masked\_autoregressive\_default\_template()} \)
- \( \text{tfb\_masked\_autoregressive\_flow()} \)
- \( \text{tfb\_masked\_dense()} \)
- \( \text{tfb\_matrix\_inverse\_tri\_l()} \)
- \( \text{tfb\_matvec\_lu()} \)
- \( \text{tfb\_normal\_cdf()} \)
- \( \text{tfb\_ordered()} \)
- \( \text{tfb\_pad()} \)
- \( \text{tfb\_permute()} \)
- \( \text{tfb\_power\_transform()} \)
- \( \text{tfb\_rational\_quadratic\_spline()} \)
- \( \text{tfb\_rayleigh\_cdf()} \)
- \( \text{tfb\_real\_nvp\_default\_template()} \)
- \( \text{tfb\_real\_nvp()} \)
- \( \text{tfb\_reciprocal()} \)
- \( \text{tfb\_reshape()} \)
- \( \text{tfb\_scale\_matvec\_diag()} \)
- \( \text{tfb\_scale\_matvec\_linear\_operator()} \)
- \( \text{tfb\_scale\_matvec\_lu()} \)
- \( \text{tfb\_scale\_matvec\_tri\_l()} \)
- \( \text{tfb\_scale\_tri\_l()} \)
- \( \text{tfb\_scale()} \)
- \( \text{tfb\_shifted\_gompertz\_cdf()} \)
- \( \text{tfb\_shift()} \)
- \( \text{tfb\_sigmoid()} \)
- \( \text{tfb\_sinh\_arcsinh()} \)
- \( \text{tfb\_softmax\_centered()} \)
- \( \text{tfb\_softplus()} \)
- \( \text{tfb\_softsign()} \)
- \( \text{tfb\_split()} \)
- \( \text{tfb\_square()} \)
- \( \text{tfb\_tanh()} \)
- \( \text{tfb\_transform\_diagonal()} \)
- \( \text{tfb\_transpose()} \)
- \( \text{tfb\_weibull\_cdf()} \)
- \( \text{tfb\_weibull()} \)

---

tfb\_masked\_autoregressive\_default\_template

\textit{Masked Autoregressive Density Estimator}

Description
This will be wrapped in a make_template to ensure the variables are only created once. It takes the input and returns the loc ("mu" in Germain et al. (2015)) and log_scale ("alpha" in Germain et al. (2015)) from the MADE network.
Usage

tfb_masked_autoregressive_default_template(
    hidden_layers,
    shift_only = FALSE,
    activation = tf$nn$relu,
    log_scale_min_clip = -5,
    log_scale_max_clip = 3,
    log_scale_clip_gradient = FALSE,
    name = NULL,
    ...
)

Arguments

hidden_layers  list-like of non-negative integer, scalars indicating the number of units in each hidden layer. Default: list(512,512).
shift_only     logical indicating if only the shift term shall be computed. Default: FALSE.
activation     Activation function (callable). Explicitly setting to NULL implies a linear activation.
log_scale_min_clip  float-like scalar Tensor, or a Tensor with the same shape as log_scale. The minimum value to clip by. Default: -5.
log_scale_max_clip  float-like scalar Tensor, or a Tensor with the same shape as log_scale. The maximum value to clip by. Default: 3.
log_scale_clip_gradient  logical indicating that the gradient of tf$clip_by_value should be preserved. Default: FALSE.
name           A name for ops managed by this function. Default: "tfb_masked_autoregressive_default_template".
...

Details

Warning: This function uses masked_dense to create randomly initialized tf$Variables. It is presumed that these will be fit, just as you would any other neural architecture which uses tf$layers$dense.

About Hidden Layers Each element of hidden_layers should be greater than the input_depth (i.e., input_depth = tf$shape(input)[-1] where input is the input to the neural network). This is necessary to ensure the autoregressivity property.

About Clipping This function also optionally clips the log_scale (but possibly not its gradient). This is useful because if log_scale is too small/large it might underflow/overflow making it impossible for the MaskedAutoregressiveFlow bijector to implement a bijection. Additionally, the log_scale_clip_gradient bool indicates whether the gradient should also be clipped. The default does not clip the gradient; this is useful because it still provides gradient information (for fitting) yet solves the numerical stability problem. I.e., log_scale_clip_gradient = FALSE means grad[exp(clip(x))] = grad[x] exp(clip(x)) rather than the usual grad[clip(x)] exp(clip(x)).
Affine MaskedAutoregressiveFlow bijector

Description

The affine autoregressive flow (Papamakarios et al., 2016) provides a relatively simple framework for user-specified (deep) architectures to learn a distribution over continuous events. Regarding terminology,

Usage

```
tfb_masked_autoregressive_flow(  
    shift_and_log_scale_fn,  
    is_constant_jacobian = FALSE,  
)```
unroll_loop = FALSE,
event_ndims = 1L,
validate_args = FALSE,
name = NULL
)

Arguments

shift_and_log_scale_fn
Function which computes shift and log_scale from both the forward domain (x) and the inverse domain (y). Calculation must respect the "autoregressive property". Suggested default: tfb_masked_autoregressive_default_template(hidden_layers=...). Typically the function contains tf$Variables and is wrapped using tf$make_template. Returning NULL for either (both) shift, log_scale is equivalent to (but more efficient than) returning zero.

is_constant_jacobian
Logical, default: FALSE. When TRUE the implementation assumes log_scale does not depend on the forward domain (x) or inverse domain (y) values. (No validation is made; is_constant_jacobian=FALSE is always safe but possibly computationally inefficient.)

unroll_loop
Logical indicating whether the tf$while_loop in _forward should be replaced with a static for loop. Requires that the final dimension of x be known at graph construction time. Defaults to FALSE.

event_ndims
integer, the intrinsic dimensionality of this bijector. 1 corresponds to a simple vector autoregressive bijector as implemented by the tfb_masked_autoregressive_default_template, 2 might be useful for a 2D convolutional shift_and_log_scale_fn and so on.

validate_args
Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name
name prefixed to Ops created by this class.

Details

"Autoregressive models decompose the joint density as a product of conditionals, and model each conditional in turn. Normalizing flows transform a base density (e.g. a standard Gaussian) into the target density by an invertible transformation with tractable Jacobian." (Papamakarios et al., 2016)

In other words, the "autoregressive property" is equivalent to the decomposition, \( p(x) = \prod_{i=0}^{d} p(x[i] | x[0:i]) \). In the simple case where the permutation is identity this reduces to:

\[
 p(x) = \prod_{i=0}^{d} p(x[i] | x[0:i])
\]

The provided shift_and_log_scale_fn, tfb_masked_autoregressive_default_template, achieves this property by zeroing out weights in its masked_dense layers. In TensorFlow Probability, "normalizing flows" are implemented as tfp.bijectors.Bijectors. The forward "autoregression" is implemented using a tf.while_loop and a deep neural network (DNN) with masked weights such that the autoregressive property is automatically met in the inverse. A TransformedDistribution using MaskedAutoregressiveFlow(...) uses the (expensive) forward-mode calculation to draw samples and the (cheap) reverse-mode calculation to compute log-probabilities. Conversely, a TransformedDistribution using Invert(MaskedAutoregressiveFlow(...)) uses the (expensive) forward-mode calculation to compute log-probabilities and the (cheap) reverse-mode calculation to compute samples.
Given a shift_and_log_scale_fn, the forward and inverse transformations are (a sequence of) affine transformations. A "valid" shift_and_log_scale_fn must compute each shift (aka loc or "mu" in Germain et al. (2015)) and log(scale) (aka "alpha" in Germain et al. (2015)) such that each are broadcastable with the arguments to forward and inverse, i.e., such that the calculations in forward, inverse below are possible.

For convenience, tfb_masked_autoregressive_default_template is offered as a possible shift_and_log_scale_fn function. It implements the MADE architecture (Germain et al., 2015). MADE is a feed-forward network that computes a shift and log(scale) using masked_dense layers in a deep neural network. Weights are masked to ensure the autoregressive property. It is possible that this architecture is suboptimal for your task. To build alternative networks, either change the arguments to tfb_masked_autoregressive_default_template, use the masked_dense function to roll-out your own, or use some other architecture, e.g., using tf.layers. Warning: no attempt is made to validate that the shift_and_log_scale_fn enforces the "autoregressive property".

Assuming shift_and_log_scale_fn has valid shape and autoregressive semantics, the forward transformation is

```python
def forward(x):
    y = zeros_like(x)
    event_size = x.shape[-event_dims:].num_elements()
    for _ in range(event_size):
        shift, log_scale = shift_and_log_scale_fn(y)
        y = x * tf.exp(log_scale) + shift
    return y
```

and the inverse transformation is

```python
def inverse(y):
    shift, log_scale = shift_and_log_scale_fn(y)
    return (y - shift) / tf.exp(log_scale)
```

Notice that the inverse does not need a for-loop. This is because in the forward pass each calculation of shift and log_scale is based on the y calculated so far (not x). In the inverse, the y is fully known, thus is equivalent to the scaling used in forward after event_size passes, i.e., the "last" y used to compute shift, log_scale. (Roughly speaking, this also proves the transform is bijective.)

Value

a bijector instance.

References

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

tfb_masked_dense  

**Autoregressively masked dense layer**

**Description**

Analogous to `tf$layers$dense`.

**Usage**

```r
tfb_masked_dense(
  inputs,
  units,
  num_blocks = NULL,
  exclusive = FALSE,
  kernel_initializer = NULL,
  reuse = NULL,
  name = NULL,
  ...
)
```

**Arguments**

- `inputs`: Tensor input.
- `units`: integer scalar representing the dimensionality of the output space.
- `num_blocks`: integer scalar representing the number of blocks for the MADE masks.
- `exclusive`: logical scalar representing whether to zero the diagonal of the mask, used for the first layer of a MADE.
tfb_masked_dense

```
kernel_initializer
  Initializer function for the weight matrix. If NULL (default), weights are initialized using the tf$glorot_random_initializer

reuse
  logical scalar representing whether to reuse the weights of a previous layer by the same name.

name
  string used to describe ops managed by this function.

... tf$layers$dense arguments

Details

See Germain et al. (2015) for detailed explanation.

Value

tensor

References


See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
  tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
  tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
  tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(),
  tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
  tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(),
  tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
  tfb_masked_autoregressive_flow(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(),
  tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(),
  tfb_rayleigh_cdf(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(),
  tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(),
  tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
  tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
  tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
  tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
```
tfb_matrix_inverse_tri_l

Computes $g(L) = \text{inv}(L)$, where $L$ is a lower-triangular matrix.

Description

$L$ must be nonsingular; equivalently, all diagonal entries of $L$ must be nonzero. The input must have rank $\geq 2$. The input is treated as a batch of matrices with batch shape $\text{input.shape}[:-2]$, where each matrix has dimensions $\text{input.shape}[:-2]$ by $\text{input.shape}[-1]$ (hence $\text{input.shape}[:-2]$ must equal $\text{input.shape}[-1]$).

Usage

```r
tfb_matrix_inverse_tri_l(validate_args = FALSE, name = "matrix_inverse_tril")
```

Arguments

- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_matvec_lu

\textit{Matrix-vector multiply using LU decomposition}

\textbf{Description}

This bijector is identical to the "Convolution1x1" used in Glow (Kingma and Dhariwal, 2018).

\textbf{Usage}

\begin{verbatim}
tfb_matvec_lu(lower_upper, permutation, validate_args = FALSE, name = NULL)
\end{verbatim}

\textbf{Arguments}

- \textit{lower_upper:} The LU factorization as returned by \texttt{tf$linalg$lu}.
- \textit{permutation:} The LU factorization permutation as returned by \texttt{tf$linalg$lu}.
- \textit{validate_args:} Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- \textit{name:} name prefixed to Ops created by this class.

\textbf{Details}

Warning: this bijector never verifies the scale matrix (as parameterized by LU decomposition) is invertible. Ensuring this is the case is the caller's responsibility.

\textbf{Value}

a bijector instance.

\textbf{References}


\textbf{See Also}

For usage examples see \texttt{tfb_forward()}, \texttt{tfb_inverse()}, \texttt{tfb_inverse_log_det_jacobian}().

Other bijectors: \texttt{tfb_absolute_value()}, \texttt{tfb_affine_linear_operator()}, \texttt{tfb_affine_scalar()}, \texttt{tfb_affine()}, \texttt{tfbAscending()}, \texttt{tfb_batch_normalization()}, \texttt{tfb_blockwise()}, \texttt{tfb_chain()}, \texttt{tfb_cholesky_outer_product()}, \texttt{tfb_cholesky_to_inv_cholesky()}, \texttt{tfb_correlation_cholesky()}, \texttt{tfb_cusum()}, \texttt{tfb_discrete_cosine_transform()}, \texttt{tfb_expm1()}, \texttt{tfb_exp()}, \texttt{tfb_ffjord()}, \texttt{tfb_fill_scale_tri_l()}, \texttt{tfb_fill_triangular()}, \texttt{tfb_glow()}, \texttt{tfb_gompertz_cdf()}, \texttt{tfb_gumbel_cdf()}, \texttt{tfb_gumbel()}, \texttt{tfb_identity()}, \texttt{tfb_inline()}, \texttt{tfb_invert()}, \texttt{tfb_iterated_sigmoid_centered()}, \texttt{tfb_kumaraswamy_cdf()}, \texttt{tfb_kumaraswamy()}, \texttt{tfb_lambert_w_tail()}, \texttt{tfb_masked_autoregressive_default_template()}, \texttt{tfb_masked_autoregressive_flow()}, \texttt{tfb_masked_dense()}, \texttt{tfb_matrix_inverse_tri_l()}, \texttt{tfb_normal_cdf()}, \texttt{tfb_ordered()}, \texttt{tfb_pad()}, \texttt{tfb_permute()}, \texttt{tfb_power_transform()}, \texttt{tfb_rational_quadratic()}, \texttt{tfb_rayleigh_cdf()}, \texttt{tfb_real_nvp_default_template()}, \texttt{tfb_real_nvp()}, \texttt{tfb_reciprocal}().
tfb_normal_cdf

Computes $Y = g(X) = \text{NormalCDF}(x)$

**Description**

This bijector maps inputs from $[-\infty, \infty]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Normal distribution:

**Usage**

```r
tfb_normal_cdf(validate_args = FALSE, name = "normal")
```

**Arguments**

- **validate_args**
  Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name**
  name prefixed to Ops created by this class.

**Details**

$Y \sim \text{Normal}(0, 1)$

$$pdf(y; 0., 1.) = \frac{1}{\sqrt{2 \pi}} \exp\left(-\frac{y^2}{2}\right)$$

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_ascending()`, `tfb_att_momentum()`, `tfb autofill()`, `tfb_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_softplus()`, `tfb_sigmoid()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()

tfb_ordered

**Bijector which maps a tensor** \( x_k \) **that has increasing elements in the**
last dimension **to an unconstrained tensor** \( y_k \)

**Description**

Both the domain and the codomain of the mapping is \([-inf, inf]\), however, the input of the forward
mapping must be strictly increasing. The inverse of the bijector applied to a normal random vector
\( y \sim N(0,1) \) gives back a sorted random vector with the same distribution \( x \sim N(0,1) \) where \( x = \)
sort(\( y \))

**Usage**

```
tfb_ordered(validate_args = FALSE, name = "ordered")
```

**Arguments**

- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args
  is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name** name prefixed to Ops created by this class.

**Details**

On the last dimension of the tensor, Ordered bijector performs: \( y[0] = x[0] \)
\( y[1:] = \text{tf}\log(x[1:] - x[:-1]) \)

**Value**

a bijector instance.

**See Also**

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_discrete_cosh_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(),
tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
tfb_transform_diagonal(), tfb_tanh(), tfb_transform_diagonal()
tfb_pad

Pads a value to the event_shape of a Tensor.

Description

The semantics of bijector_pad generally follow that of tf$pad() except that bijector_pad's paddings argument applies to the rightmost dimensions. Additionally, the new argument axis enables overriding the dimensions to which paddings is applied. Like paddings, the axis argument is also relative to the rightmost dimension and must therefore be negative. The argument paddings is a vector of integer pairs each representing the number of left and/or right constant_values to pad to the corresponding rightmost dimensions. That is, unless axis is specified, specifying k different paddings means the rightmost k dimensions will be "grown" by the sum of the respective paddings. When axis is specified, it indicates the dimension to which the corresponding padding is applied. By default, axis is NULL which means it is logically equivalent to range(start=-length(paddings), limit=0), i.e., the rightmost dimensions.

Usage

tfb_pad(
  paddings = list(c(0, 1)),
  mode = "CONSTANT",
  constant_values = 0,
  axis = NULL,
  validate_args = FALSE,
  name = NULL
)

Arguments

paddings A vector-shaped Tensor of integer pairs representing the number of elements to pad on the left and right, respectively. Default value: list(reticulate::tuple(0L,1L)).
mode One of 'CONSTANT', 'REFLECT', or 'SYMMETRIC' (case-insensitive). For more details, see tf$pad.
constant_values In "CONSTANT" mode, the scalar pad value to use. Must be same type as tensor. For more details, see tf$pad.
axis The dimensions for which paddings are applied. Must be 1:1 with paddings or NULL. Default value: NULL (i.e., tf$range(start = -length(paddings), limit = 0)).
validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
name name prefixed to Ops created by this class.
**tfb_permute**

Permutes the rightmost dimension of a Tensor

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tril()`, `tfb_scale_tril()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()

---

**Description**

Permutes the rightmost dimension of a Tensor

**Usage**

```r
tfb_permute(permutation, axis = -1L, validate_args = FALSE, name = NULL)
```

**Arguments**

- **permutation**: An integer-like vector-shaped Tensor representing the permutation to apply to the axis dimension of the transformed Tensor.
- **axis**: Scalar integer Tensor representing the dimension over which to `tf$gather`. axis must be relative to the end (reading left to right) thus must be negative. Default value: -1 (i.e., right-most).
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Value**

a bijector instance.
tfb_power_transform

 Computes \( Y = g(X) = (1 + X \cdot c)^{1/c} \), where \( X \geq -1/c \)

**Description**

The power transform maps inputs from [0, inf] to [-1/c, inf]; this is equivalent to the inverse of this bijector. This bijector is equivalent to the Exp bijector when \( c=0 \).

**Usage**

```python
tfb_power_transform(power, validate_args = FALSE, name = "power_transform")
```

**Arguments**

- **power** float scalar indicating the transform power, i.e., \( Y = g(X) = (1 + X \cdot c)^{1/c} \) where \( c \) is the power.
- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name** name prefixed to Ops created by this class.

**Value**
a bijector instance.
tfb_rational_quadratic_spline

A piecewise rational quadratic spline, as developed in Conor et al.(2019).

Description

This transformation represents a monotonically increasing piecewise rational quadratic function. Outside of the bounds of knot_x/knot_y, the transform behaves as an identity function.

Usage

```r
tfb_rational_quadratic_spline(
  bin_widths,
  bin_heights,
  knot_slopes,
  range_min = -1,
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- **bin_widths**: The widths of the spans between subsequent knot_x positions, a floating point Tensor. Must be positive, and at least 1-D. Innermost axis must sum to the same value as bin_heights. The knot_x positions will be a first at range_min, followed by knots at range_min + cumsum(bin_widths,axis=-1).
bin_heights

The heights of the spans between subsequent knot y positions, a floating point Tensor. Must be positive, and at least 1-D. Innermost axis must sum to the same value as bin_widths. The knot y positions will be a first at range_min, followed by knots at range_min + cumsum(bin_heights, axis=-1).

knot_slopes

The slope of the spline at each knot, a floating point Tensor. Must be positive. 1s are implicitly padded for the first and last implicit knots corresponding to range_min and range_min + sum(bin_widths, axis=-1). Innermost axis size should be 1 less than that of bin_widths/bin_heights, or 1 for broadcasting.

range_min

The x/y position of the first knot, which has implicit slope 1. range_max is implicit, and can be computed as range_min + sum(bin_widths, axis=-1). Scalar floating point Tensor.

validate_args

Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name

name prefixed to Ops created by this class.

Details

Typically this bijector will be used as part of a chain, with splines for trailing x dimensions conditioned on some of the earlier x dimensions, and with the inverse then solved first for unconditioned dimensions, then using conditioning derived from those inverses, and so forth.

For each argument, the innermost axis indexes bins/knots and batch axes index axes of x/y spaces. A RationalQuadraticSpline with a separate transform for each of three dimensions might have bin_widths shaped [3, 32]. To use the same spline for each of x’s three dimensions we may broadcast against x and use a bin_widths parameter shaped [32].

Parameters will be broadcast against each other and against the input x/ys, so if we want fixed slopes, we can use kwarg knot_slopes=1. A typical recipe for acquiring compatible bin widths and heights would be:

```r
nbins <- unconstrained_vector$shape[-1]
range_min <- 1
range_max <- 1
min_bin_size = 1e-2
scale <- range_max - range_min - nbins * min_bin_size
bin_widths = tf$math$softmax(unconstrained_vector) * scale + min_bin_size
```

Value

a bijector instance.

References

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expml()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shift()`, `tfb_shifted_gompertz_cdf()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

tfb_rayleigh_cdf

*Compute* $Y = g(X) = 1 - \exp\left(-\frac{(X/scale)^2}{2}\right)$, $X \geq 0$.

---

**Description**

This bijector maps inputs from $[0, \infty)$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim \mathcal{U}(0, 1)$ gives back a random variable with the Rayleigh distribution:

$$Y \sim \text{Rayleigh}(\text{scale})$$

$$\text{pdf}(y; \text{scale}, y \geq 0) = \left(\frac{1}{\text{scale}}\right) \times (y / \text{scale}) \times \exp(-\frac{(y / \text{scale})^2}{2})$$

**Usage**

`tfb_rayleigh_cdf(scale, validate_args = FALSE, name = "rayleigh_cdf")`

**Arguments**

- `scale` Positive floating-point tensor. This is 1 in $Y = g(X) = 1 - \exp\left(-\frac{(X/scale)^2}{2}\right)$, $X \geq 0$.
- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Details**

Likewise, the forward of this bijector is the Rayleigh distribution CDF.

**Value**

a bijector instance.
**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfbreshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tril()`, `tfb_scale_tril()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfbTranspose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

**tfb_real_nvp**

*RealNVP affine coupling layer for vector-valued events*

**Description**

Real NVP models a normalizing flow on a D-dimensional distribution via a single D-d-dimensional conditional distribution (Dinh et al., 2017): 
\[
y_{[d:D]} = x_{[d:D]} \ast \exp(\log_{scale}_{fn}(x_{[0:d]})) + \text{shift}_{fn}(x_{[0:d]}) y_{[0:d]} = x_{[0:d]}
\]
The last D-d units are scaled and shifted based on the first d units only, while the first d units are ’masked’ and left unchanged. Real NVP’s shift_and_log_scale_fn computes vector-valued quantities. For scale-and-shift transforms that do not depend on any masked units, i.e. d=0, use the tfb_affine bijector with learned parameters instead. Masking is currently only supported for base distributions with event_ndims=1. For more sophisticated masking schemes like checkerboard or channel-wise masking (Papamakarios et al., 2016), use the tfb_permute bijector to re-order desired masked units into the first d units. For base distributions with event_ndims > 1, use the tfb_reshape bijector to flatten the event shape.

**Usage**

```python
tfb_real_nvp(
    num_masked,
    shift_and_log_scale_fn,
    is_constant_jacobian = FALSE,
    validate_args = FALSE,
    name = NULL
)
```
Arguments

num_masked integer indicating that the first d units of the event should be masked. Must be in the closed interval [1, D-1], where D is the event size of the base distribution.

shift_and_log_scale_fn
Function which computes shift and log_scale from both the forward domain (x) and the inverse domain (y). Calculation must respect the "autoregressive property". Suggested default: tfb_real_nvp_default_template(hidden_layers=...).
Typically the function contains tfVariables and is wrapped using tf$make_template. Returning NULL for either (both) shift, log_scale is equivalent to (but more efficient than) returning zero.

is_constant_jacobian
Logical, default: FALSE. When TRUE the implementation assumes log_scale does not depend on the forward domain (x) or inverse domain (y) values. (No validation is made; is_constant_jacobian=FALSE is always safe but possibly computationally inefficient.)

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Details

Recall that the MAF bijector (Papamakarios et al., 2016) implements a normalizing flow via an autoregressive transformation. MAF and IAF have opposite computational tradeoffs - MAF can train all units in parallel but must sample units sequentially, while IAF must train units sequentially but can sample in parallel. In contrast, Real NVP can compute both forward and inverse computations in parallel. However, the lack of an autoregressive transformations makes it less expressive on a per-bijector basis.

A "valid" shift_and_log_scale_fn must compute each shift (aka loc or "mu" in Papamakarios et al. (2016) and log(scale) (aka "alpha" in Papamakarios et al. (2016)) such that each are broadcastable with the arguments to forward and inverse, i.e., such that the calculations in forward, inverse below are possible. For convenience, real_nvp_default_nvp is offered as a possible shift_and_log_scale_fn function.

NICE (Dinh et al., 2014) is a special case of the Real NVP bijector which discards the scale transformation, resulting in a constant-time inverse-log-determinant-Jacobian. To use a NICE bijector instead of Real NVP, shift_and_log_scale_fn should return (shift, NULL), and is_constant_jacobian should be set to TRUE in the RealNVP constructor. Calling tfb_real_nvp_default_template with shift_only=TRUE returns one such NICE-compatible shift_and_log_scale_fn.

Caching: the scalar input depth D of the base distribution is not known at construction time. The first call to any of forward(x), inverse(x), inverse_log_det_jacobian(x), or forward_log_det_jacobian(x) memoizes D, which is re-used in subsequent calls. This shape must be known prior to graph execution (which is the case if using tf$layers).

Value

a bijector instance.
tfb_real_nvp_default_template

Build a scale-and-shift function using a multi-layer neural network

Description

This will be wrapped in a make_template to ensure the variables are only created once. It takes the d-dimensional input \(x[0:d]\) and returns the \(D-d\) dimensional outputs loc ("mu") and log_scale ("alpha").

Usage

```r
tfb_real_nvp_default_template(
  hidden_layers,
  shift_only = FALSE,
  activation = tf$nn$relu,
)```

References

name = NULL,

Arguments

- **hidden_layers**: list-like of non-negative integer, scalars indicating the number of units in each hidden layer. Default: list(512, 512).
- **shift_only**: logical indicating if only the shift term shall be computed (i.e., NICE bijector). Default: FALSE.
- **activation**: Activation function (callable). Explicitly setting to NULL implies a linear activation.
- **name**: A name for ops managed by this function. Default: "tfb_real_nvp_default_template".

Details

The default template does not support conditioning and will raise an exception if condition_kwargs are passed to it. To use conditioning in real nvp bijector, implement a conditioned shift/scale template that handles the condition_kwargs.

Value

- shift: Float-like Tensor of shift terms
- log_scale: Float-like Tensor of log(scale) terms

References


See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfbAscending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exponential()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matrix_involve()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`,
tfb_reciprocal

A Bijector that computes \( b(x) = 1. / x \)

Description
A Bijector that computes \( b(x) = 1. / x \)

Usage
```
tfb_reciprocal(validate_args = FALSE, name = "reciprocal")
```

Arguments
- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

Value
a bijector instance.

See Also
For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matrix_tril()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
tfb_reshape

Reshapes the event_shape of a Tensor

Description

The semantics generally follow that of `tf$reshape()`, with a few differences:

- The user must provide both the input and output shape, so that the transformation can be inverted. If an input shape is not specified, the default assumes a vector-shaped input, i.e., `event_shape_in = list(-1)`.
- The Reshape bijector automatically broadcasts over the leftmost dimensions of its input (sample_shape and batch_shape); only the rightmost `event_ndims_in` dimensions are reshaped. The number of dimensions to reshape is inferred from the provided `event_shape_in` (`event_ndims_in = length(event_shape_in)`).

Usage

```r
tfb_reshape(
  event_shape_out,
  event_shape_in = c(-1),
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- `event_shape_out` An integer-like vector-shaped Tensor representing the event shape of the transformed output.
- `event_shape_in` An optional integer-like vector-shape Tensor representing the event shape of the input. This is required in order to define inverse operations; the default of `list(-1)` assumes a vector-shaped input.
- `validate_args` Logical, default `FALSE`. Whether to validate input with asserts. If `validate_args` is `FALSE`, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`.
tfb_scale

Compute $Y = g(X; \text{scale}) = \text{scale} \times X$.

Description

Examples:

```r
Y <- 2 * X
b <- tfb_scale(scale = 2)
```

Usage

```r
tfb_scale(
  scale = NULL,
  log_scale = NULL,
  validate_args = FALSE,
  name = "scale"
)
```

Arguments

- **scale**: Floating-point Tensor.
- **log_scale**: Floating-point Tensor. Logarithm of the scale. If this is set to NULL, no scale is applied. This should not be set if scale is set.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.
tfb_scale_matvec_diag

**Compute** \( Y = g(X; \text{scale}) = \text{scale} @ X \)

**Description**

In TF parlance, the scale term is logically equivalent to:

\[
\text{scale} = \text{tf} \cdot \text{diag} \left( \text{scale}_\text{diag} \right)
\]

The scale term is applied without materializing a full dense matrix.

**Usage**

```r
tfb_scale_matvec_diag(
  scale_diag,
  adjoint = FALSE,
  validate_args = FALSE,
  name = "scale_matvec_diag",
  dtype = NULL
)
```

**Arguments**

- **scale_diag**
  
  Floating-point Tensor representing the diagonal matrix. scale_diag has shape \([N1, N2, \ldots, k]\), which represents a \(k \times k\) diagonal matrix.

- **adjoint**
  
  logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`
validate_args (Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.)

name (name prefixed to Ops created by this class.)

dtype (tf$DType to prefer when converting args to Tensors. Else, we fall back to a common dtype inferred from the args, finally falling back to float32.)

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
    tfb_affine(), tfb_ascend(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
    tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
    tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(),
    tfb_fill_scale_tril(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
    tfb_gumbel(), tfb_identity(), tfb_inplace(), tfb_invert(), tfb_iterated_sigmoid_centered(),
    tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
    tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tril(),
    tfb_matvec_tri_l(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
    tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
    tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_linear_operator(),
    tfb_scale_matvec_tril(), tfb_scale_tri_l(), tfb_shifted_gompertz_cdf(),
    tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
    tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
    tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
tfb_scale_matvec_lu

Matrix-vector multiply using LU decomposition.

Arguments

- **scale**: Subclass of LinearOperator. Represents the (batch, non-singular) linear transformation by which the Bijector transforms inputs.
- **adjoint**: logical indicating whether to use the scale matrix as specified or its adjoint. Default value: FALSE.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expm1(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_lu(), tfb_shifted_gompertz_cdf(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(), tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(), tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()

description

This bijector is identical to the "Convolution1x1" used in Glow (Kingma and Dhariwal, 2018).

Usage

tfb_scale_matvec_lu(
    lower_upper,
    permutation,
    validate_args = FALSE,
    name = NULL
)
tfb_scale_matvec_tri_l

**Arguments**

- `lower_upper`: The LU factorization as returned by `tf$linalg$lu`.
- `permutation`: The LU factorization permutation as returned by `tf$linalg$lu`.
- `validate_args`: Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name`: name prefixed to Ops created by this class.

**Value**

a bijector instance.

**References**


**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`.

**Description**

The `scale` term is presumed lower-triangular and non-singular (i.e., no zeros on the diagonal), which permits efficient determinant calculation (linear in matrix dimension, instead of cubic).
Usage

tfb_scale_matvec_tri_l(
    scale_tril,
    adjoint = FALSE,
    validate_args = FALSE,
    name = "scale_matvec_tri_l",
    dtype = NULL
)

Arguments

scale_tril Floating-point Tensor representing the lower triangular matrix. scale_tril
has shape [N1, N2, ... k, k], which represents a k x k lower triangular matrix.
When NULL no scale_tril term is added to scale. The upper triangular ele-
ments above the diagonal are ignored.

adjoint logical indicating whether to use the scale matrix as specified or its adjoint.
Note that lower-triangularity is taken into account first: the region above the di-
agonal of scale_tril is treated as zero (irrespective of the adjoint setting). A
lower-triangular input with adjoint=TRUE will behave like an upper triangular
transform. Default value: FALSE.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args
is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

dtype tf$DType to prefer when converting args to
Tensor s. Else, we fall back to a
common dtype inferred from the args, finally falling back to float32.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().
Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_log_cosh_transform(), tfb_logistic(), tfb_exp(), tfb_exp(), tfb_ffjord(),
tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
tfb_gumbel(), tfb_identity(), tfb_inverse(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_templ,
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),
tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),
tfb_transpose(), tfb_weibull_cdf(), tfb_weibull()
tfb_scale_tri_l

Transforms unconstrained vectors to TriL matrices with positive diagonal

Description

This is implemented as a simple tfb_chain of tfb_fill_triangular followed by tfb_transform_diagonal, and provided mostly as a convenience. The default setup is somewhat opinionated, using a Softplus transformation followed by a small shift (1e-5) which attempts to avoid numerical issues from zeros on the diagonal.

Usage

```
    tfb_scale_tri_l(
        diag_bijector = NULL,
        diag_shift = 1e-05,
        validate_args = FALSE,
        name = "scale_tril"
    )
```

Arguments

- **diag_bijector**: Bijector instance, used to transform the output diagonal to be positive. Default value: NULL (i.e., tfb_softplus()).
- **diag_shift**: Float value broadcastable and added to all diagonal entries after applying the diag_bijector. Setting a positive value forces the output diagonal entries to be positive, but prevents inverting the transformation for matrices with diagonal entries less than this value. Default value: 1e-5.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfbAscending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_pdf(), tfb_gumbel_pdf(), tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_pdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template()
**tfb_shift**

Compute \( Y = g(X; \text{shift}) = X + \text{shift} \).

**Description**

where \( \text{shift} \) is a numeric Tensor.

**Usage**

\[
\text{tfb_shift}(\text{shift}, \text{validate}_\text{args} = \text{FALSE}, \text{name} = \text{"shift"})
\]

**Arguments**

- \( \text{shift} \) floating-point tensor
- \( \text{validate}_\text{args} \) Logical, default \( \text{FALSE} \). Whether to validate input with asserts. If \( \text{validate}_\text{args} \) is \( \text{FALSE} \), and the inputs are invalid, correct behavior is not guaranteed.
- \( \text{name} \) name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see \( \text{tfb_forward()} \), \( \text{tfb_inverse()} \), \( \text{tfb_inverse_log_det_jacobian()} \).

Other bijectors: \( \text{tfb_absolute_value()} \), \( \text{tfb_affine_linear_operator()} \), \( \text{tfb_affine_scalar()} \), \( \text{tfb_affine()} \), \( \text{tfb_ascending()} \), \( \text{tfb_batch_normalization()} \), \( \text{tfb_blockwise()} \), \( \text{tfb_chain()} \), \( \text{tfb_cholesky_outer_product()} \), \( \text{tfb_cholesky_to_inv_cholesky()} \), \( \text{tfb_correlation_cholesky()} \), \( \text{tfb_cumsum()} \), \( \text{tfb_discrete_cosine_transform()} \), \( \text{tfb_exp()} \), \( \text{tfb_expml()} \), \( \text{tfb_ffjord()} \), \( \text{tfb_fill_scale_tri_l()} \), \( \text{tfb_fill_triangular()} \), \( \text{tfb_glow()} \), \( \text{tfb_gompertz_cdf()} \), \( \text{tfb_gumbel_cdf()} \), \( \text{tfb_gumbel()} \), \( \text{tfb_identity()} \), \( \text{tfb_inline()} \), \( \text{tfb_invert()} \), \( \text{tfb_iterated_sigmoid_centered()} \), \( \text{tfb_kumaraswamy_cdf()} \), \( \text{tfb_kumaraswamy()} \), \( \text{tfb_lambert_w_tail()} \), \( \text{tfb_masked_autoregressive_default_template()} \), \( \text{tfb_masked_autoregressive_flow()} \), \( \text{tfb_masked_dense()} \), \( \text{tfb_matrix_inverse_tri_l()} \), \( \text{tfb_matvec_lu()} \), \( \text{tfb_normal_cdf()} \), \( \text{tfb_ordered()} \), \( \text{tfb_pad()} \), \( \text{tfb_permute()} \), \( \text{tfb_power_transform()} \), \( \text{tfb_rational_quadratic_spline()} \), \( \text{tfb_rayleigh_cdf()} \), \( \text{tfb_real_nvp_default_template()} \), \( \text{tfb_real_nvp()} \), \( \text{tfb_reciprocal()} \), \( \text{tfb_reshape()} \), \( \text{tfb_scale_matvec_diag()} \), \( \text{tfb_scale_matvec_linear_operator()} \), \( \text{tfb_scale_matvec_lu()} \), \( \text{tfb_scale_matvec_tri_l()} \), \( \text{tfb_scale_tri_l()} \), \( \text{tfb_shifted_gompertz_cdf()} \), \( \text{tfb_shift()} \), \( \text{tfb_sigmoid()} \), \( \text{tfb_sinh_arcsinh()} \), \( \text{tfb_sinh()} \), \( \text{tfb_softmax_centered()} \), \( \text{tfb_softplus()} \), \( \text{tfb_softsign()} \), \( \text{tfb_split()} \), \( \text{tfb_square()} \), \( \text{tfb_tanh()} \), \( \text{tfb_transform_diagonal()} \), \( \text{tfb_transpose()} \), \( \text{tfb_weibull_cdf()} \), \( \text{tfb_weibull()} \).
tfb_shifted_gompertz_cdf

Compute $Y = g(X) = (1 - \exp(-rate \cdot X)) \cdot \exp(-c \cdot \exp(-rate \cdot X))$

Description

This bijector maps inputs from $[-\infty, \infty]$ to $[0, \infty]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Shifted Gompertz distribution:

$Y \sim \text{ShiftedGompertzCDF}(\text{concentration}, \text{rate})$

pdf($y; c, r$) = $r \cdot \exp(-r \cdot y - \exp(-r \cdot y) / c) \times (1 + (1 - \exp(-r \cdot y)) / c)$

Usage

```
tfb_shifted_gompertz_cdf(
    concentration,
    rate,
    validate_args = FALSE,
    name = "shifted_gompertz_cdf"
)
```

Arguments

- **concentration**: Positive Float-like Tensor that is the same dtype and is broadcastable with concentration. This is $c$ in $Y = g(X) = (1 - \exp(-rate \cdot X)) \cdot \exp(-c \cdot \exp(-rate \cdot X))$.
- **rate**: Positive Float-like Tensor that is the same dtype and is broadcastable with concentration. This is rate in $Y = g(X) = (1 - \exp(-rate \cdot X)) \cdot \exp(-c \cdot \exp(-rate \cdot X))$.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Details

Note: Even though this is called `ShiftedGompertzCDF`, when applied to the `Uniform` distribution, this is not the same as applying a `GompertzCDF` with a Shift bijector (i.e. the Shifted Gompertz distribution is not the same as a Gompertz distribution with a location parameter).

Note: Because the Shifted Gompertz distribution concentrates its mass close to zero, for larger rates or larger concentrations, bijector$forward$ will quickly saturate to 1.
tfb_sigmoid

Value

A bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expml()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

**tfb_sigmoid**

Computes \( Y = g(X) = 1 / (1 + \exp(-X)) \)

---

**Description**

Computes \( Y = g(X) = 1 / (1 + \exp(-X)) \)

**Usage**

`tfb_sigmoid(validate_args = FALSE, name = "sigmoid")`

**Arguments**

- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- **name** name prefixed to Ops created by this class.

**Value**

A bijector instance.
See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(),
tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),
tfb_gumbel(), tfb_identity(), tfb_inline(), tfb_invert(), tfb_iterated_sigmoid_centered(),
tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),
tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),
tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),
tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),
tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),
tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),
tfb_shift(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(), tfb_softplus(),
tfb.softsign(), tfb.split(), tfb.square(), tfb.tanh(), tfb.transform_diagonal(), tfb_transpose(),
tfb_weibull_cdf(), tfb_weibull

---

tfb_sinh  

Bijector that computes $Y = \sinh(X)$.

---

Description

Bijector that computes $Y = \sinh(X)$.

Usage

tfb_sinh(validate_args = FALSE, name = "sinh")

Arguments

validate_args  Logical, default FALSE. Whether to validate input with asserts. If validate_args
is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name  name prefixed to Ops created by this class.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(),
tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),
tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),
tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_expml(), tfb_exp(), tfb_ffjord(),
tfb_sinh_arcsinh

Computes $Y = g(X) = \text{Sinh}( (\text{Arcsinh}(X) + \text{skewness}) \times \text{tailweight})$

**Description**

For skewness in (-inf, inf) and tailweight in (0, inf), this transformation is a diffeomorphism of the real line (-inf, inf). The inverse transform is $X = g^{-1}(Y) = \text{Sinh}(\text{Arcsinh}(Y) / \text{tailweight} - \text{skewness})$. The SinhArcsinh transformation of the Normal is described in [Sinh-arcsinh distributions](https://arxiv.org/abs/1703.00564).

**Usage**

tfb_sinh_arcsinh(
    skewness = NULL,
    tailweight = NULL,
    validate_args = FALSE,
    name = "SinhArcsinh"
)

**Arguments**

- **skewness**: Skewness parameter. Float-type Tensor. Default is 0 of type float32.
- **tailweight**: Tailweight parameter. Positive Tensor of same dtype as skewness and broadcastable shape. Default is 1 of type float32.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Details**

This Bijector allows a similar transformation of any distribution supported on (-inf, inf).

**Value**

a bijector instance.
Meaning of the parameters

- If skewness = 0 and tailweight = 1, this transform is the identity.
- Positive (negative) skewness leads to positive (negative) skew.
- Positive skew means, for unimodal X centered at zero, the mode of Y is "tilted" to the right.
- Positive skew means positive values of Y become more likely, and negative values become less likely.
- Larger (smaller) tailweight leads to fatter (thinner) tails.
- Fatter tails mean larger values of |Y| become more likely.
- If X is a unit Normal, tailweight < 1 leads to a distribution that is "flat" around Y = 0, and a very steep drop-off in the tails.
- If X is a unit Normal, tailweight > 1 leads to a distribution more peaked at the mode with heavier tails. To see the argument about the tails, note that for |X| » 1 and |X| » (|skewness| * tailweight), we have Y ≈ 0.5 X^tailweight, we have Y ≈ 0.5 X^tailweight.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().
Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_inverse_log_det_jacobian(), tfb_inverse(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_inverse_log_det_jacobian(), tfb_inverse(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_ascending(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky()...
**Arguments**

- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

**Details**

At first blush it may seem like the Invariance of domain theorem implies this implementation is not a bijection. However, the appended dimension makes the (forward) image non-open and the theorem does not directly apply.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`. Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_independent()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()` ,`tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sin()`, `tfb_sinh()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`

---

**tfb_softplus**

*Computes* \( Y = g(X) = \log[1 + \exp(X)] \)

**Description**

The softplus Bijector has the following two useful properties:

- The domain is the positive real numbers
- softplus(x) approx x, for large x, so it does not overflow as easily as the Exp Bijector.
tfb_softplus

Usage

tfb_softplus(
    hinge_softness = NULL,
    low = NULL,
    validate_args = FALSE,
    name = "softplus"
)

Arguments

hinge_softness Nonzero floating point Tensor. Controls the softness of what would otherwise
be a kink at the origin. Default is 1.0.

low Nonzero floating point tensor, lower bound on output values. Implicitly zero
if NULL. Otherwise, the transformation \( y = \text{softplus}(x) + \text{low} \) is implemented.
This is equivalent to a \( \text{tfb\_chain(\{\text{tfb\_shift(low)}, \text{tfb\_softplus()\}}) \) bijector and is provided for convenience.

validate_args Logical, default FALSE. Whether to validate input with asserts. If validate_args
is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name name prefixed to Ops created by this class.

Details

The optional nonzero hinge_softness parameter changes the transition at zero. With hinge_softness
= c, the bijector is:

\[
\begin{align*}
f_c(x) & := c \times g(x / c) = c \times \log[1 + \exp(x / c)]. \\
\end{align*}
\]

For large \( x \gg 1 \),

\[
\begin{align*}
c \times \log[1 + \exp(x / c)] \approx c \times \log[\exp(x / c)] = x \\
\end{align*}
\]

so the behavior for large \( x \) is the same as the standard softplus.

As \( c > 0 \) approaches 0 from the right, \( f_c(x) \) becomes less and less soft,
approaching \( \max(0, x) \).
* \( c = 1 \) is the default.
* \( c > 0 \) but small means \( f(x) \approx \text{ReLU}(x) = \max(0, x) \).
* \( c < 0 \) flips sign and reflects around the y-axis: \( f_{-c}(x) = -f_c(-x) \).
* \( c = 0 \) results in a non-bijective transformation and triggers an exception.

Note: \( \log(.) \) and \( \exp(.) \) are applied element-wise but the Jacobian is a reduction over the event space.

\[
\begin{align*}
[1 + \exp(x / c)] & : R:1\%20+\%20exp(x\%20/%20c) \\
[1 + \exp(x / c)] & : R:1\%20+\%20exp(x\%20/%20c) \\
[\exp(x / c)] & : R:exp(x\%20/%20c)
\end{align*}
\]
tfb_softsign

Computes \( Y = g(X) = X / (1 + |X|) \)

**Description**

The softsign Bijector has the following two useful properties:

- The domain is all real numbers
- softsign(x) approx sgn(x), for large lxl.

**Usage**

```python
tfb_softsign(validate_args = FALSE, name = "softsign")
```

**Arguments**

- **validate_args** Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name** name prefixed to Ops created by this class.
tfb_split

Split a Tensor event along an axis into a list of Tensors.

Description

The inverse of split concatenates a list of Tensors along axis.

Usage

tfb_split(num_or_size_splits, axis = -1, validate_args = FALSE, name = "split")

Arguments

num_or_size_splits
Either an integer indicating the number of splits along axis or a 1-D integer Tensor or Python list containing the sizes of each output tensor along axis. If a list/Tensor, it may contain at most one value of -1, which indicates a split size that is unknown and determined from input.

axis
A negative integer or scalar int32 Tensor. The dimension along which to split. Must be negative to enable the bijector to support arbitrary batch dimensions. Defaults to -1 (note that this is different from the tf$Split default of 0). Must be statically known.

validate_args
Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name
name prefixed to Ops created by this class.

Value

a bijector instance.


**tfb_square**

Computes \( g(X) = X^2 \); \( X \) is a positive real number.

---

**Description**

\( g \) is a bijection between the non-negative real numbers \((\mathbb{R}_+)\) and the non-negative real numbers.

**Usage**

```
tfb_square(validate_args = FALSE, name = "square")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expml()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_orderered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()` , `tfb_scale_matvec_diag()` , `tfb_scale_matvec_inverse()` , `tfb_scale_matvec_lu()` , `tfb_scale_matvec_tri_l()` , `tfb_scale_tri_l()` , `tfb_shift()` , `tfb_sigmoid()` , `tfb_sinh_arcsinh()` , `tfb_sinh()` , `tfb_softmax_centered()` , `tfb_softplus()` , `tfb_softsign()` , `tfb_square()` , `tfb_tanh()` , `tfb_transform_diagonal()` , `tfb_transpose()` , `tfb_weibull_cdf()` , `tfb_weibull()`
tfb_tanh Computes $Y = \tanh(X)$

**Description**

$Y = \tanh(X)$, therefore $Y$ in $(-1, 1)$.

**Usage**

```r
tfb_tanh(validate_args = FALSE, name = "tanh")
```

**Arguments**

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- `name` name prefixed to Ops created by this class.

**Details**

This can be achieved by an affine transform of the Sigmoid bijector, i.e., it is equivalent to

```r
tfb_chain(list(tfb_affine(shift = -1, scale = 2), tfb_sigmoid(), tfb_affine(scale = 2)))
```

However, using the Tanh bijector directly is slightly faster and more numerically stable.

**Value**

a bijector instance.

**See Also**

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_expm1()`, `tfb_exp()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_identity()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tril()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_recsnail_cdf()`, `tfb_real_nvp_default_template()`, `tfb_real_nvp()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tril()`, `tfb_scale_tril()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_sigm直观()`, `tfb_split()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_transpose()`, `tfb_weibull_cdf()`, `tfb_weibull()`. 
tfb_transform_diagonal

Applies a Bijector to the diagonal of a matrix

Description
Applies a Bijector to the diagonal of a matrix

Usage
```r
tfb_transform_diagonal(
  diag_bijector,
  validate_args = FALSE,
  name = "transform_diagonal"
)
```

Arguments

- **diag_bijector**: Bijector instance used to transform the diagonal.
- **validate_args**: Logical, default FALSE. Whether to validate input with asserts. If validate_args
  is FALSE, and the inputs are invalid, correct behavior is not guaranteed.
- **name**: name prefixed to Ops created by this class.

Value

- a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.  
Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expm()`, `tfb_ffjord()`, `tfb_fill_scale_tril()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`,
<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tfb_kumaraswamy_cdf()</code>, <code>tfb_kumaraswamy()</code></td>
</tr>
<tr>
<td><code>tfb_lambert_w_tail()</code>, <code>tfb_masked_autoregressive_default_template()</code></td>
</tr>
<tr>
<td><code>tfb_masked_autoregressive_flow()</code>, <code>tfb_masked_dense()</code></td>
</tr>
<tr>
<td><code>tfb_matrix_inverse_tri_l()</code></td>
</tr>
<tr>
<td><code>tfb_matvec_lu()</code>, <code>tfb_normal_cdf()</code>, <code>tfb_ordered()</code>, <code>tfb_pad()</code></td>
</tr>
<tr>
<td><code>tfb_permute()</code>, <code>tfb_power_transform()</code></td>
</tr>
<tr>
<td><code>tfb_rational_quadratic_spline()</code></td>
</tr>
<tr>
<td><code>tfb_rayleigh_cdf()</code>, <code>tfb_real_nvp_default_template()</code></td>
</tr>
<tr>
<td><code>tfb_real_nvp()</code>, <code>tfb_reciprocal()</code>, <code>tfb_reshape()</code></td>
</tr>
<tr>
<td><code>tfb_scale_matvec_diag()</code>, <code>tfb_scale_matvec_linear_operator()</code></td>
</tr>
<tr>
<td><code>tfb_scale_matvec_lu()</code>, <code>tfb_scale_matvec_tri_l()</code></td>
</tr>
<tr>
<td><code>tfb_scale_tri_l()</code>, <code>tfb_scale()</code></td>
</tr>
<tr>
<td><code>tfb_shifted_gompertz_cdf()</code></td>
</tr>
<tr>
<td><code>tfb_shift()</code></td>
</tr>
<tr>
<td><code>tfb_sigmoid()</code></td>
</tr>
<tr>
<td><code>tfb_sinh_arcsinh()</code></td>
</tr>
<tr>
<td><code>tfb_sinh()</code></td>
</tr>
<tr>
<td><code>tfb_softmax_centered()</code></td>
</tr>
<tr>
<td><code>tfb_softplus()</code></td>
</tr>
<tr>
<td><code>tfb_softsign()</code></td>
</tr>
<tr>
<td><code>tfb_split()</code></td>
</tr>
<tr>
<td><code>tfb_square()</code></td>
</tr>
<tr>
<td><code>tfb_tanh()</code></td>
</tr>
<tr>
<td><code>tfb_transpose()</code></td>
</tr>
<tr>
<td><code>tfb_weibull_cdf()</code>, <code>tfb_weibull()</code></td>
</tr>
</tbody>
</table>

**tfb_transpose**

Computes $Y = g(X) = transpose_rightmost dims(X, rightmost_perm)$

**Description**

This bijector is semantically similar to `tf.transpose` except that it transposes only the rightmost "event" dimensions. That is, unlike `tf.transpose` the perm argument is itself a permutation of `tf.range(rightmost_transposed_ndims)` rather than `tf.range(tf.rank(x))`, i.e., users specify the (rightmost) dimensions to permute, not all dimensions.

**Usage**

```r
tfb_transpose(
  perm = NULL,
  rightmost_transposed_ndims = NULL,
  validate_args = FALSE,
  name = "transpose"
)
```

**Arguments**

- `perm` Positive integer vector-shaped Tensor representing permutation of rightmost dims (for forward transformation). Note that the 0th index represents the first of the rightmost dims and the largest value must be `rightmost_transposed_ndims` - 1 and corresponds to `tf.rank(x) - 1`. Only one of `perm` and `rightmost_transposed_ndims` can (and must) be specified. Default value: `tf.range(start=rightmost_transposed_ndims, limit=-1, delta=-1)`.

- `rightmost_transposed_ndims` Positive integer scalar-shaped Tensor representing the number of rightmost dimensions to permute. Only one of `perm` and `rightmost_transposed_ndims` can (and must) be specified. Default value: `tf.size(perm)`.

- `validate_args` Logical, default FALSE. Whether to validate input with asserts. If `validate_args` is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

- `name` name prefixed to Ops created by this class.
The actual (forward) transformation is:

```r
sample_batch_ndims <- tf$rank(x) - tf$size(parm) perm = tf$concat(list(tf$range(sample_batch_ndims), sample_batch_ndims + parm), axis=0) tf$transpose(x, perm)
```

Value

a bijector instance.

See Also

For usage examples see `tfb_forward()`, `tfb_inverse()`, `tfb_inverse_log_det_jacobian()`.

Other bijectors: `tfb_absolute_value()`, `tfb_affine_linear_operator()`, `tfb_affine_scalar()`, `tfb_affine()`, `tfb_ascending()`, `tfb_batch_normalization()`, `tfb_blockwise()`, `tfb_chain()`, `tfb_cholesky_outer_product()`, `tfb_cholesky_to_inv_cholesky()`, `tfb_correlation_cholesky()`, `tfb_cumsum()`, `tfb_discrete_cosine_transform()`, `tfb_exp()`, `tfb_expm1()`, `tfb_ffjord()`, `tfb_fill_scale_tri_l()`, `tfb_fill_triangular()`, `tfb_glow()`, `tfb_gompertz_cdf()`, `tfb_gumbel_cdf()`, `tfb_gumbel()`, `tfb_identity()`, `tfb_inline()`, `tfb_invert()`, `tfb_iterated_sigmoid_centered()`, `tfb_kumaraswamy_cdf()`, `tfb_kumaraswamy()`, `tfb_lambert_w_tail()`, `tfb_masked_autoregressive_default_template()`, `tfb_masked_autoregressive_flow()`, `tfb_masked_dense()`, `tfb_matrix_inverse_tri_l()`, `tfb_matvec_lu()`, `tfb_normal_cdf()`, `tfb_ordered()`, `tfb_pad()`, `tfb_permute()`, `tfb_power_transform()`, `tfb_rational_quadratic_spline()`, `tfb_rayleigh_cdf()`, `tfb_real_nvp_default_template()`, `tfb_reciprocal()`, `tfb_reshape()`, `tfb_scale_matvec_diag()`, `tfb_scale_matvec_linear_operator()`, `tfb_scale_matvec_lu()`, `tfb_scale_matvec_tri_l()`, `tfb_scale_tri_l()`, `tfb_scale()`, `tfb_shifted_gompertz_cdf()`, `tfb_shift()`, `tfb_sigmoid()`, `tfb_sinh_arcsinh()`, `tfb_sinh()`, `tfb_softmax_centered()`, `tfb_softplus()`, `tfb_softsign()`, `tfb_split()`, `tfb_square()`, `tfb_tanh()`, `tfb_transform_diagonal()`, `tfb_weibull_cdf()`, `tfb_weibull()``

**tfb_weibull**

Computes $Y = g(X) = 1 - \exp((-X / \text{scale}) \times \text{concentration})$

where $X \geq 0$

Description

This bijector maps inputs from $[0, \infty]$ to $[0, 1]$. The inverse of the bijector applied to a uniform random variable $X \sim U(0, 1)$ gives back a random variable with the Weibull distribution:

Usage

```r
tfb_weibull(
  scale = 1,
  concentration = 1,
  validate_args = FALSE,
  name = "weibull"
)
```
tfb_weibull_cdf

Compute \( Y = g(X) = 1 - \exp((-X / \text{scale}) ** \text{concentration}), X \geq 0. \)

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>Positive Float-type Tensor that is the same dtype and is broadcastable with concentration. This is ( l ) in ( Y = g(X) = 1 - \exp((-x / l) ** k) ).</td>
</tr>
<tr>
<td>concentration</td>
<td>Positive Float-type Tensor that is the same dtype and is broadcastable with scale. This is ( k ) in ( Y = g(X) = 1 - \exp((-x / l) ** k) ).</td>
</tr>
<tr>
<td>validate_args</td>
<td>Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.</td>
</tr>
<tr>
<td>name</td>
<td>name prefixed to Ops created by this class.</td>
</tr>
</tbody>
</table>

**Details**

\( Y \sim \text{Weibull}(\text{scale}, \text{concentration}) \)

**Value**

a bijector instance.

**See Also**

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine.linear_operator(), tfb_affine.scalar(),

tfb_affine(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(),

tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(),

tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_expmi(), tfb_exp(), tfb_ffjord(),

tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(),

tfb_gumbel(), tfb_identity(), tfb_invert(), tfb_iterated_siomial_centered(),

tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(),

tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(),

tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(),

tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(),

tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(),

tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(),

tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),

tfb_softplus(), tfb_softsign(), tfb_split(), tfb_square(), tfb_tanh(), tfb_transform_diagonal(),

tfb_transpose(), tfb_weibull_cdf()
Y ~ Weibull(scale, concentration)

\[
\text{pdf}(y; \text{scale}, \text{concentration}, y \geq 0) = \\
\left(\frac{\text{concentration}}{\text{scale}}\right) \times (\frac{y}{\text{scale}})^{\text{concentration} - 1} \times \\
\exp\left(-\left(\frac{y}{\text{scale}}\right)^{\text{concentration}}\right)
\]

Usage

tfb_weibull_cdf(
    scale = 1,
    concentration = 1,
    validate_args = FALSE,
    name = "weibull_cdf"
)

Arguments

scale : Positive Float-type Tensor that is the same dtype and is broadcastable with concentration. This is \(l\) in \(Y = g(X) = 1 - \exp\left(-\frac{x}{l}\right)^k\).

concentration : Positive Float-type Tensor that is the same dtype and is broadcastable with scale. This is \(k\) in \(Y = g(X) = 1 - \exp\left(-\frac{x}{l}\right)^k\).

validate_args : Logical, default FALSE. Whether to validate input with asserts. If validate_args is FALSE, and the inputs are invalid, correct behavior is not guaranteed.

name : name prefixed to Ops created by this class.

Details

Likewise, the forward of this bijector is the Weibull distribution CDF.

Value

a bijector instance.

See Also

For usage examples see tfb_forward(), tfb_inverse(), tfb_inverse_log_det_jacobian().

Other bijectors: tfb_absolute_value(), tfb_affine_linear_operator(), tfb_affine_scalar(), tfb_ascending(), tfb_batch_normalization(), tfb_blockwise(), tfb_chain(), tfb_cholesky_outer_product(), tfb_cholesky_to_inv_cholesky(), tfb_correlation_cholesky(), tfb_cumsum(), tfb_discrete_cosine_transform(), tfb_exp(), tfb_expm1(), tfb_ffjord(), tfb_fill_scale_tri_l(), tfb_fill_triangular(), tfb_glow(), tfb_gompertz_cdf(), tfb_gumbel_cdf(), tfb_gumbel(), tfb_identity(), tfb_independent(), tfb_invert(), tfb_iterated_sigmoid_centered(), tfb_kumaraswamy_cdf(), tfb_kumaraswamy(), tfb_lambert_w_tail(), tfb_masked_autoregressive_default_template(), tfb_masked_autoregressive_flow(), tfb_masked_dense(), tfb_matrix_inverse_tri_l(), tfb_matvec_lu(), tfb_normal_cdf(), tfb_ordered(), tfb_pad(), tfb_permute(), tfb_power_transform(), tfb_rational_quadratic_spline(), tfb_rayleigh_cdf(), tfb_real_nvp_default_template(), tfb_real_nvp(), tfb_reciprocal(), tfb_reshape(), tfb_scale_matvec_diag(), tfb_scale_matvec_linear_operator(), tfb_scale_matvec_lu(), tfb_scale_matvec_tri_l(), tfb_scale_tri_l(), tfb_scale(), tfb_shifted_gompertz_cdf(), tfb_shift(), tfb_sigmoid(), tfb_sinh_arcsinh(), tfb_sinh(), tfb_softmax_centered(),...
**tfd_autoregressive**

**Autoregressive distribution**

### Description

The Autoregressive distribution enables learning (often) richer multivariate distributions by repeatedly applying a diffeomorphic transformation (such as implemented by Bijectors).

### Usage

```python
tfd_autoregressive(
    distribution_fn,
    sample0 = NULL,
    num_steps = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Autoregressive"
)
```

### Arguments

- **distribution_fn**
  Function which constructs a `tfd$Distribution`-like instance from a Tensor (e.g., `sample0`). The function must respect the "autoregressive property", i.e., there exists a permutation of event such that each coordinate is a diffeomorphic function of on preceding coordinates.

- **sample0**
  Initial input to `distribution_fn`; used to build the distribution in `__init__` which in turn specifies this distribution's properties, e.g., event_shape, batch_shape, dtype. If unspecified, then `distribution_fn` should be default constructable.

- **num_steps**
  Number of times `distribution_fn` is composed from samples, e.g., `num_steps=2` implies `distribution_fn(distribution_fn(sample0)$sample(n))$sample()`.

- **validate_args**
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

- **name**
  name prefixed to Ops created by this class.
Regarding terminology, "Autoregressive models decompose the joint density as a product of conditions, and model each conditional in turn. Normalizing flows transform a base density (e.g. a standard Gaussian) into the target density by an invertible transformation with tractable Jacobian." (Papamakarios et al., 2016)

In other words, the "autoregressive property" is equivalent to the decomposition, \( p(x) = \prod \{ p(x[i] | x[0:i]) : i=0, \ldots, d \} \). The provided `shift_and_log_scale_fn`, `tfb_masked_autoregressive_default_template`, achieves this property by zeroing out weights in its `masked_dense` layers. Practically speaking the autoregressive property means that there exists a permutation of the event coordinates such that each coordinate is a diffeomorphic function of only preceding coordinates (van den Oord et al., 2016).

Mathematical Details

The probability function is

\[ \text{prob}(x; \text{fn}, n) = \text{fn}(x).\text{prob}(x) \]

And a sample is generated by

\[ x = \text{fn}(\ldots\text{fn}(\text{fn}(x_0).\text{sample()}).\text{sample()}).\text{sample()} \]

where the ellipses (\ldots) represent \( n-2 \) composed calls to \( \text{fn} \), \( \text{fn} \) constructs a `tfd$Distribution`-like instance, and \( x_0 \) is a fixed initializing Tensor.

Value

A distribution instance.

References


See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisquare()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_gamma()`, `tfd_exp_inv_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma()`.

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisquare()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inv_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma()`.
tfd_batch_reshape

Batch-Reshaping distribution

Description

This "meta-distribution" reshapes the batch dimensions of another distribution.

Usage

tfd_batch_reshape(
  distribution,
  batch_shape,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = NULL
)

Arguments

distribution The base distribution instance to reshape. Typically an instance of Distribution.
batch_shape Positive integer-like vector-shaped Tensor representing the new shape of the batch dimensions. Up to one dimension may contain -1, meaning the remainder of the batch size.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.
The Bates distribution is the distribution of the average of \( \text{total_count} \) independent samples from \( \text{Uniform}(\text{low}, \text{high}) \). It is parameterized by the interval bounds \( \text{low} \) and \( \text{high} \), and \( \text{total_count} \), the number of samples. Although some care has been taken to avoid numerical issues, the pdf, cdf, and log versions thereof may still exhibit numerical instability. They are relatively stable near the tails; however near the mode they are unstable if \( \text{total_count} \) is greater than about 75 for \( \text{tf}\_\text{float64} \), 25 for \( \text{tf}\_\text{float32} \), and 7 for \( \text{tf}\_\text{float16} \). Beyond these limits a warning will be shown if \( \text{validate_args}=\text{FALSE} \); otherwise an exception is thrown. For high \( \text{total_count} \), consider using a Normal approximation.
tfd_bates

Usage
tfd_bates(
    total_count,
    low = 0,
    high = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Bates"
)

Arguments

  total_count  Non-negative integer-valued Tensor with shape broadcastable to the batch shape 
               [N1, ..., Nm], m >= 0. This controls the number of samples of Uniform(low, high) 
               to take the mean of.

  low          Floating point Tensor representing the lower bounds of the support. Should be 
               broadcastable to [N1, ..., Nm] with m >= 0, the same dtype as total_count, and 
               low < high component-wise, after broadcasting. Defaults to 0.

  high        Floating point Tensor representing the upper bounds of the support. Should be 
              broadcastable to [N1, ..., Nm] with m >= 0, the same dtype as total_count, and 
              low < high component-wise, after broadcasting. Defaults to 1.

  validate_args Logical, default FALSE. When TRUE distribution parameters are checked for 
                   validity despite possibly degrading runtime performance. When FALSE invalid 
                   inputs may silently render incorrect outputs. Default value: FALSE.

  allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use 
                     the value NaN to indicate the result is undefined. When FALSE, an exception is 
                     raised if one or more of the statistic’s batch members are undefined.

  name         name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is supported in the interval [low, high]. If [low, high] is the 
unit interval [0, 1], the pdf is,

\[
\text{pdf}(x; n, 0, 1) = ((n / (n-1)!) \sum_{k=0}^{j} (-1)^k \binom{n}{k} (nx - k)^{n-1})
\]

where
- total_count = n,
- j = floor(nx)
- n! is the factorial of n,
- \( \binom{n}{k} \) is the binomial coefficient \( n! / (k!(n - k)) \)

For arbitrary intervals [low, high], the pdf is,

\[
\text{pdf}(x; n, \text{low}, \text{high}) = \text{pdf}((x - \text{low}) / (\text{high} - \text{low}); n, 0, 1) / (\text{high} - \text{low})
\]
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


`tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture()`, `tfd_mixture()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t()`.

`tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises()`.

`tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart_tri_l()`, `tfd_wishart()`, `tfd_zipf()`.

---

**tfd_bernoulli**

---

**Bernoulli distribution**

**Description**

The Bernoulli distribution with `probs` parameter, i.e., the probability of a 1 outcome (vs a 0 outcome).

**Usage**

```r
tfd_bernoulli(
  logits = NULL,
  probs = NULL,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Bernoulli"
)
```
Arguments

logits An N-D Tensor representing the log-odds of a 1 event. Each entry in the Tensor parametrizes an independent Bernoulli distribution where the probability of an event is sigmoid(logits). Only one of logits or probs should be passed in.

probs An N-D Tensor representing the probability of a 1 event. Each entry in the Tensor parameterizes an independent Bernoulli distribution. Only one of logits or probs should be passed in.

dtype The type of the event samples. Default: int32.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixure_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(cos), tfd_multivariate_normal_full_c covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_logistic(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinkhorn_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
**Beta distribution**

**Description**

The Beta distribution is defined over the (0, 1) interval using parameters concentration1 (aka "alpha") and concentration0 (aka "beta").

**Usage**

```r
tfd_beta(  
  concentration1 = NULL,  
  concentration0 = NULL,  
  validate_args = FALSE,  
  allow_nan_stats = TRUE,  
  name = "Beta"  
)
```

**Arguments**

- `concentration1` Positive floating-point Tensor indicating mean number of successes; aka "alpha". Implies self$dtype and self$batch_shape, i.e., concentration1$shape = [N1, N2, ..., Nm] = self$batch_shape.
- `concentration0` Positive floating-point Tensor indicating mean number of failures; aka "beta". Otherwise has same semantics as concentration1.
- `validate_args` Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The probability density function (pdf) is,

\[pdf(x; \alpha, \beta) = x^{(\alpha - 1)} (1 - x)^{(\beta - 1)} / \Gamma(\alpha) \Gamma(\beta) / \Gamma(\alpha + \beta)\]

where:

- concentration1 = \(\alpha\),
- concentration0 = \(\beta\),
- \(\Gamma\) is the normalization constant, and,
• Gamma is the gamma function. The concentration parameters represent mean total counts of a 1 or a 0, i.e.,

\[
\text{concentration}_1 = \alpha = \text{mean} \times \text{total_concentration} \\
\text{concentration}_0 = \beta = (1. - \text{mean}) \times \text{total_concentration}
\]

where mean in (0, 1) and total_concentration is a positive real number representing a mean total_count = concentration1 + concentration0. Distribution parameters are automatically broadcast in all functions; see examples for details. Warning: The samples can be zero due to finite precision. This happens more often when some of the concentrations are very small. Make sure to round the samples to np$\text{finfo}(\text{dtype})$tiny before computing the density. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Beta-Binomial compound distribution

Description

The Beta-Binomial distribution is parameterized by (a batch of) total_count parameters, the number of trials per draw from Binomial distributions where the probabilities of success per trial are drawn from underlying Beta distributions; the Beta distributions are parameterized by concentration1 (aka 'alpha') and concentration0 (aka 'beta'). Mathematically, it is (equivalent to) a special case of the Dirichlet-Multinomial over two classes, although the computational representation is slightly different: while the Beta-Binomial is a distribution over the number of successes in total_count trials, the two-class Dirichlet-Multinomial is a distribution over the number of successes and failures.

Usage

```python
tfd_beta_binomial(
    total_count,
    concentration1,
    concentration0,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "BetaBinomial"
)
```

Arguments

total_count  Non-negative integer-valued tensor, whose dtype is the same as concentration1 and concentration0. The shape is broadcastable to [N1,...,Nm] with m >= 0. When total_count is broadcast with concentration1 and concentration0, it defines the distribution as a batch of N1 x ... x Nm different Beta-Binomial distributions. Its components should be equal to integer values.

concentration1  Positive floating-point Tensor indicating mean number of successes. Specifically, the expected number of successes is total_count * concentration1 / (concentration1 + concentration0).

concentration0  Positive floating-point Tensor indicating mean number of failures; see description of concentration1 for details.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

ame  name prefixed to Ops created by this class.
Details

Mathematical Details

The Beta-Binomial is a distribution over the number of successes in total_count independent Binomial trials, with each trial having the same probability of success, the underlying probability being unknown but drawn from a Beta distribution with known parameters. The probability mass function (pmf) is,

$$\text{pmf}(k; n, a, b) = \frac{\text{Beta}(k + a, n - k + b)}{Z}$$

$$Z = \frac{(k! \ (n - k)! \ / \ n!)}{\text{Beta}(a, b)}$$

where:
- concentration1 = a > 0,
- concentration0 = b > 0,
- total_count = n, n a positive integer,
- n! is n factorial,
- Beta(x, y) = Gamma(x) Gamma(y) / Gamma(x + y) is the beta function, and
- Gamma is the gamma function.

Dirichlet-Multinomial is a compound distribution, i.e., its samples are generated as follows.

1. Choose success probabilities: probs ~ Beta(concentration1, concentration0)
2. Draw integers representing the number of successes: counts ~ Binomial(total_count, probs)

Distribution parameters are automatically broadcast in all functions; see examples for details.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space()`, `tfd_lkj()`, `tfd_log_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature`. 
tfd_binomial

| tfd_binomial | Binomial distribution |

Description

This distribution is parameterized by \( \text{probs} \), a (batch of) probabilities for drawing a 1 and \( \text{total_count} \), the number of trials per draw from the Binomial.

Usage

```r
tfd_binomial(
  total_count,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Beta"
)
```

Arguments

- **total_count** - Non-negative floating point tensor with shape broadcastable to \([N_1, ..., N_m]\) with \( m \geq 0 \) and the same dtype as \( \text{probs} \) or \( \text{logits} \). Defines this as a batch of \( N_1 \times ... \times N_m \) different Binomial distributions. Its components should be equal to integer values.

- **logits** - Floating point tensor representing the log-odds of a positive event with shape broadcastable to \([N_1, ..., N_m]\) \( m \geq 0 \), and the same dtype as \( \text{total_count} \). Each entry represents logits for the probability of success for independent Binomial distributions. Only one of \( \text{logits} \) or \( \text{probs} \) should be passed in.

- **probs** - Positive floating point tensor with shape broadcastable to \([N_1, ..., N_m]\) \( m \geq 0 \), \( \text{probs} \) in \([0, 1]\). Each entry represents the probability of success for independent Binomial distributions. Only one of \( \text{logits} \) or \( \text{probs} \) should be passed in.

- **validate_args** - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name

name prefixed to Ops created by this class.

Details

Mathematical Details

The Binomial is a distribution over the number of 1’s in total_count independent trials, with each trial having the same probability of 1, i.e., probs.

The probability mass function (pmf) is,

\[pmf(k; n, p) = p^k (1 - p)^{(n - k)} / Z\]

\[Z = k! (n - k)! / n!\]

where:

- total_count = n,
- probs = p,
- Z is the normalizing constant, and,
- n! is the factorial of n.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),

tfd_beta_binomial(), tfd_beta(), tfd_categorical(), tfd_cauchy(), tfd_chisquare(), tfd_chi,
tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(),
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
tfd_generalized_normal(), tfd_geometric(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space,
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal,
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature,
tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(),
tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distri
Blockwise distribution

Usage

```r
tfd_blockwise(
  distributions,
  dtype_override = NULL,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "Blockwise"
)
```

Arguments

- **distributions**: list of Distribution instances. All distribution instances must have the same batch_shape and all must have `event_ndims==1`, i.e., be vector-variate distributions.
- **dtype_override**: samples of distributions will be cast to this dtype. If unspecified, all distributions must have the same dtype. Default value: `NULL` (i.e., do not cast).
- **validate_args**: Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.
- **allow_nan_stats**: Logical, default `TRUE`. When `TRUE`, statistics (e.g., mean, mode, variance) use the value `NaN` to indicate the result is undefined. When `FALSE`, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.
Description

The Categorical distribution is parameterized by either probabilities or log-probabilities of a set of K classes. It is defined over the integers \{0, 1, ..., K-1\}.

Usage

```r
tfd_categorical(
  logits = NULL,
  probs = NULL,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Categorical"
)
```

Arguments

- **logits**: An N-D Tensor, N >= 1, representing the log probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.
- **probs**: An N-D Tensor, N >= 1, representing the probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.
- **dtype**: The type of the event samples (default: int32).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

The Categorical distribution is closely related to the OneHotCategorical and Multinomial distributions. The Categorical distribution can be intuited as generating samples according to argmax\{ OneHotCategorical(probs) \} itself being identical to argmax\{ Multinomial(probs, total_count=1) \}.

Mathematical Details

The probability mass function (pmf) is,
tfd_cauchy

Cauchy distribution with location \( \text{loc} \) and scale \( \text{scale} \)

### Description

Mathematical details

\[
\text{pmf}(k; \text{pi}) = \prod_j \text{pi}_j^{[k == j]}
\]

**Pitfalls**

The number of classes, \( K \), must not exceed:

- the largest integer representable by \( \text{self.dtype} \), i.e., \( 2^{(*)} \text{mantissa_bits} + 1 \) (IEEE 754),
- the maximum Tensor index, i.e., \( 2^{*31} - 1 \).

Note: This condition is validated only when \( \text{validate_args = TRUE} \).

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. \( \text{tfd_sample()}, \text{tfd_log_prob()}, \text{tfd_mean}() \).

Other distributions: \( \text{tfd_autoregressive()}, \text{tfd_batch_reshape()}, \text{tfd_bates()}, \text{tfd_bernoulli}(), \text{tfd_beta_binary()}, \text{tfd_beta()}, \text{tfd_binomial}(), \text{tfd_cauchy}(), \text{tfd_chi2}(), \text{tfd_chi()}, \text{tfd_cholesky_lkj()}, \text{tfd_continuous_bernoulli}(), \text{tfd_deterministic}(), \text{tfd_dirichlet_multinomial}(), \text{tfd_dirichlet}(), \text{tfd_empirical}(), \text{tfd_exp_gamma}(), \text{tfd_exp_inverse_gamma}(), \text{tfd_exponential}(), \text{tfd_gamma}(\text{gamma})(), \text{tfd_gaussian_process_regression_model}(), \text{tfd_gaussian_process}(), \text{tfd_generalized_normal}(), \text{tfd_geometric}(), \text{tfd_gumbel}(), \text{tfd_half_cauchy}(), \text{tfd_half_normal}(), \text{tfd_hidden_markov_model}(), \text{tfd_horseshoe}(), \text{tfd_independent}(), \text{tfd_inverse_gaussian}(), \text{tfd_inverse_gaussian}(), \text{tfd_johnson_s_u}(), \text{tfd_joint_distribution_named_auto_batched}(), \text{tfd_joint_distribution_named()}(), \text{tfd_joint_distribution_sequential_auto_batched}(), \text{tfd_joint_distribution_sequential()}(), \text{tfd_kumaraswamy}(), \text{tfd_laplace}(), \text{tfd_linear_gaussian_state_space}(), \text{tfd_lkj}(), \text{tfd_logistic}(), \text{tfd_lgamma()}(), \text{tfd_mix_same_family}(), \text{tfd_multinomial}(), \text{tfd_multivariate_normal_diag_plus_low_rank}(), \text{tfd_multivariate_normal_full_covariance}(), \text{tfd_multivariate_normal_linear_operator}(), \text{tfd_multivariate_normal_tri_l}(), \text{tfd_multivariate_student_t_linear_operator}(), \text{tfd_negative_binomial}(), \text{tfd_normal}(), \text{tfd_one_hot_categorical}(), \text{tfd_pareto}(), \text{tfd_pixel_cnn}(), \text{tfd_poisson_log_normal_quadrature}(), \text{tfd_poisson}(), \text{tfd_power_spherical}(), \text{tfd_probit_bernoulli}(), \text{tfd_quantized}(), \text{tfd_relaxed_bernoulli}(), \text{tfd_relexed_one_hot_categorical}(), \text{tfd_sample_distribution}(), \text{tfd_sinh_arcsinh}(), \text{tfd_skellam}(), \text{tfd_spherical_uniform}(), \text{tfd_student_t_process}(), \text{tfd_student_t}(), \text{tfd_transformed_distri}(), \text{tfd_triangular}(), \text{tfd_truncated_cauchy}(), \text{tfd_truncated_normal}(), \text{tfd_uniform}(), \text{tfd_von_mises}(), \text{tfd_von_mises_fisher}(), \text{tfd_von_mises}(), \text{tfd_weibull}(), \text{tfd_wishart_linear_operator}(), \text{tfd_wishart_tri_l}(), \text{tfd_wishart}(), \text{tfd_zipf}()
tfd_cauchy

Usage

tfd_cauchy(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Cauchy"
)

Arguments

- **loc**: Floating point tensor; the modes of the distribution(s).
- **scale**: Floating point tensor; the locations of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

The probability density function (pdf) is,

\[
pdf(x; \text{loc}, \text{scale}) = \frac{1}{\pi \text{scale} (1 + z^2)}
\]

\[
z = (x - \text{loc}) / \text{scale}
\]

where loc is the location, and scale is the scale. The Cauchy distribution is a member of the location-scale family, i.e. \(Y \sim \text{Cauchy}(\text{loc}, \text{scale})\) is equivalent to,

\[
X \sim \text{Cauchy}(\text{loc}=0, \text{scale}=1)
\]

\[
Y = \text{loc} + \text{scale} \times X
\]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`. 
Cumulative distribution function. Given random variable $X$, the cumulative distribution function $cdf$ is: $cdf(x) := P[X \leq x]$

### Description

Cumulative distribution function. Given random variable $X$, the cumulative distribution function $cdf$ is: $cdf(x) := P[X \leq x]$

### Usage

```python
tfd_cdf(distribution, value, ...)
```

### Arguments

- **distribution**: The distribution being used.
- **value**: float or double Tensor.
- **...**: Additional parameters passed to Python.

### Value

$a$ Tensor of shape `sample_shape(x) + self$batch_shape` with values of type self$dtype.

### See Also

Other distribution_methods: `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`
Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_cdf(x)

Chi distribution

Description

The Chi distribution is defined over nonnegative real numbers and uses a degrees of freedom ("df") parameter.

Usage

tfd_chi(df, validate_args = FALSE, allow_nan_stats = TRUE, name = "Chi")

Arguments

df Floating point tensor, the degrees of freedom of the distribution(s). df must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; df, x \geq 0) = x^{df-1} \exp\left(-0.5 x^2\right) / Z
\]
\[
Z = 2^{(0.5 df - 1)} \Gamma(0.5 df)
\]

where:

- df denotes the degrees of freedom,
- Z is the normalization constant, and,
- \(\Gamma\) is the gamma function.

The Chi distribution is a transformation of the Chi2 distribution; it is the distribution of the positive square root of a variable obeying a Chi distribution.
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
`tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`,
`tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`,
`tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
`tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`,
`tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`,
`tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`,
`tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_regression()`,
`tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`,
`tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`,
`tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`,
`tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`,
`tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`,
`tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_iwi()`,
`tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_student_t()`,
`tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`,
`tfd_wishart_tri_l()`, `tfd_wishart()`, `tfd_zipf()`

---

Chisquare distribution

Description

The Chi2 distribution is defined over positive real numbers using a degrees of freedom ("df") parameter.

Usage

`tfd_chi2(df, validate_args = FALSE, allow_nan_stats = TRUE, name = "Chi2")`

Arguments

df

Floating point tensor, the degrees of freedom of the distribution(s). df must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[ pdf(x; df, x > 0) = x^{(0.5 \ df - 1)} \exp(-0.5 x) / Z \]
\[ Z = 2^{(0.5 \ df)} \Gamma(0.5 \ df) \]

where

- \( df \) denotes the degrees of freedom,
- \( Z \) is the normalization constant, and,
- \( \Gamma \) is the gamma function. The Chi2 distribution is a special case of the Gamma distribution, i.e.,

\[ \text{Chi2}(df) = \Gamma(\text{concentration}=0.5 * df, \text{rate}=0.5) \]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi()`.
The CholeskyLKJ distribution on cholesky factors of correlation matrices

This is a one-parameter family of distributions on cholesky factors of correlation matrices. In other words, if \( X \sim \text{CholeskyLKJ}(c) \), then \( X @ X^T \sim \text{LKJ}(c) \). For more details on the LKJ distribution, see `tfd_lkj`.

Usage

```r
tfd_cholesky_lkj(
  dimension,
  concentration,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "CholeskyLKJ"
)
```

Arguments

dimension
  integer. The dimension of the correlation matrices to sample.

concentration
  float or double Tensor. The positive concentration parameter of the CholeskyLKJ distributions.

validate_args
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name
  name prefixed to Ops created by this class.

Value

a distribution instance.
See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

```r
tfd_continuous_bernoulli
Continuous Bernoulli distribution.

Description
This distribution is parameterized by `probs`, a (batch of) parameters taking values in (0, 1). Note that, unlike in the Bernoulli case, `probs` does not correspond to a probability, but the same name is used due to the similarity with the Bernoulli.

Usage
```r
tfd_continuous_bernoulli(
  logits = NULL,
  probs = NULL,
  lims = c(0.499, 0.501),
  dtype = tf$float32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "ContinuousBernoulli"
)```
Arguments

logits  An N-D Tensor. Each entry in the Tensor parameterizes an independent continuous Bernoulli distribution with parameter sigmoid(logits). Only one of logits or probs should be passed in. Note that this does not correspond to the log-odds as in the Bernoulli case.

probs  An N-D Tensor representing the parameter of a continuous Bernoulli. Each entry in the Tensor parameterizes an independent continuous Bernoulli distribution. Only one of logits or probs should be passed in. Note that this also does not correspond to a probability as in the Bernoulli case.

lims  A list with two floats containing the lower and upper limits used to approximate the continuous Bernoulli around 0.5 for numerical stability purposes.

dtype  The type of the event samples. Default: float32.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details

Mathematical Details

The continuous Bernoulli is a distribution over the interval [0, 1], parameterized by probs in (0, 1). The probability density function (pdf) is,

$$pdf(x; \text{probs}) = \text{probs}^x \cdot (1 - \text{probs})^{1 - x} \cdot C(\text{probs})$$

$$C(\text{probs}) = \begin{cases} 
(2 \cdot \text{atanh}(1 - 2 \cdot \text{probs}) / (1 - 2 \cdot \text{probs}) & \text{if } \text{probs} \neq 0.5 \\
2 & \text{else}
\end{cases}$$

While the normalizing constant C(probs) is a continuous function of probs (even at probs = 0.5), computing it at values close to 0.5 can result in numerical instabilities due to 0/0 errors. A Taylor approximation of C(probs) is thus used for values of probs in a small interval [lims[0], lims[1]] around 0.5. For more details, see Loaiza-Ganem and Cunningham (2019). NOTE: Unlike the Bernoulli, numerical instabilities can happen for probs very close to 0 or 1. Current implementation allows any value in (0, 1), but this could be changed to (1e-6, 1-1e-6) to avoid these issues.

Value

a distribution instance.

References

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
`tfd_cholesky_lkj()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`,
`tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`,
`tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`,
`tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`,
`tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`,
`tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`,
`tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`,
`tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`,
`tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`,
`tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`,
`tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`,
`tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`,
`tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises()`,
`tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_sinh_arcsinh_diag()`,
`tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`,
`tfd_wishart_tril()`, `tfd_wishart()`, `tfd_zipf()`
Alternatively, for non-vector, multivariate distributions (e.g., matrix-valued, Wishart), Covariance shall return a (batch of) matrices under some vectorization of the events, i.e., Cov[i, j] = Covariance(Vec(X)_i, Vec(X)_j) = [as above] where Cov is a (batch of) k x k matrices, 0 <= (i, j) < k = reduce_prod(event_shape), and Vec is some function mapping indices of this distribution’s event dimensions to indices of a length-k vector.

Value

Floating-point Tensor with shape [B1, ..., Bn, k, k] where the first n dimensions are batch coordinates and k = reduce_prod(self.event_shape).

See Also

Other distribution methods: tfd_cdf(), tfd_cross_entropy(), tfd_entropy(), tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_variance()
**tfd_deterministic**  

**Value**

cross_entropy: self.dtype Tensor with shape [B1, ..., Bn] representing n different calculations of (Shannon) cross entropy.

**See Also**

Other distribution_methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

**Examples**

```r
  d1 <- tfd_normal(loc = 1, scale = 1)
  d2 <- tfd_normal(loc = 2, scale = 1)
  d1 %>% tfd_cross_entropy(d2)
```

---

**tfd_deterministic**

*Scalar Deterministic distribution on the real line*

**Description**

The scalar Deterministic distribution is parameterized by a (batch) point `loc` on the real line. The distribution is supported at this point only, and corresponds to a random variable that is constant, equal to `loc`. See `Degenerate rv`.

**Usage**

```r
  tfd_deterministic(
    loc,
    atol = NULL,
    rtol = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Deterministic"
  )
```

**Arguments**

- `loc`
  - Numeric Tensor of shape [B1, ..., Bb], with b >= 0. The point (or batch of points) on which this distribution is supported.

- `atol`
  - Non-negative Tensor of same dtype as `loc` and broadcastable shape. The absolute tolerance for comparing closeness to `loc`. Default is 0.

- `rtol`
  - Non-negative Tensor of same dtype as `loc` and broadcastable shape. The relative tolerance for comparing closeness to `loc`. Default is 0.
validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability mass function (pmf) and cumulative distribution function (cdf) are

\[ \text{pmf}(x; \text{loc}) = 1, \text{if } x == \text{loc}, \text{else } 0 \]
\[ \text{cdf}(x; \text{loc}) = 1, \text{if } x \geq \text{loc}, \text{else } 0 \]

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

**Description**

The Dirichlet distribution is defined over the \((k-1)\)-simplex using a positive, length-\(k\) vector concentration \((k > 1)\). The Dirichlet is identically the Beta distribution when \(k = 2\).

**Usage**

```python
tfd_dirichlet(
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Dirichlet"
)
```

**Arguments**

- **concentration**: Positive floating-point Tensor indicating mean number of class occurrences; aka "alpha". Implies `self$dtype`, and `self$batch_shape`, `self$event_shape`, i.e., if `concentration$shape = [N1, N2, ..., Nm, k]` then `batch_shape = [N1, N2, ..., Nm]` and `event_shape = [k].`
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

**Details**

Mathematical Details

The Dirichlet is a distribution over the open \((k-1)\)-simplex, i.e.,

\[ S^{k-1} = \{ (x_0, \ldots, x_{k-1}) \text{ in } R^k : \text{sum}_j x_j = 1 \text{ and all}_j x_j > 0 \} \].

The probability density function (pdf) is,

\[
\text{pdf}(x; \alpha) = \frac{\prod_j x_j^{\alpha_j - 1}}{Z} \quad Z = \frac{\prod_j \Gamma(\alpha_j)}{\Gamma(\text{sum}_j \alpha_j)}
\]

where:

- \(x\) in \(S^{k-1}\), i.e., the \((k-1)\)-simplex,
• concentration = alpha = [alpha_0, ..., alpha_(k-1)], alpha_j > 0,
• Z is the normalization constant aka the multivariate beta function, and,
• Gamma is the gamma function.

The concentration represents mean total counts of class occurrence, i.e.,

concentration = alpha = mean * total_concentration

where mean in S^{k-1} and total_concentration is a positive real number representing a mean total count. Distribution parameters are automatically broadcast in all functions; see examples for details. Warning: Some components of the samples can be zero due to finite precision. This happens more often when some of the concentrations are very small. Make sure to round the samples to npfinfo(dtype)$tiny before computing the density. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mix distribution_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_mixture_same_family(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distributions(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
**Description**

The Dirichlet-Multinomial distribution is parameterized by a (batch of) length-K concentration vectors \( (K > 1) \) and a total_count number of trials, i.e., the number of trials per draw from the DirichletMultinomial. It is defined over a (batch of) length-K vector counts such that \( \text{tf$reduce\_sum(counts,-1)} = \text{total\_count} \). The Dirichlet-Multinomial is identically the Beta-Binomial distribution when \( K = 2 \).

**Usage**

```r
  tfd_dirichlet_multinomial(
    total_count,
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "DirichletMultinomial"
  )
```

**Arguments**

- **total_count** Non-negative floating point tensor, whose dtype is the same as concentration. The shape is broadcastable to \([N1, ..., Nm]\) with \( m >= 0 \). Defines this as a batch of \( N1 \times ... \times Nm \) different Dirichlet multinomial distributions. Its components should be equal to integer values.

- **concentration** Positive floating point tensor, whose dtype is the same as \( n \) with shape broadcastable to \([N1, ..., Nm, K]\) \( m >= 0 \). Defines this as a batch of \( N1 \times ... \times Nm \) different \( K \) class Dirichlet multinomial distributions.

- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats** Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name** name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The Dirichlet-Multinomial is a distribution over \( K \)-class counts, i.e., a length-\( K \) vector of non-negative integer counts \( n = [n_0, ..., n_{(K-1)}] \).

The probability mass function (pmf) is,
pmf(n; alpha, N) = Beta(alpha + n) / (prod_j n_j!) / Z
Z = Beta(alpha) / N!

where:
• concentration = alpha = [alpha_0, ..., alpha_{K-1}], alpha_j > 0,
• total_count = N, N a positive integer,
• N! is N factorial, and,
• Beta(x) = prod_j Gamma(x_j) / Gamma(sum_j x_j) is the multivariate beta function, and,
• Gamma is the gamma function.

Dirichlet-Multinomial is a compound distribution, i.e., its samples are generated as follows.
1. Choose class probabilities: probs = [p_0,...,p_{K-1}] ~ Dir(concentration)
2. Draw integers: counts = [n_0,...,n_{K-1}] ~ Multinomial(total_count, probs)

The last concentration dimension parametrizes a single Dirichlet-Multinomial distribution. When calling distribution functions (e.g., dist$prob(counts)), concentration, total_count and counts are broadcast to the same shape. The last dimension of counts corresponds single Dirichlet-Multinomial distributions. Distribution parameters are automatically broadcast in all functions; see examples for details.

Pitfalls The number of classes, K, must not exceed:
• the largest integer representable by self$dtype, i.e., 2**(mantissa_bits+1) (IEE754),
• the maximum Tensor index, i.e., 2**31-1.

Note: This condition is validated only when validate_args = TRUE.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(),
tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(),
tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(),
tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(),
tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(),
tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequence,
tfd_joint_distribution_named_auto_batched(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space,
tfd_lkj(), tfd_log_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixtures(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal,
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature,
Double-sided Maxwell distribution.

Description

This distribution is useful to compute measure valued derivatives for Gaussian distributions. See Mohamed et al. (2019) for more details.

Usage

tfd_doublesided_maxwell(loc, scale, validate_args = FALSE, allow_nan_stats = TRUE, name = "doublesided_maxwell")

Arguments

loc Floating point tensor; location of the distribution
scale Floating point tensor; the scales of the distribution. Must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.
name string prefixed to Ops created by this class. Default value: 'doublesided_maxwell'.
Details

Mathematical details

The double-sided Maxwell distribution generalizes the Maxwell distribution to the entire real line.

\[
\text{pdf}(x; \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} * ((x-\mu)/\sigma)^2 * \exp(-0.5 ((x-\mu)/\sigma)^2)
\]

where \( \text{loc} = \mu \) and \( \text{scale} = \sigma \). The DoublesidedMaxwell distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{DoublesidedMaxwell(\text{loc}=0, \text{scale}=1)}
\]

\[
Y = \text{loc} + \text{scale} * X
\]

The double-sided Maxwell is a symmetric distribution that extends the one-sided maxwell from \( \mathbb{R}^+ \) to the entire real line. Their densities are therefore the same up to a factor of 0.5.

It has several methods for generating random variates from it. The version here uses 3 Gaussian variates and a uniform variate to generate the samples The sampling path is:

\[
\mu + \sigma \times \text{sgn(U-0.5)} \times \sqrt{X^2 + Y^2 + Z^2} \quad \text{U-Unif; } X,Y,Z \sim \mathcal{N}(0,1)
\]

In the sampling process above, the random variates generated by \( \sqrt{X^2 + Y^2 + Z^2} \) are samples from the one-sided Maxwell (or Maxwell-Boltzmann) distribution.

Value

a distribution instance.

References

- Mohamed, et all, "Monte Carlo Gradient Estimation in Machine Learning.", 2019

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

---

tfd_empirical  

Empirical distribution

Description

The Empirical distribution is parameterized by a (batch) multiset of samples. It describes the empirical measure (observations) of a variable. Note: some methods (log_prob, prob, cdf, mode, entropy) are not differentiable with regard to samples.
tfd_empirical

Usage
tfd_empirical(
samples,
event_ndims = 0,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "Empirical"
)

Arguments
  samples  Numeric Tensor of shape [B1, ..., Bk, S, E1, ..., En], k, n >= 0. Samples or
  batches of samples on which the distribution is based. The first k dimensions
  index into a batch of independent distributions. Length of S dimension determines
  number of samples in each multiset. The last n dimension represents samples
  for each distribution. n is specified by argument event_ndims.
  event_ndims int32, default 0. number of dimensions for each event. When 0 this distribution
  has scalar samples. When 1 this distribution has vector-like samples.
  validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
  validity despite possibly degrading runtime performance. When FALSE invalid
  inputs may silently render incorrect outputs. Default value: FALSE.
  allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
  the value NaN to indicate the result is undefined. When FALSE, an exception is
  raised if one or more of the statistic’s batch members are undefined.
  name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability mass function (pmf) and cumulative distribution function (cdf) are

\[
\text{pmf}(k; s_1, \ldots, s_n) = \sum_i \mathbb{I}(k == s_i) / n \\
\mathbb{I}(k == s_i) = 1, \text{ if } k == s_i, \text{ else } 0.
\]

\[
\text{cdf}(k; s_1, \ldots, s_n) = \sum_i \mathbb{I}(k >= s_i) / n \\
\mathbb{I}(k >= s_i) = 1, \text{ if } k >= s_i, \text{ else } 0.
\]

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial
Shannon entropy in nats.

description

Shannon entropy in nats.

usage

tfd_entropy(distribution, ...)

arguments

distribution The distribution being used.

... Additional parameters passed to Python.

value

a Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.

see also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_kl_divergence(),
tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(),
tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()
Examples

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_entropy()
```

**tfd_exponential**  
*Exponential distribution*

**Description**

The Exponential distribution is parameterized by an event rate parameter.

**Usage**

```r
tfd_exponential(
  rate,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Exponential"
)
```

**Arguments**

- `rate`  
  Floating point tensor, equivalent to $1 / \text{mean}$. Must contain only positive values.

- `validate_args`  
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- `allow_nan_stats`  
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- `name`  
  name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The probability density function (pdf) is,

$$pdf(x; \lambda, x > 0) = \exp(-\lambda x) / Z$$

where $Z = 1 / \lambda$ and $Z$ is the normalizing constant.

The Exponential distribution is a special case of the Gamma distribution, i.e.,

$$\text{Exponential}(rate) = \text{Gamma}(\text{concentration}=1., rate)$$
The Exponential distribution uses a rate parameter, or "inverse scale", which can be intuited as,

\[ X \sim \text{Exponential}(rate=1) \]
\[ Y = \frac{X}{\text{rate}} \]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet()`.

Kumaraswamy Distribution.

The Kumaraswamy distribution is defined over the real line using parameters concentration (aka "alpha") and shape (aka "beta"). This distribution is a transformation of the Beta distribution such that X ~ Kumaraswamy(alpha, beta) => (X^alpha) / (1 + (X^alpha)) ~ Beta(alpha, beta).

Value

X is a real number.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet()`.

ffd_kumaraswamy

Kumaraswamy distribution.

Description

The Kumaraswamy distribution is defined over the real line using parameters concentration (aka "alpha") and shape (aka "beta"). This distribution is a transformation of the Beta distribution such that X ~ Kumaraswamy(alpha, beta) => (X^alpha) / (1 + (X^alpha)) ~ Beta(alpha, beta).
Usage

tfd_exp_gamma(
    concentration,
    rate = NULL,
    log_rate = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "ExpGamma"
)

Arguments

concentration  Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
rate           Floating point tensor, the inverse scale params of the distribution(s). Must contain only positive values. Mutually exclusive with log_rate.
log_rate       Floating point tensor, natural logarithm of the inverse scale params of the distribution(s). Mutually exclusive with rate.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.
name           name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) can be derived from the change of variables rule (since the distribution is logically equivalent to tfb_log()(tfd_gamma(..))):

\[
pdf(x; \alpha, \beta > 0) = \exp(x)^{\alpha - 1} \exp(-\exp(x) \beta) / Z + x
\]

\[
Z = \text{Gamma}(\alpha) \beta^{\alpha - 1} \exp(-\exp(x) \beta)
\]

where:

- concentration = \alpha, \alpha > 0.
- rate = \beta, \beta > 0.
- Z is the normalizing constant of the corresponding Gamma distribution, and
- Gamma is the gamma function.

The cumulative density function (cdf) is,

\[
cdf(x; \alpha, \beta, x) = \text{GammaInc}(\alpha, \beta \exp(x)) / \text{Gamma}(\alpha)
\]
where \text{GammaInc} is the lower incomplete Gamma function.

Distribution parameters are automatically broadcast in all functions. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in Figurnov et al., 2018.

**Value**

a distribution instance.

**References**


**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisquare()`,
`tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet()`,
`tfd_dirichlet_multinomial()`, `tfd_dirichlet_categorical()`, `tfd_bernoulli()`, `tfd_categorical()`,
`tfd_crossed_normal()`, `tfd_degrees_of_freedom()`, `tfd_positive_normal()`,
`tfd_uniform()`, `tfd_weibull()`, `tfd_wishart()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet_categorical()`, `tfd_bernoulli()`, `tfd_categorical()`,
`tfd_crossed_normal()`, `tfd_degrees_of_freedom()`, `tfd_positive_normal()`,
`tfd_uniform()`, `tfd_weibull()`, `tfd_wishart()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet_categorical()`, `tfd_bernoulli()`, `tfd_categorical()`,
`tfd_crossed_normal()`, `tfd_degrees_of_freedom()`, `tfd_positive_normal()`,
`tfd_uniform()`, `tfd_weibull()`, `tfd_wishart()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet_categorical()`, `tfd_bernoulli()`, `tfd_categorical()`,
`tfd_crossed_normal()`, `tfd_degrees_of_freedom()`, `tfd_positive_normal()`,
`tfd_uniform()`, `tfd_weibull()`, `tfd_wishart()`,

**Description**

The ExpInverseGamma distribution is defined over the real numbers such that \( X \sim \text{ExpInverseGamma}(\cdot) \Rightarrow \exp(X) \sim \text{InverseGamma}(\cdot) \). The distribution is logically equivalent to `tfb_log()(tfd_inverse_gamma(\cdot))`, but can be sampled with much better precision.
tfd_exp_inverse_gamma

Usage

tfd_exp_inverse_gamma(
    concentration,
    scale = NULL,
    log_scale = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "ExpGamma"
)

Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
- **scale**: Floating point tensor, the scale params of the distribution(s). Must contain only positive values. Mutually exclusive with `log_scale`.
- **log_scale**: Floating point tensor, the natural logarithm of the scale params of the distribution(s). Mutually exclusive with `scale`.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is very similar to ExpGamma,

\[
pdf(x; \alpha, \beta > 0) = \exp(-x)^{\alpha - 1} \exp(-\exp(-x) \beta) / Z - x
\]

\[
Z = \text{Gamma}(\alpha) \beta^{-\alpha}
\]

where:

- concentration = \(\alpha\),
- scale = \(\beta\),
- \(Z\) is the normalizing constant, and,
- Gamma is the **gamma function**.

The cumulative density function (cdf) is,

\[
cdf(x; \alpha, \beta, x) = 1 - \text{GammaInc}(\alpha, \beta \exp(-x)) / \text{Gamma}(\alpha)
\]
where \( \Gamma_{\text{Inc}} \) is the upper incomplete Gamma function.

Distribution parameters are automatically broadcast in all functions. Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in Figurnov et al, 2018.

**Value**

a distribution instance.

**References**


**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Description

ExpRelaxedOneHotCategorical distribution with temperature and logits.

Usage

tfd_exp_relaxed_one_hot_categorical(
    temperature,
    logits = NULL,
    probs = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "ExpRelaxedOneHotCategorical"
)

Arguments

- **temperature**: An 0-D Tensor, representing the temperature of a set of ExpRelaxedCategorical distributions. The temperature should be positive.
- **logits**: An N-D Tensor, N >= 1, representing the log probabilities of a set of ExpRelaxedCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.
- **probs**: An N-D Tensor, N >= 1, representing the probabilities of a set of ExpRelaxedCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. **tfd_sample**, **tfd_log_prob**, **tfd_mean**.
tfd_finite_discrete  

The finite discrete distribution.

Description

The FiniteDiscrete distribution is parameterized by either probabilities or log-probabilities of a set of \( K \) possible outcomes, which is defined by a strictly ascending list of \( K \) values.

Usage

```r
tfd_finite_discrete(
  outcomes,
  logits = NULL,
  probs = NULL,
  rtol = NULL,
  atol = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "FiniteDiscrete"
)
```

Arguments

- **outcomes**: A 1-D floating or integer Tensor, representing a list of possible outcomes in strictly ascending order.
- **logits**: A floating N-D Tensor, \( N \geq 1 \), representing the log probabilities of a set of FiniteDiscrete distributions. The first \( N - 1 \) dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each discrete value. Only one of logits or probs should be passed in.
- **probs**: A floating N-D Tensor, \( N \geq 1 \), representing the probabilities of a set of FiniteDiscrete distributions. The first \( N - 1 \) dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each discrete value. Only one of logits or probs should be passed in.
- **rtol**: Tensor with same dtype as outcomes. The relative tolerance for floating number comparison. Only effective when outcomes is a floating Tensor. Default is \( 10 \times \text{eps} \).
- **atol**: Tensor with same dtype as outcomes. The absolute tolerance for floating number comparison. Only effective when outcomes is a floating Tensor. Default is \( 10 \times \text{eps} \).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name: string prefixed to Ops created by this class.

Details

Note: log_prob, prob, cdf, mode, and entropy are differentiable with respect to logits or probs but not with respect to outcomes.

Mathematical Details

The probability mass function (pmf) is,
\[ \text{pmf}(x; \pi, q_i) = \prod_j \pi_j^{[x == q_i_j]} \]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

---

tfd_gamma  

**Gamma distribution**

Description

The Gamma distribution is defined over positive real numbers using parameters concentration (aka "alpha") and rate (aka "beta").

Usage

```r
  tfd_gamma(  
    concentration,  
    rate,  
    validate_args = FALSE,  
    allow_nan_stats = TRUE,  
    name = "Gamma"  
  )
```

Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
- **rate**: Floating point tensor, the inverse scale params of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; \alpha, \beta, x > 0) = x^{(\alpha - 1)} \exp(-x \beta) / Z
\]

where

- concentration = \alpha, \alpha > 0,
- rate = \beta, \beta > 0,
- Z is the normalizing constant, and,
- Gamma is the gamma function.

The cumulative density function (cdf) is,

\[
cdf(x; \alpha, \beta, x > 0) = \text{GammaInc}(\alpha, \beta x) / \text{Gamma}(\alpha)
\]

where GammaInc is the lower incomplete Gamma function. The parameters can be intuited via their
relationship to mean and stddev,

concentration = \alpha = (mean / stddev)**2
rate = \beta = mean / stddev**2 = concentration / mean

Distribution parameters are automatically broadcast in all functions; see examples for details.

Warning: The samples of this distribution are always non-negative. However, the samples that are
smaller than np\$finfo(dtype)\$tiny are rounded to this value, so it appears more often than it
should. This should only be noticeable when the concentration is very small, or the rate is very
large. See note in tf\$random\_gamma docstring. Samples of this distribution are reparameterized
(pathwise differentiable). The derivatives are computed using the approach described in the paper
Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018

Value

a distribution instance.
**tfd_gamma_gamma**

*Gamma-Gamma distribution*

**Description**

Gamma-Gamma is a compound distribution defined over positive real numbers using parameters concentration, mixing_concentration and mixing_rate.

**Usage**

```r
tfd_gamma_gamma(
    concentration,
    mixing_concentration,
    mixing_rate,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "GammaGamma"
)
```
Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
- **mixing_concentration**: Floating point tensor, the concentration params of the mixing Gamma distribution(s). Must contain only positive values.
- **mixing_rate**: Floating point tensor, the rate params of the mixing Gamma distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

This distribution is also referred to as the beta of the second kind (B2), and can be useful for transaction value modeling, as in Fader and Hardi, 2013.

Mathematical Details

It is derived from the following Gamma-Gamma hierarchical model by integrating out the random variable beta.

\[ \text{beta} \sim \text{Gamma}(\alpha_0, \beta_0) \]
\[ X \mid \text{beta} \sim \text{Gamma}(\alpha, \beta_0) \]

where

- concentration = \alpha
- mixing_concentration = \alpha_0
- mixing_rate = \beta_0

The probability density function (pdf) is

\[ f(x; \alpha, \alpha_0, \beta_0) = Z \ast (x + \beta_0)^(\alpha + \alpha_0) \]

where the normalizing constant \( Z = \text{Beta}(\alpha, \alpha_0) \ast \beta_0^-(\alpha + \alpha_0) \). Samples of this distribution are reparameterized as samples of the Gamma distribution are reparameterized using the technique described in (Figurnov et al., 2018).

@section References:

tfd_gaussian_process

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chol_util(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(),
tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(),
tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hiddmarkov()
tfd_horseshoe(), tfd_independent(), tfd_inverse_gaussian(), tfd_johnson_s_u(),
tfd_joint_distribution_name_auto_batched(), tfd_joint_distribution_name(),
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space()
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family().
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal()
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator().
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(),
tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh().
tfd_skellam(), tfd_spherical_uniform(), tfd_student_t(), tfd_student_t()

Description

A Gaussian process (GP) is an indexed collection of random variables, any finite collection of which
are jointly Gaussian. While this definition applies to finite index sets, it is typically implicit that the
index set is infinite; in applications, it is often some finite dimensional real or complex vector space.
In such cases, the GP may be thought of as a distribution over (real- or complex-valued) functions
defined over the index set.

Usage

tfd_gaussian_process(
    kernel,  
    index_points,  
    mean_fn = NULL,  
)
`observation_noise_variance = 0,`
`jitter = 1e-06,`
`validate_args = FALSE,`
`allow_nan_stats = FALSE,`
`name = "GaussianProcess"  
)

**Arguments**

- **kernel**: PositiveSemidefiniteKernel-like instance representing the GP’s covariance function.

- **index_points**: float Tensor representing finite (batch of) vector(s) of points in the index set over which the GP is defined. Shape has the form \([b_1, ..., b_B, e_1, f_1, ..., f_F]\) where \(F\) is the number of feature dimensions and must equal `kernel.feature_ndims` and \(e_1\) is the number (size) of index points in each batch (we denote it \(e_1\) to distinguish it from the number of inducing index points, denoted \(e_2\) below). Ultimately the GaussianProcess distribution corresponds to an \(e_1\)-dimensional multivariate normal. The batch shape must be broadcastable with `kernel.batch_shape`, the batch shape of `inducing_index_points`, and any batch dims yielded by `mean_fn`.

- **mean_fn**: function that acts on index points to produce a (batch of) vector(s) of mean values at those index points. Takes a Tensor of shape \([b_1, ..., b_B, f_1, ..., f_F]\) and returns a Tensor whose shape is (broadcastable with) \([b_1, ..., b_B]\). Default value: NULL implies constant zero function.

- **observation_noise_variance**: float Tensor representing the variance of the noise in the Normal likelihood distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (`kernel.batch_shape`, `index_points`, etc.). Default value: 0.

- **jitter**: float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: 1e-6.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.

**Details**

Just as Gaussian distributions are fully specified by their first and second moments, a Gaussian process can be completely specified by a mean and covariance function. Let \(S\) denote the index set and \(\mathbb{K}\) the space in which each indexed random variable takes its values (again, often \(\mathbb{R}\) or \(\mathbb{C}\)). The mean function is then a map \(m: S \rightarrow \mathbb{K}\), and the covariance function, or kernel, is a positive-definite
function \( k: (S \times S) \rightarrow K \). The properties of functions drawn from a GP are entirely dictated (up to translation) by the form of the kernel function.

This Distribution represents the marginal joint distribution over function values at a given finite collection of points \([x[1], ..., x[N]]\) from the index set \(S\). By definition, this marginal distribution is just a multivariate normal distribution, whose mean is given by the vector \([m(x[1]), ..., m(x[N])]\) and whose covariance matrix is constructed from pairwise applications of the kernel function to the given inputs:

\[
\begin{pmatrix}
  k(x[1], x[1]) & k(x[1], x[2]) & \cdots & k(x[1], x[N]) \\
  k(x[2], x[1]) & k(x[2], x[2]) & \cdots & k(x[2], x[N]) \\
  \vdots & \vdots & \ddots & \vdots \\
  k(x[N], x[1]) & k(x[N], x[2]) & \cdots & k(x[N], x[N])
\end{pmatrix}
\]

For this to be a valid covariance matrix, it must be symmetric and positive definite; hence the requirement that \( k \) be a positive definite function (which, by definition, says that the above procedure will yield PD matrices).

We also support the inclusion of zero-mean Gaussian noise in the model, via the \texttt{observation_noise_variance} parameter. This augments the generative model to

\[
f \sim \text{GP}(m, k) \\
(y[i] | f, x[i]) \sim \text{Normal}(f(x[i]), s)
\]

where

- \( m \) is the mean function
- \( k \) is the covariance kernel function
- \( f \) is the function drawn from the GP
- \( x[i] \) are the index points at which the function is observed
- \( y[i] \) are the observed values at the index points
- \( s \) is the scale of the observation noise.

Note that this class represents an \textit{unconditional} Gaussian process; it does not implement posterior inference conditional on observed function evaluations. This class is useful, for example, if one wishes to combine a GP prior with a non-conjugate likelihood using MCMC to sample from the posterior.

Mathematical Details

The probability density function (pdf) is a multivariate normal whose parameters are derived from the GP’s properties:

\[
\text{pdf}(x; \text{index_points}, \text{mean_fn}, \text{kernel}) = \exp(-0.5 \ast y) / Z
\]

\[
K = (\text{kernel.matrix}(\text{index_points, index_points}) + \\
(\text{observation_noise_variance} + \text{jitter}) \ast \text{eye}(N))
\]

\[
y = (x - \text{mean_fn}(\text{index_points}))^T \ast K \ast (x - \text{mean_fn}(\text{index_points}))
\]

\[
Z = (2 \ast \pi)^{0.5 \ast N} \mid \text{det}(K)\mid^{0.5}
\]

where:
• index_points are points in the index set over which the GP is defined,
• mean_fn is a callable mapping the index set to the GP’s mean values,
• kernel is PositiveSemidefiniteKernel-like and represents the covariance function of the GP,
• observation_noise_variance represents (optional) observation noise.
• jitter is added to the diagonal to ensure positive definiteness up to machine precision (otherwise Cholesky-decomposition is prone to failure),
• eye(N) is an N-by-N identity matrix.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chol(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(), tfd_dirichlet_multinomial(), tfd_dirichlet_categorical(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gammma(), tfd_gaussian_process_regression_model(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_student_t(), tfd_negate_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
Usage

tfd_gaussian_process_regression_model(
    kernel,
    index_points = NULL,
    observation_index_points = NULL,
    observations = NULL,
    observation_noise_variance = 0,
    predictive_noise_variance = NULL,
    mean_fn = NULL,
    jitter = 1e-06,
    validate_args = FALSE,
    allow_nan_stats = FALSE,
    name = "GaussianProcessRegressionModel"
)

Arguments

kernel PositiveSemidefiniteKernel-like instance representing the GP's covariance function.

index_points float Tensor representing finite (batch of) vector(s) of points in the index set over which the GP is defined. Shape has the form [b1, ..., bB, e1, f1, ..., fF] where F is the number of feature dimensions and must equal kernel$feature_ndims and e1 is the number (size) of index points in each batch (we denote it e1 to distinguish it from the number of inducing index points, denoted e2 below). Ultimately the GaussianProcess distribution corresponds to an e1-dimensional multivariate normal. The batch shape must be broadcastable with kernel$batch_shape, the batch shape of inducing_index_points, and any batch dims yielded by mean_fn.

observation_index_points Tensor representing finite collection, or batch of collections, of points in the index set for which some data has been observed. Shape has the form [b1, ..., bB, e, f1, ..., fF] where F is the number of feature dimensions and must equal kernel$feature_ndims, and e is the number (size) of index points in each batch. [b1, ..., bB, e] must be broadcastable with the shape of observations, and [b1, ..., bB] must be broadcastable with the shapes of all other batched parameters (kernel.batch_shape, index_points, etc). The default value is None, which corresponds to the empty set of observations, and simply results in the prior predictive model (a GP with noise of variance predictive_noise_variance).

observations Tensor representing collection, or batch of collections, of observations corresponding to observation_index_points. Shape has the form [b1, ..., bB, e], which must be broadcastable with the batch and example shapes of observation_index_points. The batch shape [b1, ..., bB] must be broadcastable with the shapes of all other batched parameters (kernel.batch_shape, index_points, etc.). The default value is None, which corresponds to the empty set of observations, and simply results in the prior predictive model (a GP with noise of variance predictive_noise_variance).

observation_noise_variance float Tensor representing the variance of the noise in the Normal likelihood
distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (kernel\$batch_shape, index_points, etc.). Default value: 0.

**predictive_noise_variance**

Tensor representing the variance in the posterior predictive model. If None, we simply re-use observation_noise_variance for the posterior predictive noise. If set explicitly, however, we use this value. This allows us, for example, to omit predictive noise variance (by setting this to zero) to obtain noiseless posterior predictions of function values, conditioned on noisy observations.

**mean_fn**

Callable that acts on index_points to produce a collection, or batch of collections, of mean values at index_points. Takes a Tensor of shape \([b1, ..., bB, f1, ..., fF]\) and returns a Tensor whose shape is broadcastable with \([b1, ..., bB]\). Default value: None implies the constant zero function.

**jitter**

Float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: 1e-6.

**validate_args**

Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

**allow_nan_stats**

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

**name**

Name prefixed to Ops created by this class.

**Value**

A distribution instance.

**See Also**

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator().
The Generalized Normal distribution.

Description

The Generalized Normal (or Generalized Gaussian) generalizes the Normal distribution with an additional shape parameter. It is parameterized by location `loc`, scale `scale` and shape `power`.

Usage

```r
  tfd_generalized_normal(
    loc,    scale,    power,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "GeneralizedNormal"
  )
```

Arguments

- `loc` Floating point tensor; the means of the distribution(s).
- `scale` Floating point tensor; the scale of the distribution(s). Must contain only positive values.
- `power` Floating point tensor; the shape parameter of the distribution(s). Must contain only positive values. `loc`, `scale` and `power` must have compatible shapes for broadcasting.
- `validate_args` Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` name prefixed to Ops created by this class.
Details

Mathematical details The probability density function (pdf) is,

\[
\text{pdf}(x; \text{loc}, \text{scale}, \text{power}) = \frac{1}{2 \times \text{scale} \times \Gamma(1 + 1 / \text{power})} \times \\
\exp(-(|x - \text{loc}| / \text{scale})^\text{power})
\]

where \(\text{loc}\) is the mean, \(\text{scale}\) is the scale, and, \(\text{power}\) is the shape parameter. If the power is above two, the distribution becomes platykurtic. A power equal to two results in a Normal distribution. A power smaller than two produces a leptokurtic (heavy-tailed) distribution. Mean and scale behave the same way as in the equivalent Normal distribution.


Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfdDeterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tril(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
The Generalized Pareto distribution.

Description

The Generalized Pareto distributions are a family of continuous distributions on the reals. Special cases include Exponential (when $\text{loc} = 0$, $\text{concentration} = 0$), Pareto (when $\text{concentration} > 0$, $\text{loc} = \text{scale} / \text{concentration}$), and Uniform (when $\text{concentration} = -1$).

Usage

tfd_generalized_pareto(
    loc,  # The location / shift of the distribution. GeneralizedPareto is a location-scale distribution. This parameter lower bounds the distribution’s support. Must broadcast with scale, concentration. Floating point Tensor.
    scale,  # The scale of the distribution. GeneralizedPareto is a location-scale distribution, so doubling the scale doubles a sample and halves the density. Strictly positive floating point Tensor. Must broadcast with loc, concentration.
    concentration,  # The shape parameter of the distribution. The larger the magnitude, the more the distribution concentrates near loc (for concentration $\geq 0$) or near $\text{loc} - (\text{scale}/\text{concentration})$ (for concentration $< 0$). Floating point Tensor.
    validate_args = FALSE,  # Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
    allow_nan_stats = TRUE,  # Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
    name = NULL  # name prefixed to Ops created by this class.
)

Arguments

- **loc**: The location / shift of the distribution. GeneralizedPareto is a location-scale distribution. This parameter lower bounds the distribution’s support. Must broadcast with scale, concentration. Floating point Tensor.
- **scale**: The scale of the distribution. GeneralizedPareto is a location-scale distribution, so doubling the scale doubles a sample and halves the density. Strictly positive floating point Tensor. Must broadcast with loc, concentration.
- **concentration**: The shape parameter of the distribution. The larger the magnitude, the more the distribution concentrates near loc (for concentration $\geq 0$) or near $\text{loc} - (\text{scale}/\text{concentration})$ (for concentration $< 0$). Floating point Tensor.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

This distribution is often used to model the tails of other distributions. As a member of the location-scale family, $X \sim \text{GeneralizedPareto}(\text{loc}=\text{loc}, \text{scale}=\text{scale}, \text{concentration}=\text{conc})$ maps to $Y \sim \text{GeneralizedPareto}(\text{loc}=0, \text{scale}=1, \text{concentration}=\text{conc})$ via $Y = (X - \text{loc}) / \text{scale}$. 
For positive concentrations, the distribution is equivalent to a hierarchical Exponential-Gamma
model with \(X|\text{rate} \sim \text{Exponential}(\text{rate})\) and \(\text{rate} \sim \text{Gamma}(\text{concentration}=1/\text{concentration}, \text{scale}=\text{scale} / \text{concentration})\). In the following, \(\text{samps1}\) and \(\text{samps2}\) are identically distributed:

\[
\text{genp} \leftarrow \text{tfd\_generalized\_pareto}(\text{loc} = 0, \text{scale} = \text{scale}, \text{concentration} = \text{conc})
\]
\[
\text{samps1} \leftarrow \text{genp} \%\% \text{tfd\_sample}(1000)
\]
\[
\text{jd} \leftarrow \text{tfd\_joint\_distribution\_named}(\text{list}(
\text{  rate} = \text{tfd\_gamma}(1 / \text{genp\$concentration}, \text{genp\$scale} / \text{genp\$concentration}),
\text{  x} = \text{function}(\text{rate}) \text{tfd\_exponential}(\text{rate}))
\text{samps2} \leftarrow \text{jd} \%\% \text{tfd\_sample}(1000) \%\% .x
\]

The support of the distribution is always lower bounded by \(\text{loc}\). When \(\text{concentration} < 0\), the
support is also upper bounded by \(\text{loc} + \text{scale} / \text{abs(concentration)}\).

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \text{mu}, \text{sigma}, \text{shp}, x > \text{mu}) = \frac{1 + \text{shp} \times (x - \text{mu}) / \text{sigma}}{\text{sigma}}^{-1 / \text{shp} - 1}
\]

where:

- \(\text{concentration} = \text{shp}\), any real value,
- \(\text{scale} = \text{sigma}\), \(\text{sigma} > 0\),
- \(\text{loc} = \text{mu}\).

The cumulative density function (cdf) is,

\[
\text{cdf}(x; \text{mu}, \text{sigma}, \text{shp}, x > \text{mu}) = 1 - \left(1 + \text{shp} \times (x - \text{mu}) / \text{sigma}\right)^{(-1 / \text{shp})}
\]

Distribution parameters are automatically broadcast in all functions; see examples for details. Sam-
ples of this distribution are reparameterized (pathwise differentiable).

Value

a distribution instance.

See Also

For usage examples see e.g. \text{tfd\_sample()}, \text{tfd\_log\_prob()}, \text{tfd\_mean()}.
The Geometric distribution is parameterized by $p$, the probability of a positive event. It represents the probability that in $k + 1$ Bernoulli trials, the first $k$ trials failed, before seeing a success. The pmf of this distribution is:

$$p(k; p) = (1 - p)^k \times p$$

where:

- $p$ is the success probability, $0 < p \leq 1$, and,
- $k$ is a non-negative integer.

**Arguments**

- **logits**: Floating-point Tensor with shape $[B1, ..., Bb]$ where $b \geq 0$ indicates the number of batch dimensions. Each entry represents logits for the probability of success for independent Geometric distributions and must be in the range $(-\infty, \infty]$. Only one of logits or probs should be specified.

- **probs**: Positive floating-point Tensor with shape $[B1, ..., Bb]$ where $b \geq 0$ indicates the number of batch dimensions. Each entry represents the probability of success for independent Geometric distributions and must be in the range $(0, 1]$. Only one of logits or probs should be specified.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nans**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.
Scalar Gumbel distribution with location `loc` and scale parameters

### Description

Mathematical details

### Usage

```python
tfd_gumbel(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Gumbel"
)```
Arguments

- **loc**
  - Floating point tensor, the means of the distribution(s).

- **scale**
  - Floating point tensor, the scales of the distribution(s). ‘scale” must contain only positive values.

- **validate_args**
  - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  - Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  - name prefixed to Ops created by this class.

Details

The probability density function (pdf) of this distribution is,

\[
\text{pdf}(x; \mu, \sigma) = \exp\left(-\frac{x - \mu}{\sigma} - \exp\left(-\frac{x - \mu}{\sigma}\right)\right) / \sigma
\]

where \(\text{loc} = \mu\) and \(\text{scale} = \sigma\).

The cumulative density function of this distribution is, \(\text{cdf}(x; \mu, \sigma) = \exp(-\exp(-(x - \mu) / \sigma))\)

The Gumbel distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{Gumbel}(\text{loc}=0, \text{scale}=1)
\]

\[
Y = \text{loc} + \text{scale} * X
\]

Value

- a distribution instance.

See Also

Other distributions: 
- `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
- `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
- `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`,
- `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`,
- `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`,
- `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`,
- `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`,
- `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`,
- `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`,
- `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_log_normal_quadrature_compound()`,
- `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`,
- `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_diag_plus_low_rank()`,
- `tfd_multivariate_normal_linear_operator()`,
- `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`,
- `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`,
- `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`,
- `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`,
tfd_half_cauchy

Half-Cauchy distribution

Description
The half-Cauchy distribution is parameterized by a loc and a scale parameter. It represents the right half of the two symmetric halves in a Cauchy distribution.

Usage
tfd_half_cauchy(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "HalfCauchy"
)

Arguments
loc Floating-point Tensor; the location(s) of the distribution(s).
scale Floating-point Tensor; the scale(s) of the distribution(s). Must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) for the half-Cauchy distribution is given by

\[
pdf(x; \text{loc}, \text{scale}) = \frac{2}{\pi \text{scale} (1 + z^2)}
\]

\[
z = \frac{x - \text{loc}}{\text{scale}}
\]
where `loc` is a scalar in \( \mathbb{R} \) and `scale` is a positive scalar in \( \mathbb{R} \). The support of the distribution is given by the interval \([loc, \infty)\).

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**Description**

Mathematical details

**Usage**

```python
tfd_half_normal(
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "HalfNormal"
)
```
Arguments

scale: Floating point tensor; the scales of the distribution(s). Must contain only positive values.

validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name: name prefixed to Ops created by this class.

Details

The half normal is a transformation of a centered normal distribution. If some random variable $X$ has normal distribution,

$$X \sim \text{Normal}(0.0, \text{scale})$$

$$Y = |X|$$

Then $Y$ will have half normal distribution. The probability density function (pdf) is:

$$\text{pdf}(x; \text{scale}, x > 0) = \sqrt{2} / (\text{scale} \times \sqrt{\pi}) \times \exp(- 1/2 \times (x / \text{scale})^2))$$

Where $\text{scale} = \sigma$ is the standard deviation of the underlying normal distribution.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
`tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gaussian()`, `tfd_exponential()`,
`tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`,
`tfd_gaussian_process_multivariate()`, `tfd_gaussian_process_regression_model()`, `tfd_hidden_markov_model()`,
`tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_bernoulli()`,
`tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential()`,
`tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`,
`tfd_lkj()`, `tfd_log_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`,
`tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`,
`tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tril()`}, `tfd_multivariate_student_t_linear_operator()`, `tfd_neg_binomial()`,
`tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature()`,
`tfd_poisson()`}, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`}, `tfd_relaxed_bernoulli()`
tfd_hidden_markov_model

Hidden Markov model distribution

Description
The HiddenMarkovModel distribution implements a (batch of) hidden Markov models where the initial states, transition probabilities and observed states are all given by user-provided distributions.

Usage
```r
  tfd_hidden_markov_model(
    initial_distribution,
    transition_distribution,
    observation_distribution,
    num_steps,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "HiddenMarkovModel"
  )
```

Arguments

- **initial_distribution**: A Categorical-like instance. Determines probability of first hidden state in Markov chain. The number of categories must match the number of categories of transition_distribution as well as both the rightmost batch dimension of transition_distribution and the rightmost batch dimension of observation_distribution.

- **transition_distribution**: A Categorical-like instance. The rightmost batch dimension indexes the probability distribution of each hidden state conditioned on the previous hidden state.

- **observation_distribution**: A tfp$distributions$Distribution-like instance. The rightmost batch dimension indexes the distribution of each observation conditioned on the corresponding hidden state.

- **num_steps**: The number of steps taken in Markov chain. An integer.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
This model assumes that the transition matrices are fixed over time. In this model, there is a se-
quence of integer-valued hidden states: z[0], z[1], ..., z[num_steps - 1] and a sequence of observed
states: x[0], ..., x[num_steps - 1].

The distribution of z[0] is given by initial_distribution. The conditional probability of z[i +
1] given z[i] is described by the batch of distributions in transition_distribution. For a batch
of hidden Markov models, the coordinates before the rightmost one of the transition_distribution
batch correspond to indices into the hidden Markov model batch. The rightmost coordinate of the
batch is used to select which distribution z[i + 1] is drawn from. The distributions corresponding
to the probability of z[i + 1] conditional on z[i] == k is given by the elements of the batch whose
rightmost coordinate is k.

Similarly, the conditional distribution of z[i] given x[i] is given by the batch of observation_distribution.
When the rightmost coordinate of observation_distribution is k it gives the conditional proba-
bilities of x[i] given z[i] == k. The probability distribution associated with the HiddenMarkovModel
distribution is the marginal distribution of x[0],...,x[num_steps - 1].

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial,
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(),
tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_seq-
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space,
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal,
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature,
tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(),
tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribu-
tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian,...
**tfd_horseshoe**

**Horseshoe distribution**

**Description**

The so-called 'horseshoe' distribution is a Cauchy-Normal scale mixture, proposed as a sparsity-inducing prior for Bayesian regression. It is symmetric around zero, has heavy (Cauchy-like) tails, so that large coefficients face relatively little shrinkage, but an infinitely tall spike at 0, which pushes small coefficients towards zero. It is parameterized by a positive scalar scale parameter: higher values yield a weaker sparsity-inducing effect.

**Usage**

```r
@tfd_horseshoe(
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Horseshoe"
)
```

**Arguments**

- **scale**: Floating point tensor; the scales of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

**Details**

Mathematical details

The Horseshoe distribution is centered at zero, with scale parameter $\lambda$. It is defined by:

$$\text{horseshoe}(\text{scale} = \lambda) \sim \text{Normal}(0, \lambda \sigma)$$

where $\sigma \sim \text{half}\text{-}\text{cauchy}(0,1)$
tfd_independent

Independent distribution from batch of distributions

**Description**

This distribution is useful for regarding a collection of independent, non-identical distributions as a single random variable. For example, the Independent distribution composed of a collection of Bernoulli distributions might define a distribution over an image (where each Bernoulli is a distribution over each pixel).
tfd_independent

Usage

tfd_independent(
    distribution,
    reinterpreted_batch_ndims = NULL,
    validate_args = FALSE,
    name = paste0("Independent", distribution$name)
)

Arguments

distribution The base distribution instance to transform. Typically an instance of Distribution
reinterpreted_batch_ndims Scalar, integer number of rightmost batch dims which will be regarded as event
dims. When NULL all but the first batch axis (batch axis 0) will be transferred
to event dimensions (analogous to tf$layers$flatten).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.
name The name for ops managed by the distribution. Default value: Independent +
distribution.name.

Details

More precisely, a collection of B (independent) E-variate random variables (rv) \{X_1, ..., X_B\}, can
be regarded as a \([B, E]\)-variate random variable \((X_1, ..., X_B)\) with probability
\(p(x_1, \ldots, x_B) = p_1(x_1) \times \ldots \times p_B(x_B)\) where \(p_b(X_b)\) is the probability of the \(b\)-th rv. More generally
\(B, E\) can be arbitrary shapes. Similarly, the Independent distribution specifies a distribution over
\([B, E]\)-shaped events. It operates by reinterpreting the rightmost batch dims as part of the event
dimensions. The \texttt{reinterpreted_batch_ndims} parameter controls the number of batch dims which
are absorbed as event dims; \texttt{reinterpreted_batch_ndims} <= \texttt{len(batch_shape)}. For example,
the \texttt{log_prob} function entails a \texttt{reduce_sum} over the rightmost \texttt{reinterpreted_batch_ndims} af-
ter calling the base distribution’s \texttt{log_prob}. In other words, since the batch dimension(s) index
independent distributions, the resultant multivariate will have independent components.

Mathematical Details

The probability function is,

\[
\text{prob}(x; \text{reinterpreted_batch_ndims}) =
\text{tf.reduce_prod(dist.prob}(x), \text{axis}=-1-\text{range(reinterpreted_batch_ndims))}
\]

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}. 

---

### Description

The `InverseGamma` distribution is defined over positive real numbers using parameters concentration (aka "alpha") and scale (aka "beta").

### Usage

```python
  tfd_inverse_gamma(
    concentration,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "InverseGamma"
  )
```

### Arguments

- **concentration**: Floating point tensor, the concentration params of the distribution(s). Must contain only positive values.
scale Floating point tensor, the scale params of the distribution(s). Must contain only positive values. This parameter was called rate before release 0.8.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
pdf(x; \alpha, \beta, x > 0) = x^{-(\alpha + 1)} \exp\left(-\frac{\beta}{x}\right) / Z
\]

\[
Z = \Gamma(\alpha) \beta^{-\alpha}
\]

where:

- concentration = \(\alpha\),
- scale = \(\beta\),
- \(Z\) is the normalizing constant, and,
- \(\Gamma\) is the gamma function.

The cumulative density function (cdf) is,

\[
cdf(x; \alpha, \beta, x > 0) = \frac{\Gamma_{\text{inc}}(\alpha, \beta / x)}{\Gamma(\alpha)}
\]

where \(\Gamma_{\text{inc}}\) is the [upper incomplete Gamma function](https://en.wikipedia.org/wiki/Incomplete_gamma_function).

The parameters can be intuited via their relationship to mean and variance when these moments exist,

mean = \(\beta / (\alpha - 1)\) when \(\alpha > 1\)
variance = \(\beta^2 / (\alpha - 1)^2 / (\alpha - 2)\) when \(\alpha > 2\)

i.e., under the same conditions:

\[
\alpha = \text{mean}^2 / \text{variance} + 2 \quad \text{beta} = \text{mean} * (\text{mean}^2 / \text{variance} + 1)
\]

Distribution parameters are automatically broadcast in all functions; see examples for details.

Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper


[gamma function]: R:gamma

[upper incomplete Gamma function]: R:upper_incgamma

[Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018]: R:Michael
Inverse Gaussian distribution

Description

The inverse Gaussian distribution is parameterized by a loc and a concentration parameter. It’s also known as the Wald distribution. Some, e.g., the Python scipy package, refer to the special case when loc is 1 as the Wald distribution.

Usage

```python
tfd_inverse_gaussian(
    loc,
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "InverseGaussian"
)
```
**Arguments**

- **loc**: Floating-point Tensor, the loc params. Must contain only positive values.
- **concentration**: Floating-point Tensor, the concentration params. Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

**Details**

The "inverse" in the name does not refer to the distribution associated to the multiplicative inverse of a random variable. Rather, the cumulant generating function of this distribution is the inverse to that of a Gaussian random variable.

**Mathematical Details**

The probability density function (pdf) is,

\[
\text{pdf}(x; \mu, \lambda) = \left(\frac{\lambda}{2 \pi x^3}\right)^{0.5} \exp\left\{-\frac{\lambda(x - \mu)^2}{2\mu^2 x}\right\}
\]

where

- **loc = mu**
- **concentration = lambda**.

The support of the distribution is defined on (0, infinity). Mapping to R and Python scipy’s parameterization:

- **R**: statmod::invgauss
- **mean = loc**
- **shape = concentration**
- **dispersion = 1 / concentration. Used only if shape is NULL.**
- **Python**: scipy.stats.invgauss
- **mu = loc / concentration**
- **scale = concentration**

**Value**

a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma()`.


---

**tfd_johnson_s_u**  
*Johnson’s SU-distribution.*

---

**Description**

This distribution has parameters: shape parameters skewness and tailweight, location `loc`, and scale.

**Usage**

```r
  tfd_johnson_s_u(
    skewness,
    tailweight,
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = NULL
  )
```
Arguments

- **skewness** (Floating-point Tensor): Skewness of the distribution(s).
- **tailweight** (Floating-point Tensor): Tail weight of the distribution(s). tailweight must contain only positive values.
- **loc** (Floating-point Tensor): The mean(s) of the distribution(s).
- **scale** (Floating-point Tensor): The scaling factor(s) for the distribution(s). Note that scale is not technically the standard deviation of this distribution but has semantics more similar to standard deviation than variance.
- **validate_args** (Logical, default FALSE): When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats** (Logical, default TRUE): When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name** (name prefixed to Ops created by this class).

Details

Mathematical details

The probability density function (pdf) is,

\[
\text{pdf}(x; s, t, xi, sigma) = \exp(-0.5 \cdot (s + t \cdot \text{arcsinh}(y))**2) / Z
\]

where,

- \(s = \text{skewness}\)
- \(t = \text{tailweight}\)
- \(y = (x - xi) / sigma\)
- \(Z = sigma \sqrt{2 \pi} \sqrt{1 + y**2} / t\)

where:

- \(\text{loc} = xi\),
- \(\text{scale} = sigma\), and,
- \(Z\) is the normalization constant. The JohnsonSU distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{JohnsonSU}(\text{skewness}, \text{tailweight}, \text{loc}=0, \text{scale}=1)
\]

\[
Y = \text{loc} + \text{scale} * X
\]

Value

a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


tfd_joint_distribution_named

*Joint distribution parameterized by named distribution-making functions.*

Description

This distribution enables both sampling and joint probability computation from a single model specification. A joint distribution is a collection of possibly interdependent distributions. Like `JointDistributionSequential`, `JointDistributionNamed` is parameterized by several distribution-making functions. Unlike `JointDistributionNamed`, each distribution-making function must have its own key. Additionally every distribution-making function’s arguments must refer to only specified keys.

Usage

```r
  tfd_joint_distribution_named(model, validate_args = FALSE, name = NULL)
```
Arguments

- **model**: named list of distribution-making functions each with required args corresponding only to other keys in the named list.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **name**: The name for ops managed by the distribution. Default value: "JointDistributionNamed".

Details

Mathematical Details

Internally JointDistributionNamed implements the chain rule of probability. That is, the probability function of a length- \(d\) vector \(x\) is,

\[
p(x) = \prod_{i=0}^{d-1} p(x[i] | x[:i])
\]

The JointDistributionNamed is parameterized by a dict (or namedtuple) composed of either:

1. `tfp$ distributions$ Distribution-like instances or,
2. functions which return a `tfp$ distributions$ Distribution-like instance. The "conditioned on" elements are represented by the function’s required arguments; every argument must correspond to a key in the named distribution-making functions. Distribution-makers which are directly a Distribution-like instance are allowed for convenience and semantically identical a zero argument function. When the maker takes no arguments it is preferable to directly provide the distribution instance.

Value

A distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`.
tfd_joint_distribution_named_auto_batched

Joint distribution parameterized by named distribution-making functions.

Description

This class provides automatic vectorization and alternative semantics for `tfd_joint_distribution_named()`, which in many cases allows for simplifications in the model specification.

Usage

```r
tfd_joint_distribution_named_auto_batched(
  model,
  batch_ndims = 0,
  use_vectorized_map = TRUE,
  validate_args = FALSE,
  name = NULL
)
```

Arguments

- **model**: A generator that yields a sequence of `tfd$Distribution`-like instances.
- **batch_ndims**: integer Tensor number of batch dimensions. The batch shapes of all component distributions must be such that the prefixes of length `batch_ndims` broadcast to a consistent joint batch shape. Default value: 0.
- **use_vectorized_map**: logical. Whether to use `tf$vectorized_map` to automatically vectorize evaluation of the model. This allows the model specification to focus on drawing a single sample, which is often simpler, but some ops may not be supported. Default value: TRUE.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **name**: name prefixed to Ops created by this class.
Details

Automatic vectorization

Auto-vectorized variants of JointDistribution allow the user to avoid explicitly annotating a model's vectorization semantics. When using manually-vectorized joint distributions, each operation in the model must account for the possibility of batch dimensions in Distributions and their samples. By contrast, auto-vectorized models need only describe a single sample from the joint distribution; any batch evaluation is automated using tf$vectorized_map as required. In many cases this allows for significant simplifications. For example, the following manually-vectorized `tfd_joint_distribution_named()` model:

```r
model <- tfd_joint_distribution_sequential(
  list(
    x = tfd_normal(loc = 0, scale = tf$ones(3L)),
    y = tfd_normal(loc = 0, scale = 1),
    z = function(y, x) {
      tfd_normal(loc = x[reticulate::py_ellipsis(), 1:2] + y[reticulate::py_ellipsis(), tf$newaxis], scale = 1)
    }
  )
)
```

can be written in auto-vectorized form as

```r
model <- tfd_joint_distribution_sequential_auto_batched(
  list(
    x = tfd_normal(loc = 0, scale = tf$ones(3L)),
    y = tfd_normal(loc = 0, scale = 1),
    z = function(y, x) {tfd_normal(loc = x[1:2] + y, scale = 1)}
  )
)
```
in which we were able to avoid explicitly accounting for batch dimensions when indexing and slicing computed quantities in the third line. Note: auto-vectorization is still experimental and some TensorFlow ops may be unsupported. It can be disabled by setting use_vectorized_map=FALSE.

Alternative batch semantics

This class also provides alternative semantics for specifying a batch of independent (non-identical) joint distributions. Instead of simply summing the `log_prob` of component distributions (which may have different shapes), it first reduces the component `log_probs` to ensure that `jd$log_prob(jd$sample())` always returns a scalar, unless `batch_ndims` is explicitly set to a nonzero value (in which case the result will have the corresponding tensor rank).

The essential changes are:

- An event of JointDistributionNamedAutoBatched is the list of tensors produced by `$sample()`; thus, the `event_shape` is the list containing the shapes of sampled tensors. These combine both the event and batch dimensions of the component distributions. By contrast, the event shape of a base JointDistributions does not include batch dimensions of component distributions.
- The `batch_shape` is a global property of the entire model, rather than a per-component property as in base JointDistributions. The global batch shape must be a prefix of the batch shapes of each component; the length of this prefix is specified by an optional argument `batch_ndims`. If `batch_ndims` is not specified, the model has batch shape().

tfd_joint_distribution_sequential

Joint distribution parameterized by distribution-making functions

Description

This distribution enables both sampling and joint probability computation from a single model specification.

Usage

tfd_joint_distribution_sequential(model, validate_args = FALSE, name = NULL)

Arguments

model list of either tfp$Distributions$Distribution instances and/or functions which take the k previous distributions and returns a new tfp$Distributions$Distribution instance.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

name name prefixed to Ops created by this class.

Details

A joint distribution is a collection of possibly interdependent distributions. Like tf.keras.Sequential, the JointDistributionSequential can be specified via a list of functions (each responsible for making a tfp.distributions.Distribution-like instance). Unlike tf.keras.Sequential, each function can depend on the output of all previous elements rather than only the immediately previous.

Mathematical Details

The JointDistributionSequential implements the chain rule of probability.
That is, the probability function of a length-d vector \( x \) is,

\[
p(x) = \prod_{i=0}^{d-1} p(x[i] | x[:i])
\]

The JointDistributionSequential is parameterized by a list comprised of either:

1. tfp.distributions.Distribution-like instances or,
2. callables which return a tfp.distributions.Distribution-like instance. Each list element implements the \( i \)-th full conditional distribution, \( p(x[i] | x[:i]) \). The "conditioned on" elements are represented by the callable's required arguments. Directly providing a Distribution-like instance is a convenience and is semantically identical a zero argument callable. Denote the \( i \)-th callables non-default arguments as \( args[i] \). Since the callable is the conditional manifest, \( 0 \leq \text{len}(args[i]) \leq i - 1 \). When \( \text{len}(args[i]) < i - 1 \), the callable only depends on a subset of the previous distributions, specifically those at indexes: \( \text{range}(i - 1, i - 1 - \text{num_args[i]}, -1) \).

Name resolution: The names of JointDistributionSequential components are defined by explicit-name arguments passed to distributions (tfd.Normal(0., 1., name='x')) and/or by the argument names in distribution-making functions (lambda x: tfd.Normal(x., 1.)). Both approaches may be used in the same distribution, as long as they are consistent; referring to a single component by multiple names will raise a ValueError. Unnamed components will be assigned a dummy name.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial,
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
Joint distribution parameterized by distribution-making functions.

Description
This class provides automatic vectorization and alternative semantics for \texttt{tfd\_joint\_distribution\_sequential()}, which in many cases allows for simplifications in the model specification.

Usage
\begin{verbatim}
tfd\_joint\_distribution\_sequential\_auto\_batched(
    model,
    batch_ndims = 0,
    use\_vectorized\_map = TRUE,
    validate\_args = FALSE,
    name = NULL
)
\end{verbatim}

Arguments
- **model**: A generator that yields a sequence of \texttt{tfd\_Distribution}-like instances.
- **batch_ndims**: integer \texttt{Tensor} number of batch dimensions. The batch shapes of all component distributions must be such that the prefixes of length \texttt{batch\_ndims} broadcast to a consistent joint batch shape. Default value: 0.
use_vectorized_map

  logical. Whether to use \$vectorized_map to automatically vectorize evaluation of the model. This allows the model specification to focus on drawing a single sample, which is often simpler, but some ops may not be supported. Default value: TRUE.

validate_args

  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

name

  name prefixed to Ops created by this class.

Details

Automatic vectorization

Auto-vectorized variants of JointDistribution allow the user to avoid explicitly annotating a model’s vectorization semantics. When using manually-vectorized joint distributions, each operation in the model must account for the possibility of batch dimensions in Distributions and their samples. By contrast, auto-vectorized models need only describe a single sample from the joint distribution; any batch evaluation is automated using \$vectorized_map as required. In many cases this allows for significant simplifications. For example, the following manually-vectorized tfd_joint_distribution_sequential() model:

```r
model <- tfd_joint_distribution_sequential(
  list(
    tfd_normal(loc = 0, scale = tf$ones(3L)),
    tfd_normal(loc = 0, scale = 1),
    function(y, x) {
      tfd_normal(loc = x[reticulate::py_ellipsis(), 1:2] + y[reticulate::py_ellipsis(), tf$newaxis], scale = 1)
    }
  )
)
```

can be written in auto-vectorized form as

```r
model <- tfd_joint_distribution_sequential_auto_batched(
  list(
    tfd_normal(loc = 0, scale = tf$ones(3L)),
    tfd_normal(loc = 0, scale = 1),
    function(y, x) {tfd_normal(loc = x[1:2] + y, scale = 1)}
  )
)
```

in which we were able to avoid explicitly accounting for batch dimensions when indexing and slicing computed quantities in the third line. Note: auto-vectorization is still experimental and some TensorFlow ops may be unsupported. It can be disabled by setting use_vectorized_map=FALSE.

Alternative batch semantics This class also provides alternative semantics for specifying a batch of independent (non-identical) joint distributions. Instead of simply summing the log_probs of component distributions (which may have different shapes), it first reduces the component log_probs to
ensure that \( \log_{\text{prob}}(\text{sample}) \) always returns a scalar, unless \( \text{batch_ndims} \) is explicitly set to a nonzero value (in which case the result will have the corresponding tensor rank).

The essential changes are:

- An event of JointDistributionSequentialAutoBatched is the list of tensors produced by \( \text{sample}() \); thus, the event shape is the list containing the shapes of sampled tensors. These combine both the event and batch dimensions of the component distributions. By contrast, the event shape of a base JointDistributions does not include batch dimensions of component distributions.

- The batch shape is a global property of the entire model, rather than a per-component property as in base JointDistributions. The global batch shape must be a prefix of the batch shapes of each component; the length of this prefix is specified by an optional argument \( \text{batch_ndims} \). If \( \text{batch_ndims} \) is not specified, the model has batch shape \( () \).

Value

a distribution instance.

See Also

For usage examples see e.g. \( \text{tfd_sample}, \text{tfd_log_prob}, \text{tfd_mean} \).

Other distributions: \( \text{tfd_autoregressive}, \text{tfd_batch_reshape}, \text{tfd_bates}, \text{tfd_bernoulli}, \text{tfd_beta_binomial}, \text{tfd_beta}, \text{tfd_binomial}, \text{tfd_categorical}, \text{tfd_cauchy}, \text{tfd_chiz}, \text{tfd_cholesky_lkj}, \text{tfd_continuous_bernoulli}, \text{tfd_deterministic}, \text{tfd_dirichlet_multinomial}, \text{tfd_dirichlet}, \text{tfd_empirical}, \text{tfd_exp_gamma}, \text{tfd_exp_inverse_gamma}, \text{tfd_exponential}, \text{tfd_gaussian_process}, \text{tfd gaussian_process_regression_model}, \text{tfd_generalized_normal}, \text{tfd_geometric}, \text{tfd_half_cauchy}, \text{tfd_half_normal}, \text{tfd_hidden_markov_model}, \text{tfd_horseshoe}, \text{tfd Independent}, \text{tfd_inverse_gaussian}, \text{tfd_johnson_s_u}, \text{tfd_joint_distribution_named_auto_batched}, \text{tfd_joint_distribution_named}, \text{tfd joint_distribution_sequential}, \text{tfd_kumaraswamy}, \text{tfd_laplace}, \text{tfd_linear_gaussian_state_space_model}, \text{tfd_lkj}, \text{tfd_logistic}, \text{tfd_log_normal}, \text{tfd_logistic}, \text{tfd_mixture_same_family}, \text{tfd_mixture}, \text{tfd_multinomial}, \text{tfd_multivariate_normal_diag plus_low_rank}, \text{tfd_multivariate_normal_diag}, \text{tfd_multivariate_normal_linear_operator}, \text{tfd_multivariate_normal_tri_l}, \text{tfd_multivariate_student_t}, \text{tfd negative_binomial}, \text{tfd_normal}, \text{tfd one_hot_categorical}, \text{tfd_pareto}, \text{tfd_pixel_cnn}, \text{tfd poisson log_normal quadrature compound}, \text{tfd poisson}, \text{tfd power_spherical}, \text{tfd probit_bernoulli}, \text{tfd quantized}, \text{tfd relaxed_bernoulli}, \text{tfd relaxed_one hot_categorical}, \text{tfd sample distribution}, \text{tfd sinh_arcsinh}, \text{tfd skellam}, \text{tfd spherical_uniform}, \text{tfd student_t_process}, \text{tfd student_t}, \text{tfd transformed_distribution}, \text{tfd triangular}, \text{tfd truncated_cauchy}, \text{tfd truncated_normal}, \text{tfd uniform}, \text{tfd variational_gaussian_process}, \text{tfd vector_diffeomixture}, \text{tfd vector_exponential_diag}, \text{tfd vector_exponential_linear_operator}, \text{tfd vector_laplace_diag}, \text{tfd vector_laplace_linear_operator}, \text{tfd vector sinh_arcsinh_diag}, \text{tfd von mises_fisher}, \text{tfd von mises}, \text{tfd weibull}, \text{tfd wishart_linear_operator}, \text{tfd wishart}, \text{tfd wishart}, \text{tfd_zipf} \).
tfd_kl_divergence

Computes the Kullback–Leibler divergence.

Description

Denote this distribution by p and the other distribution by q. Assuming p, q are absolutely continuous with respect to reference measure r, the KL divergence is defined as: KL[p, q] = E_p[log(p(X)/q(X))] = - \int_F p(x) \log q(x) \, dr(x) + \int_F p(x) \log p(x) \, dr(x) = H[p, q] - H[p] where F denotes the support of the random variable X ~ p, H[.,.] denotes (Shannon) cross entropy, and H[.] denotes (Shannon) entropy.

Usage

tfd_kl_divergence(distribution, other, name = "kl_divergence")

Arguments

distribution The distribution being used.
other tfp$distributions$Distribution instance.
name String prepended to names of ops created by this function.

Value

self$dtype Tensor with shape [B1, ..., Bn] representing n different calculations of the Kullback-Leibler divergence.

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

d1 <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d2 <- tfd_normal(loc = c(1.5, 2), scale = c(1, 0.5))
d1 %>% tfd_kl_divergence(d2)
**Description**

The Kumaraswamy distribution is defined over the (0, 1) interval using parameters \( \text{concentration1} \) (aka "alpha") and \( \text{concentration0} \) (aka "beta"). It has a shape similar to the Beta distribution, but is easier to reparameterize.

**Usage**

```python
tfd_kumaraswamy(
    concentration1 = 1,
    concentration0 = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Kumaraswamy"
)
```

**Arguments**

- **concentration1**: Positive floating-point Tensor indicating mean number of successes; aka "alpha". Implies self$dtype and self$batch_shape, i.e., concentration1$shape = [N1, N2, ..., Nm] = self$batch_shape.
- **concentration0**: Positive floating-point Tensor indicating mean number of failures; aka "beta". Otherwise has same semantics as concentration1.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.
- **name**: name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The probability density function (pdf) is,

\[
pdf(x; \alpha, \beta) = \alpha \cdot \beta \cdot x^{(\alpha - 1)} \cdot (1 - x^\alpha)^{\beta - 1}
\]

where:

- \( \text{concentration1} = \alpha \),
- \( \text{concentration0} = \beta \). Distribution parameters are automatically broadcast in all functions.
**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_laplace**

*Laplace distribution with location \( \text{loc} \) and scale parameters*  

**Description**

Mathematical details

**Usage**

```python
tfd_laplace(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Laplace"
)
```
Arguments

loc: Floating point tensor which characterizes the location (center) of the distribution.
scale: Positive floating point tensor which characterizes the spread of the distribution.
validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name: name prefixed to Ops created by this class.

Details

The probability density function (pdf) of this distribution is,

\[
\text{pdf}(x; \mu, \sigma) = \frac{\exp(-|x - \mu| / \sigma)}{Z}
\]

\[Z = 2 \sigma\]

where \(\text{loc} = \mu\), \(\text{scale} = \sigma\), and \(Z\) is the normalization constant.

Note that the Laplace distribution can be thought of two exponential distributions spliced together “back-to-back.” The Laplace distribution is a member of the location-scale family, i.e., it can be constructed as,

\[X \sim \text{Laplace}(\text{loc}=0, \text{scale}=1)\]
\[Y = \text{loc} + \text{scale} \times X\]

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chisq()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_deterministic()}, \texttt{tfd_empirical()}, \texttt{tfd_exp_gamma()}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}.
tfd_linear_gaussian_state_space_model

Observation distribution from a linear Gaussian state space model

Description
The state space model, sometimes called a Kalman filter, posits a latent state vector $z_t$ of dimension `latent_size` that evolves over time following linear Gaussian transitions, $z_{t+1} = F * z_t + N(b; Q)$ for transition matrix $F$, bias $b$ and covariance matrix $Q$. At each timestep, we observe a noisy projection of the latent state $x_t = H * z_t + N(c; R)$. The transition and observation models may be fixed or may vary between timesteps.

Usage
```r
  tfd_linear_gaussian_state_space_model(
    num_timesteps,
    transition_matrix,
    transition_noise,
    observation_matrix,
    observation_noise,
    initial_state_prior,
    initial_step = 0L,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "LinearGaussianStateSpaceModel"
  )
```

Arguments
- `num_timesteps` Integer Tensor total number of timesteps.
- `transition_matrix` A transition operator, represented by a Tensor or LinearOperator of shape `[latent_size, latent_size]`, or by a callable taking as argument a scalar integer Tensor `t` and returning a Tensor or LinearOperator representing the transition operator from latent state at time `t` to time `t + 1`. 
- `transition_noise`
- `observation_matrix`
- `observation_noise`
- `initial_state_prior`
- `initial_step`
- `validate_args`
- `allow_nan_stats`
- `name`
transition_noise
An instance of `tfd.MultivariateNormalLinearOperator` with event shape `[latent_size]`, representing the mean and covariance of the transition noise model, or a callable taking as argument a scalar integer Tensor `t` and returning such a distribution representing the noise in the transition from time `t` to time `t + 1`.

observation_matrix
An observation operator, represented by a Tensor or LinearOperator of shape `[observation_size, latent_size]`, or by a callable taking as argument a scalar integer Tensor `t` and returning a timestep-specific Tensor or LinearOperator.

observation_noise
An instance of `tfd.MultivariateNormalLinearOperator` with event shape `[observation_size]`, representing the mean and covariance of the observation noise model, or a callable taking as argument a scalar integer Tensor `t` and returning a timestep-specific noise model.

initial_state_prior
An instance of `MultivariateNormalLinearOperator` representing the prior distribution on latent states; must have event shape `[latent_size]`.

initial_step
optional integer specifying the time of the first modeled timestep. This is added as an offset when passing timesteps `t` to (optional) callables specifying timestep-specific transition and observation models.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
This Distribution represents the marginal distribution on observations, `p(x)`. The marginal `log_prob` is computed by Kalman filtering, and `sample` by an efficient forward recursion. Both operations require time linear in `T`, the total number of timesteps.

Shapes
The event shape is `[num_timesteps, observation_size]`, where `observation_size` is the dimension of each observation `x_t`. The observation and transition models must return consistent shapes. This implementation supports vectorized computation over a batch of models. All of the parameters (prior distribution, transition and observation operators and noise models) must have a consistent batch shape.

Time-varying processes
Any of the model-defining parameters (prior distribution, transition and observation operators and noise models) may be specified as a callable taking an integer timestep `t` and returning a time-dependent value. The dimensionality (latent_size and observation_size) must be the same at all timesteps.
Importantly, the timestep is passed as a Tensor, not a Python integer, so any conditional behavior must occur \textit{inside} the TensorFlow graph. For example, suppose we want to use a different transition model on even days than odd days. It does \textit{not} work to write

```r
transition_matrix <- function(t) {
  if(t %% 2 == 0) even_day_matrix else odd_day_matrix
}
```

since the value of \( t \) is not fixed at graph-construction time. Instead we need to write

```r
transition_matrix <- function(t) {
  tf$cond(tf$equal(tf$mod(t, 2), 0), function() even_day_matrix, function() odd_day_matrix)
}
```

so that TensorFlow can switch between operators appropriately at runtime.

\textbf{Value}

a distribution instance.

\textbf{See Also}

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exp()}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_generalized_normal()}, \texttt{tfd_geometric()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}.

\texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_lkj()}, \texttt{tfd_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_diag()}, \texttt{tfd_multivariate_normal()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_normal_tri_l()}, \texttt{tfd_multivariate_student_t()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_spherical()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_related_bernoulli()}, \texttt{tfd_related_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_spherical_uniform()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_transformed_distribution()}, \texttt{tfd_triangular()}, \texttt{tfd_truncated_cauchy()}, \texttt{tfd_truncated_normal()}, \texttt{tfd_uniform()}, \texttt{tfd_variational_gaussian_process()}, \texttt{tfd_vector_diffeomixture()}, \texttt{tfd_vector_exponential_diag()}, \texttt{tfd_vector_exponential_linear_operator()}, \texttt{tfd_vector_laplace_diag()}, \texttt{tfd_vector_laplace_linear_operator()}, \texttt{tfd_vector_sinh_arcsinh_diag()}, \texttt{tfd_von_mises_fisher()}, \texttt{tfd_von_mises()}, \texttt{tfd_weibull()}, \texttt{tfd_wishart_linear_operator()}, \texttt{tfd_wishart_tri_l()}, \texttt{tfd_wishart()}, \texttt{tfd_zipf()}

\texttt{tfd_linear_gaussian_state_space_model}
tfd_lkj  

**LKJ distribution on correlation matrices**

**Description**

This is a one-parameter of distributions on correlation matrices. The probability density is proportional to the determinant raised to the power of the parameter: \( \text{pdf}(X; \eta) = Z(\eta) \times \det(X)^{\eta - 1} \), where \( Z(\eta) \) is a normalization constant. The uniform distribution on correlation matrices is the special case \( \eta = 1 \).

**Usage**

```r
  tfd_lkj(
    dimension,  
    concentration,  
    input_output_cholesky = FALSE,  
    validate_args = FALSE,  
    allow_nan_stats = TRUE,  
    name = "LKJ"
  )
```

**Arguments**

- **dimension** integer. The dimension of the correlation matrices to sample.
- **concentration** float or double Tensor. The positive concentration parameter of the LKJ distributions. The pdf of a sample matrix \( X \) is proportional to \( \det(X)^{\eta - 1} \).
- **input_output_cholesky** Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample is of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution. Additionally, validation checks which are only defined on the multiplied-out form are omitted, even if validate_args is TRUE. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats** Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name** name prefixed to Ops created by this class.
Details

The distribution is named after Lewandowski, Kurowicka, and Joe, who gave a sampler for the distribution in Lewandowski, Kurowicka, Joe, 2009.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_logistic

Usage
tfd_logistic(
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Logistic"
)

Arguments

loc         Floating point tensor, the means of the distribution(s).
scale       Floating point tensor, the scales of the distribution(s). Must contain only positive values.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name         name prefixed to Ops created by this class.

Details

The cumulative density function of this distribution is:

cdf(x; mu, sigma) = 1 / (1 + exp(-(x - mu) / sigma))

where loc = mu and scale = sigma.
The Logistic distribution is a member of the location-scale family, i.e., it can be constructed as,

X ~ Logistic(loc=0, scale=1)
Y = loc + scale * X

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
tfd_logit_normal() The Logit-Normal distribution

Description
The Logit-Normal distribution models positive-valued random variables whose logit (i.e., sigmoid\_inverse, i.e., $\log(p) - \log(1-p)$) is normally distributed with mean loc and standard deviation scale. It is constructed as the sigmoid transformation, (i.e., $1 / (1 + \exp(-x))$) of a Normal distribution.

Usage
```
tfd_logit_normal(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "LogitNormal"
)
```

Arguments
- **loc** Floating point tensor; the means of the distribution(s).
- **scale** floating point tensor; the stddevs of the distribution(s). Must contain only positive values.
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Description
Given random variable X, the cumulative distribution function cdf is: \( \text{tfd_log_cdf}(x) := \log[P[X \leq x]] \) Often, a numerical approximation can be used for \( \text{tfd_log_cdf}(x) \) that yields a more accurate answer than simply taking the logarithm of the cdf when \( x \ll -1 \).

Usage
tfd_log_cdf(distribution, value, ...)

Arguments
distribution The distribution being used.
value float or double Tensor.
... Additional parameters passed to Python.

Value
a Tensor of shape \( \text{sample_shape}(x) + \text{self$batch_shape} \) with values of type self$dtype.

See Also
Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(),
tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_log_cdf(x)
The LogLogistic distribution models positive-valued random variables whose logarithm is a logistic distribution with `loc` and scale `scale`. It is constructed as the exponential transformation of a Logistic distribution.

**Usage**

```python
def tfd_log_logistic(
    loc,  # Floating-point Tensor; the loc of the underlying logistic distribution(s).
scale,  # Floating-point Tensor; the scale of the underlying logistic distribution(s).
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "LogLogistic"
)
```

**Arguments**

- `loc`  
  - Floating-point Tensor; the loc of the underlying logistic distribution(s).
- `scale`  
  - Floating-point Tensor; the scale of the underlying logistic distribution(s).
- `validate_args`  
  - Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats`  
  - Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name`  
  - name prefixed to Ops created by this class.

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`,...
The LogNormal distribution models positive-valued random variables whose logarithm is normally distributed with mean `loc` and standard deviation `scale`. It is constructed as the exponential transformation of a Normal distribution.

The LogNormal distribution models positive-valued random variables whose logarithm is normally distributed with mean `loc` and standard deviation `scale`. It is constructed as the exponential transformation of a Normal distribution.

**Usage**

```r
# Example usage

tfd_log_normal(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "LogNormal"
)

tfd_log_normal(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "LogNormal"
)
```
Arguments

loc Floating-point Tensor; the means of the underlying Normal distribution(s).

scale Floating-point Tensor; the stddevs of the underlying Normal distribution(s).

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

tfd_log_prob

Log probability density/mass function.

Description

Log probability density/mass function.

Usage

tfd_log_prob(distribution, value, ...)

Arguments

distribution  The distribution being used.
value  float or double Tensor.
...  Additional parameters passed to Python.

Value

a Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_cdf(), tfd_log_survival_function(), tfd_mean(), tfd_mode(),
tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  x <- d %>% tfd_sample()
  d %>% tfd_log_prob(x)
```

tfd_log_survival_function

Log survival function.

Description

Given random variable X, the survival function is defined: tfd_log_survival_function(x) =
Log[ P[X > x] ] = Log[ 1 -P[X <= x] ] = Log[ 1 -cdf(x) ]
tfd_mean

Usage

tfd_log_survival_function(distribution, value, ...)

Arguments

distribution The distribution being used.
value float or double Tensor.
... Additional parameters passed to Python.

Details

Typically, different numerical approximations can be used for the log survival function, which are more accurate than 1 - cdf(x) when x » 1.

Value

a Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.

See Also

Other distribution_methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(), tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  x <- d %>% tfd_sample()
  d %>% tfd_log_survival_function(x)
```

---

**tfd_mean**

*Mean.*

Description

Mean.

Usage

tfd_mean(distribution, ...)

Arguments

distribution The distribution being used.
... Additional parameters passed to Python.
Value

A Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mode(),
tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_mean()

tfd_mixture

Description

The Mixture object implements batched mixture distributions. The mixture model is defined by a
Categorical distribution (the mixture) and a list of Distribution objects.

Usage

tfd_mixture(
  cat,
  components,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  use_static_graph = FALSE,
  name = "Mixture"
)

Arguments

cat A Categorical distribution instance, representing the probabilities of distributions.
components A list or tuple of Distribution instances. Each instance must have the same
type, be defined on the same domain, and have matching event_shape and
batch_shape.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.
tfd_mixture_same_family

use_static_graph

Calls to sample will not rely on dynamic tensor indexing, allowing for some static graph compilation optimizations, but at the expense of sampling all underlying distributions in the mixture. (Possibly useful when running on TPUs). Default value: FALSE (i.e., use dynamic indexing).

name

name prefixed to Ops created by this class.

Details

Methods supported include tfd_log_prob, tfd_prob, tfd_mean, tfd_sample, and entropy_lower_bound.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(), tfd_dirichlet_multinomial(), tfd_dirichlet_multivariate(), tfd_dirichlet_multivariate_labeled(), tfd_dirichlet_labeled(), tfd_dirichlet_multinomial_labeled(), tfd_dirichlet_labeled(), tfd_dirichlet_multinomial_labeled(), tfd_dirichlet_labeled(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gaussian_process(), tfd_gaussian_process_regression_model(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfdVariationalGaussianProcess(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
Description

The `MixtureSameFamily` distribution implements a (batch of) mixture distribution where all components are from different parameterizations of the same distribution type. It is parameterized by a `Categorical` "selecting distribution" (over k components) and a components distribution, i.e., a `Distribution` with a rightmost batch shape (equal to [k]) which indexes each (batch of) component.

Usage

```r
fn_mixture_same_family(
  mixture_distribution,
  components_distribution,
  reparameterize = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "MixtureSameFamily"
)
```

Arguments

- **mixture_distribution**: `tfp$distributions$Categorical-like instance`. Manages the probability of selecting components. The number of categories must match the rightmost batch dimension of the `components_distribution`. Must have either scalar `batch_shape` or `batch_shape` matching `components_distribution$batch_shape[:-1]`.

- **components_distribution**: `tfp$distributions$Distribution-like instance`. Right-most batch dimension indexes components.

- **reparameterize**: Logical, default `FALSE`. Whether to reparameterize samples of the distribution using implicit reparameterization gradients (Figurnov et al., 2018). The gradients for the mixture logits are equivalent to the ones described by (Graves, 2016). The gradients for the components parameters are also computed using implicit reparameterization (as opposed to ancestral sampling), meaning that all components are updated every step. Only works when: (1) `components_distribution` is fully reparameterized; (2) `components_distribution` is either a scalar distribution or fully factorized (`tfd.Independent` applied to a scalar distribution); (3) batch shape has a known rank. Experimental, may be slow and produce infs/NaNs.

- **validate_args**: Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.

- **allow_nan_stats**: Logical, default `TRUE`. When `TRUE`, statistics (e.g., mean, mode, variance) use the value `NaN` to indicate the result is undefined. When `FALSE`, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.
**tfd_mode**

**Value**

a distribution instance.

**References**


**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


---

**tfd_mode**

**Mode.**

**Description**

Mode.

**Usage**

```
tfd_mode(distribution, ...)```
**tfd_multinomial**

**Arguments**

- **distribution**: The distribution being used.
- **...**: Additional parameters passed to Python.

**Value**

A Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

**See Also**

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`.

**Examples**

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  d %>% tfd_mode()
```

**Description**

This Multinomial distribution is parameterized by `probs`, a (batch of) length-K prob (probability) vectors (`K > 1`) such that `tf.reduce_sum(probs,-1) = 1`, and a `total_count` number of trials, i.e., the number of trials per draw from the Multinomial. It is defined over a (batch of) length-K vector `counts` such that `tf$reduce_sum(counts,-1) = total_count`. The Multinomial is identically the Binomial distribution when `K = 2`.

**Usage**

```r
tfd_multinomial(
  total_count,
  logits = NULL,
  probs = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Multinomial"
)
```
**Arguments**

- **total_count**: Non-negative floating point tensor with shape broadcastable to \([N1, ..., Nm]\) with \(m \geq 0\). Defines this as a batch of \(N1 \times ... \times Nm\) different Multinomial distributions. Its components should be equal to integer values.

- **logits**: Floating point tensor representing unnormalized log-probabilities of a positive event with shape broadcastable to \([N1, ..., Nm, K]\) \(m \geq 0\), and the same dtype as total_count. Defines this as a batch of \(N1 \times ... \times Nm\) different \(K\) class Multinomial distributions. Only one of logits or probs should be passed in.

- **probs**: Positive floating point tensor with shape broadcastable to \([N1, ..., Nm, K]\) \(m \geq 0\) and same dtype as total_count. Defines this as a batch of \(N1 \times ... \times Nm\) different \(K\) class Multinomial distributions. Only one of logits or probs should be passed in.

- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**: name prefixed to Ops created by this class.

**Details**

**Mathematical Details**

The Multinomial is a distribution over \(K\)-class counts, i.e., a length-\(K\) vector of non-negative integer counts \(n = [n_0, ..., n_{(K-1)}]\). The probability mass function (pmf) is,

\[
\text{pmf}(n; \pi, N) = \prod_j (\pi_j)^{n_j} / Z \\
Z = (\prod_j n_j!) / N!
\]

where:

- \(\pi = [\pi_0, ..., \pi_{(K-1)}]\), \(\pi_j > 0\), sum_j \(\pi_j = 1\),
- total_count = \(N\), \(N\) a positive integer,
- \(Z\) is the normalization constant, and,
- \(N!\) denotes \(N\) factorial.

Distribution parameters are automatically broadcast in all functions; see examples for details.

**Pitfalls**

The number of classes, \(K\), must not exceed:

- the largest integer representable by self$\text{dtype}$, i.e., \(2^{(\text{mantissa_bits}+1)}\) (IEEE754),
- the maximum Tensor index, i.e., \(2^{31}-1\).

Note: This condition is validated only when validate_args = TRUE.
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd الصينى()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negation()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart()`, `tfd_zipf()`

tfd_multivariate_normal_diag

Multivariate normal distribution on \( \mathbb{R}^k \)

Description

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length- \( k \) loc vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; covariance = scale @ scale. \( \text{Twhere} @ \) denotes matrix-multiplication.

Usage

tfd_multivariate_normal_diag(  
  loc = NULL, 
  scale_diag = NULL, 
  scale_identity_multiplier = NULL,
validate_args = FALSE,
allow_nan_stats = TRUE,
name = "MultivariateNormalDiag"
)

Arguments

loc Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.

scale_diag Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape [B1, ..., Bb, k], b >= 0, and characterizes b-batches of k x k diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

scale_identity_multiplier Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape [B1, ..., Bb], b >= 0, and characterizes b-batches of scaled k x k identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[pdf(x; \text{loc}, \text{scale}) = \exp(-0.5 \ ||y||^2) / Z\]

\[y = \text{inv}(\text{scale}) @ (x - \text{loc})\]

\[Z = (2 \pi)^{0.5 k} |\text{det}(\text{scale})|\]

where:

- \text{loc} is a vector in \(\mathbb{R}^k\),
- \text{scale} is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.T\),
- \(Z\) denotes the normalization constant, and,
- \(||y||^2\) denotes the squared Euclidean norm of \(y\).

A (non-batch) scale matrix is:

\[\text{scale} = \text{diag}(\text{scale}_\text{diag} + \text{scale}_\text{identity}_\text{multiplier} \times \text{ones}(k))\]

where:
• scale_diag.shape = [k], and,
• scale_identity_multiplier.shape = [].

Additional leading dimensions (if any) will index batches.

If both scale_diag and scale_identity_multiplier are NULL, then scale is the Identity matrix. The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \] \hspace{1cm} \# Identity scale, zero shift.

\[ Y = \text{scale} @ X + \text{loc} \]

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chisquare()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_empirical()}, \texttt{tfd_exp_gamma()}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_generalized_normal()}, \texttt{tfd_geometric()}, \texttt{tfd_global_scale()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()} \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixuture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_normal_tri_l()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_uniform()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_uniform()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_uniform()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_uniform()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_poisson()}, \texttt{tfd_power_uniform()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaed_bernoulli()}, \texttt{tfd_relaed_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_skellam()}, \texttt{tfd_spherical_uniform()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_transformed_distribution()}, \texttt{tfd_triangular()}, \texttt{tfd_truncated_cauchy()}, \texttt{tfd_truncated_normal()}, \texttt{tfd_uniform()}, \texttt{tfd_vairational_gaussian_process()}, \texttt{tfd_vector_diffemixture()}, \texttt{tfd_vector_exponential_diag()}, \texttt{tfd_vector_exponential_linear_operator()}, \texttt{tfd_vector_laplace_diag()}, \texttt{tfd_vector_laplace_linear_operator()}, \texttt{tfd_vector_sinh_arcsinh_diag()}, \texttt{tfd_von_mises_fisher()}, \texttt{tfd_von_mises()}, \texttt{tfd_weibull()}, \texttt{tfd_wishart_linear_operator()}, \texttt{tfd_wishart_tri_l()}, \texttt{tfd_wishart()}, \texttt{tfd_zipf()}.
Description

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length-\( k \) loc vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; covariance = scale \( \odot \) scale. Where \( \odot \) denotes matrix-multiplication.

Usage

tfd_multivariate_normal_diag_plus_low_rank(
  loc = NULL,
  scale_diag = NULL,
  scale_identity_multiplier = NULL,
  scale_perturb_factor = NULL,
  scale_perturb_diag = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "MultivariateNormalDiagPlusLowRank"
)

Arguments

loc Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, ..., Bb, k]\) where \( b \geq 0 \) and \( k \) is the event size.

scale_diag Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape \([B1, ..., Bb, k]\), \( b \geq 0 \), and characterizes \( b \)-batches of \( k \times k \) diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

scale_identity_multiplier Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape \([B1, ..., Bb, k]\), \( b \geq 0 \), and characterizes \( b \)-batches of \( k \times k \) identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

scale_perturb_factor Floating-point Tensor representing a rank-\( r \) perturbation added to scale. May have shape \([B1, ..., Bb, k, r]\), \( b \geq 0 \), and characterizes \( b \)-batches of rank-\( r \) updates to scale. When NULL, no rank-\( r \) update is added to scale.#'

scale_perturb_diag Floating-point Tensor representing a diagonal matrix inside the rank-\( r \) perturbation added to scale. May have shape \([B1, ..., Bb, r]\), \( b \geq 0 \), and characterizes \( b \)-batches of \( r \times r \) diagonal matrices inside the perturbation added to scale. When NULL, an identity matrix is used inside the perturbation. Can only be specified if scale_perturb_factor is also specified.
validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[
pdf(x; \text{loc, scale}) = \exp(-0.5 ||y||^2) / Z
\]

\[
y = \text{inv}(scale) @ (x - \text{loc})
\]

\[
Z = (2 \pi)^{(0.5 k)} |\text{det}(scale)|
\]

where:

- \(\text{loc}\) is a vector in \(\mathbb{R}^k\),
- \(\text{scale}\) is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.T\),
- \(Z\) denotes the normalization constant, and,
- ||y||^2 denotes the squared Euclidean norm of \(y\).

A (non-batch) scale matrix is:

\[
\text{scale} = \text{diag}(\text{scale\_diag} + \text{scale\_identity\_multiplier\ one}(k)) + \\
\text{scale\_perturb\_factor} @ \text{diag}(\text{scale\_perturb\_diag}) @ \text{scale\_perturb\_factor}.T
\]

where:

- \(\text{scale\_diag\.shape} = [k]\),
- \(\text{scale\_identity\_multiplier\.shape} = []\),
- \(\text{scale\_perturb\_factor\.shape} = [k, r]\), typically \(k >> r\), and,
- \(\text{scale\_perturb\_diag\.shape} = [r]\).

Additional leading dimensions (if any) will index batches. If both \(\text{scale\_diag}\) and \(\text{scale\_identity\_multiplier}\)
are NULL, then \(\text{scale}\) is the Identity matrix. The MultivariateNormal distribution is a member of the
\text{location-scale family}, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=I) \quad \# \text{Identity scale, zero shift.}
\]

\[
Y = \text{scale} @ X + \text{loc}
\]

Value
a distribution instance.
See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial,
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gaussian(),
tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space,
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t(),
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(),
tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_power_spherical(),
tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(),
tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(),
tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(),
tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(),
tfd_vector_diffemixture(), tfd_vector_exponential_diagonal(), tfd_vector_exponential_linear_operator(),
tfd_vector_laplace_diagonal(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diagonal(),
tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(),
tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
Arguments

loc  Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, ..., Bb, k]\) where \(b >= 0\) and \(k\) is the event size.

covariance_matrix  Floating-point, symmetric positive definite Tensor of same dtype as loc. The strict upper triangle of covariance_matrix is ignored, so if covariance_matrix is not symmetric no error will be raised (unless validate_args is TRUE). covariance_matrix has shape \([B1, ..., Bb, k, k]\) where \(b >= 0\) and \(k\) is the event size.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name  name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
\begin{align*}
pdf(x; \text{loc}, \text{scale}) &= \exp(-0.5 \ ||y||^2) / Z \\
y &= \text{inv(scale)} @ (x - \text{loc}) \\
Z &= (2 \pi)^{0.5 k} |\text{det}(\text{scale})|
\end{align*}
\]

where:

- \(\text{loc}\) is a vector in \(\mathbb{R}^k\),
- \(\text{scale}\) is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.T\),
- \(Z\) denotes the normalization constant, and,
- \(||y||^2\) denotes the squared Euclidean norm of \(y\).

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
\begin{align*}
X &\sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad # \text{Identity scale, zero shift.} \\
Y &= \text{scale} @ X + \text{loc}
\end{align*}
\]

The batch_shape is the broadcast shape between loc and covariance_matrix arguments. The event_shape is given by last dimension of the matrix implied by covariance_matrix. The last dimension of loc (if provided) must broadcast with this. A non-batch covariance_matrix matrix is a \(k \times k\) symmetric positive definite matrix. In other words it is (real) symmetric with all eigenvalues strictly positive. Additional leading dimensions (if any) will index batches.

Value

a distribution instance.
See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson(), tfd_poisson_log_normal_quadrature_compound(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

tfd_multivariate_normal_linear_operator

The multivariate normal distribution on \( \mathbb{R}^k \)

Description

The Multivariate Normal distribution is defined over \( \mathbb{R}^k \) and parameterized by a (batch of) length- \( k \) loc vector (aka "mu") and a (batch of) \( k \times k \) scale matrix; covariance = scale @ scale. \( @ \) denotes matrix-multiplication.

Usage

tfd_multivariate_normal_linear_operator(
  loc = NULL,
  scale = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "MultivariateNormalLinearOperator"
)
Arguments

- **loc**: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, ..., Bb, k]\) where \(b \geq 0\) and \(k\) is the event size.
- **scale**: Instance of LinearOperator with same dtype as loc and shape \([B1, ..., Bb, k, k]\).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 \ ||y||^2) / Z
\]

\[
y = \text{inv}(\text{scale}) @ (x - \text{loc})
\]

\[
Z = (2 \pi)^{(0.5 \ k)} \ |\text{det}(\text{scale})|
\]

where:

- \(\text{loc}\) is a vector in \(\mathbb{R}^k\),
- \(\text{scale}\) is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}^T\),
- \(Z\) denotes the normalization constant, and,
- \(||y||^2\) denotes the squared Euclidean norm of \(y\).

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift}.
\]

\[
Y = \text{scale} @ X + \text{loc}
\]

The batch_shape is the broadcast shape between loc and scale arguments. The event_shape is given by last dimension of the matrix implied by scale. The last dimension of loc (if provided) must broadcast with this. Recall that covariance = scale @ scale.T. Additional leading dimensions (if any) will index batches.

Value

a distribution instance.
See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`. Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture()`, `tfd_mixture_same_family()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises()`, `tfd_von_mises_fisher()`, `tfd_wishart_tril()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`, `tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_spherical_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart()`, `tfd_zipf()`

tfd_multivariate_normal_tril

The multivariate normal distribution on R^k

Description

The Multivariate Normal distribution is defined over R^k and parameterized by a (batch of) length-k loc vector (aka "mu") and a (batch of) k x k scale matrix; covariance = scale @ scale. Where @ denotes matrix-multiplication.

Usage

tfd_multivariate_normal_tril(  
  loc = NULL,  
  scale_tril = NULL,  
  validate_args = FALSE,  
  allow_nan_stats = TRUE,  
  name = "MultivariateNormalTriL"
)
Arguments

loc
Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.

scale_tril
Floating-point, lower-triangular Tensor with non-zero diagonal elements. scale_tril has shape [B1, ..., Bb, k, k] where b >= 0 and k is the event size.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

\[
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-0.5 \|y\|^2) / Z
\]
\[
y = \text{inv(scale)} @ (x - \text{loc})
\]
\[
Z = (2 \pi)^{0.5 k} |\det(\text{scale})|
\]

where:

- loc is a vector in R^k,
- scale is a linear operator in R^[k x k], \( \text{cov} = \text{scale} @ \text{scale}.T \),
- Z denotes the normalization constant, and,
- \( \|y\|^2 \) denotes the squared Euclidean norm of y.

A (non-batch) scale matrix is:

\[
\text{scale} = \text{scale_tril}
\]

where scale_tril is lower-triangular k x k matrix with non-zero diagonal, i.e., \( \text{tf$diag_part(scale_tril)} \neq 0 \). Additional leading dimensions (if any) will index batches.

The MultivariateNormal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateNormal}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.}
\]
\[
Y = \text{scale} @ X + \text{loc}
\]

Value

a distribution instance.
tfd_multivariate_student_t_linear_operator

Multivariate Student's t-distribution on \( \mathbb{R}^k \)

Description

Mathematical Details

Usage

tfd_multivariate_student_t_linear_operator(
    df,
    loc,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "MultivariateStudentTLinearOperator"
)
tfd_multivariate_student_t_linear_operator

Arguments

- **df**: A positive floating-point Tensor. Has shape \([B_1, ..., B_b]\) where \(b >= 0\).
- **loc**: Floating-point Tensor. Has shape \([B_1, ..., B_b, k]\) where \(k\) is the event size.
- **scale**: Instance of LinearOperator with a floating dtype and shape \([B_1, ..., B_b, k, k]\).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

The probability density function (pdf) is,

\[
pdf(x; df, loc, Sigma) = \frac{1 + \|y\|^2 / df^{\cdot}(0.5 (df + k))}{Z}
\]

where,

\[
y = inv(Sigma) (x - loc)
\]

\[
Z = abs(det(Sigma)) \sqrt{\text{det}(df \cdot pi^{\cdot}k) \text{Gamma}(0.5 df) / \text{Gamma}(0.5 (df + k))}
\]

where:

- \(df\) is a positive scalar.
- \(loc\) is a vector in \(\mathbb{R}^k\).
- \(Sigma\) is a positive definite shape matrix in \(\mathbb{R}^{k \times k}\), parameterized as \(scale \odot scale^\top\) in this class,
- \(Z\) denotes the normalization constant, and,
- \(\|y\|^2\) denotes the squared Euclidean norm of \(y\).

The Multivariate Student’s t-distribution distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{MultivariateT}(\text{loc}=0, \text{scale}=1) \quad \# \text{Identity scale, zero shift.}
\]

\[
Y = scale \odot X + loc
\]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
`tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`.
tfd_negative_binomial

**Description**

The NegativeBinomial distribution is related to the experiment of performing Bernoulli trials in sequence. Given a Bernoulli trial with probability $p$ of success, the NegativeBinomial distribution represents the distribution over the number of successes $s$ that occur until we observe $f$ failures.

**Usage**

```r
tfd_negative_binomial(
    total_count,
    logits = NULL,
    probs = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "NegativeBinomial"
)
```

**Arguments**

- **total_count**: Non-negative floating-point Tensor with shape broadcastable to $[B_1, ..., B_b]$ with $b >= 0$ and the same dtype as probs or logits. Defines this as a batch of $N_1 x ... x N_m$ different Negative Binomial distributions. In practice, this represents the number of negative Bernoulli trials to stop at (the total_count of failures), but this is still a valid distribution when total_count is a non-integer.
logits Floating-point Tensor with shape broadcastable to [B1, ..., Bb] where b >= 0 indicates the number of batch dimensions. Each entry represents logits for the probability of success for independent Negative Binomial distributions and must be in the open interval (-inf, inf). Only one of logits or probs should be specified.

probs Positive floating-point Tensor with shape broadcastable to [B1, ..., Bb] where b >= 0 indicates the number of batch dimensions. Each entry represents the probability of success for independent Negative Binomial distributions and must be in the open interval (0, 1). Only one of logits or probs should be specified.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details
The probability mass function (pmf) is,

\[
\text{pmf}(s; f, p) = p^s (1 - p)^f / Z
\]
\[
Z = s! (f - 1)! / (s + f - 1)!
\]

where:

- total_count = f,
- probs = p,
- Z is the normalizing constant, and,
- n! is the factorial of n.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
 tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
 tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(),
 tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
 tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
 tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
 tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
 tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
 tfd_kl_divergence(), tfd_lgamma(), tfd_laplace(), tfd_log_normal(), tfd_logistic(),
 tfd_lognormal(), tfd_lgamma(), tfd_levy(), tfd_levy_ksi(), tfd_linear_cong(), tfd_mixture(),
 tfd_multinomial(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_triangular(),
 tfd_negative_binomial(), tfd_normal(), tfd_categorical(),
**tfd_normal**

Normal distribution with loc and scale parameters

**Description**
Mathematical details

**Usage**
```
tfd_normal(
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Normal"
)
```

**Arguments**
- **loc**: Floating point tensor; the means of the distribution(s).
- **scale**: Floating point tensor; the stddevs of the distribution(s). Must contain only positive values.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
Details

The probability density function (pdf) is,

\[
pdf(x; \mu, \sigma) = \exp(-0.5 (x - \mu)^2 / \sigma^2) / Z
\]
\[
Z = (2 \pi \sigma^2)^{0.5}
\]

where \(\text{loc} = \mu\) is the mean, \(\text{scale} = \sigma\) is the std. deviation, and, \(Z\) is the normalization constant. The Normal distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{Normal}(\text{loc}=0, \text{scale}=1)
\]
\[
Y = \text{loc} + \text{scale} \times X
\]

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

OneHotCategorical distribution

Description

The categorical distribution is parameterized by the log-probabilities of a set of classes. The difference between OneHotCategorical and Categorical distributions is that OneHotCategorical is a discrete distribution over one-hot bit vectors whereas Categorical is a discrete distribution over positive integers. OneHotCategorical is equivalent to Categorical except Categorical has event_dim=() while OneHotCategorical has event_dim=K, where K is the number of classes.

Usage

```python
  tfd_one_hot_categorical(
    logits = NULL,
    probs = NULL,
    dtype = tf$int32,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "OneHotCategorical"
  )
```

Arguments

- **logits**
  
  An N-D Tensor, N >= 1, representing the log probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.

- **probs**
  
  An N-D Tensor, N >= 1, representing the probabilities of a set of Categorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.

- **dtype**
  
  The type of the event samples (default: int32).

- **validate_args**
  
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **allow_nan_stats**
  
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

- **name**
  
  name prefixed to Ops created by this class.
This class provides methods to create indexed batches of OneHotCategorical distributions. If the provided logits or probs is rank 2 or higher, for every fixed set of leading dimensions, the last dimension represents one single OneHotCategorical distribution. When calling distribution functions (e.g. dist.prob(x)), logits and x are broadcast to the same shape (if possible). In all cases, the last dimension of logits, x represents single OneHotCategorical distributions.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_log_normal(), tfd_mixtures(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_diffusion_mixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
tfd_pareto

Usage

tfd_pareto(
    concentration,
    scale = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Pareto"
)

Arguments

collection  Floating point tensor. Must contain only positive values.

scale  Floating point tensor, equivalent to mode. scale also restricts the domain of
this distribution to be in [scale, inf). Must contain only positive values. Default
value: 1.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid
inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) is,

pdf(x; alpha, scale, x >= scale) = alpha * scale ** alpha / x ** (alpha + 1)

where ‘concentration = alpha’.

Note that ‘scale’ acts as a scaling parameter, since
‘Pareto(c, scale).pdf(x) == Pareto(c, 1.).pdf(x / scale)’.
The support of the distribution is defined on `[scale, infinity)`.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli().
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_ch12(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial
tfd_pert

**Modified PERT distribution for modeling expert predictions.**

**Description**

The PERT distribution is a loc-scale family of Beta distributions fit onto a real interval between low and high values set by the user, along with a peak to indicate the expert’s most frequent prediction, and temperature to control how sharp the peak is.

**Usage**

```r
  tfd_pert(
    low, peak, high,
    temperature = 4,
    validate_args = FALSE,
    allow_nan_stats = FALSE,
    name = "Pert"
  )
```

**Arguments**

- **low**: lower bound
- **peak**: most frequent value
- **high**: upper bound
temperature  controls the shape of the distribution

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details

The distribution is similar to a Triangular distribution (i.e. tfd.Triangular) but with a smooth peak.

Mathematical Details

In terms of a Beta distribution, PERT can be expressed as

\[
\text{PERT} \sim \text{loc} + \text{scale} \times \text{Beta}(\text{concentration1}, \text{concentration0})
\]

where

\[
\begin{align*}
\text{loc} & = \text{low} \\
\text{scale} & = \text{high} - \text{low} \\
\text{concentration1} & = 1 + \text{temperature} \times (\text{peak} - \text{low})/(\text{high} - \text{low}) \\
\text{concentration0} & = 1 + \text{temperature} \times (\text{high} - \text{peak})/(\text{high} - \text{low}) \\
\text{temperature} & > 0
\end{align*}
\]

The support is [low, high]. The peak must fit in that interval: low < peak < high. The temperature is a positive parameter that controls the shape of the distribution. Higher values yield a sharper peak. The standard PERT distribution is obtained when temperature = 4.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
The Pixel CNN++ distribution

Description

Pixel CNN++ (Salimans et al., 2017) models a distribution over image data, parameterized by a neural network. It builds on Pixel CNN and Conditional Pixel CNN, as originally proposed by (van den Oord et al., 2016). The model expresses the joint distribution over pixels as the product of conditional distributions: $p(x|h) = \text{prod} \{ p(x[i]|x[:i], h) : i=0, ..., d \}$, in which $p(x[i]|x[:i], h) : i=0, ..., d$ is the probability of the $i$-th pixel conditional on the pixels that preceded it in raster order (color channels in RGB order, then left to right, then top to bottom). $h$ is optional additional data on which to condition the image distribution, such as class labels or VAE embeddings. The Pixel CNN++ network enforces the dependency structure among pixels by applying a mask to the kernels of the convolutional layers that ensures that the values for each pixel depend only on other pixels up and to the left. Pixel values are modeled with a mixture of quantized logistic distributions, which can take on a set of distinct integer values (e.g., between 0 and 255 for an 8-bit image). Color intensity $v$ of each pixel is modeled as: $v \sim \text{sum}\{q[i] \ast \text{quantized_logistic}(\text{loc}[i], \text{scale}[i]) : i = 0, ..., k \}$, in which $k$ is the number of mixture components and the $q[i]$ are the Categorical probabilities over the components.

Usage

tfd_pixel_cnn(
  image_shape,
  conditional_shape = NULL,
  num_resnet = 5,
  num_hierarchies = 3,
  num_filters = 160,
  num_logistic_mix = 10,
  receptive_field_dims = c(3, 3),
  dropout_p = 0.5,
  resnet_activation = "concat_elu",
  use_weight_norm = TRUE,
  use_data_init = TRUE,
  high = 255,
  low = 0,
  dtype = tf$float32,
  name = "PixelCNN"
)

Arguments

- **image_shape**: 3D TensorShape or tuple for the [height, width, channels] dimensions of the image.
- **conditional_shape**: TensorShape or tuple for the shape of the conditional input, or NULL if there is no conditional input.
num_resnet  integer, the number of layers (shown in Figure 2 of https://arxiv.org/abs/1606.05328) within each highest-level block of Figure 2 of https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.

num_hierarchies  integer, the number of highest-level blocks (separated by expansions/contractions of dimensions in Figure 2 of https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.)

num_filters  integer, the number of convolutional filters.

num_logistic_mix  integer, number of components in the logistic mixture distribution.

receptive_field_dims  tuple, height and width in pixels of the receptive field of the convolutional layers above and to the left of a given pixel. The width (second element of the tuple) should be odd. Figure 1 (middle) of https://arxiv.org/abs/1606.05328 shows a receptive field of (3, 5) (the row containing the current pixel is included in the height). The default of (3, 3) was used to produce the results in https://pdfs.semanticscholar.org/9e90/6792f67cbdda7b7777b69284a81044857656.pdf.

dropout_p  float, the dropout probability. Should be between 0 and 1.

resnet_activation  string, the type of activation to use in the resnet blocks. May be 'concat_elu', 'elu', or 'relu'.

use_weight_norm  logical, if TRUE then use weight normalization (works only in Eager mode).

use_data_init  logical, if TRUE then use data-dependent initialization (has no effect if use_weight_norm is FALSE).

high  integer, the maximum value of the input data (255 for an 8-bit image).

low  integer, the minimum value of the input data.

dtype  Data type of the Distribution.

name  string, the name of the Distribution.

Value

a distribution instance.

References


See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


tfd_plackett_luce  
Plackett-Luce distribution over permutations.

Description

The Plackett-Luce distribution is defined over permutations of fixed length. It is parameterized by a positive score vector of same length. This class provides methods to create indexed batches of PlackettLuce distributions. If the provided scores is rank 2 or higher, for every fixed set of leading dimensions, the last dimension represents one single PlackettLuce distribution. When calling distribution functions (e.g. `dist.log_prob(x)`), scores and x are broadcast to the same shape (if possible). In all cases, the last dimension of scores, x represents single PlackettLuce distributions.

Usage

tfd_plackett_luce(
  scores,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "PlackettLuce"
)

Arguments

scores An N-D Tensor, N >= 1, representing the scores of a set of elements to be permuted. The first N -1 dimensions index into a batch of independent distributions and the last dimension represents a vector of scores for the elements.
dtype The type of the event samples (default: int32).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical Details

The Plackett-Luce is a distribution over permutation vectors p of length k where the permutation p is an arbitrary ordering of k indices {0, 1, ..., k-1}. The probability mass function (pmf) is,

\[ \text{pmf}(p; s) = \prod_i s_{p_i} / (Z - Z_i) \]

\[ Z = \sum_{j=0}^{k-1} s_j \]

\[ Z_i = \sum_{j=0}^{i-1} s_{p_j} \text{ for } i > 0 \text{ and } 0 \text{ for } i = 0 \]

where scores = s = [s_0, ..., s_{k-1}], s_i >= 0.

Samples from Plackett-Luce distribution are generated sequentially as follows.

Initialize normalization \( N_0 = Z \)

For \( \{0, 1, ..., k-1\} \):

1. Sample i-th element of permutation
   \( p_i \sim \text{Categorical}(\text{probs}=[s_0/N_i, ..., s_{k-1}/N_i]) \)

2. Update normalization
   \( N_{i+1} = N_i - s_{p_i} \)

3. Mask out sampled index for subsequent rounds
   \( s_{p_i} = 0 \)

Return p

Alternately, an equivalent way to sample from this distribution is to sort Gumbel perturbed log-scores (Aditya et al. 2019)

\[ p = \text{argsort}(\log s + g) \sim \text{PlackettLuce}(s) \]

\[ g = [g_0, ..., g_{k-1}], g_i \sim \text{Gumbel}(0, 1) \]

Value

a distribution instance.
tfd_poisson

References


See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

tfd_poisson  Poisson distribution

Description

The Poisson distribution is parameterized by an event rate parameter.

Usage

tfd_poisson(
    rate = NULL,
    log_rate = NULL,
    interpolate_nondiscrete = TRUE,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Poisson"
)

Arguments

rate  Floating point tensor, the rate parameter. rate must be positive. Must specify exactly one of rate and log_rate.

log_rate  Floating point tensor, the log of the rate parameter. Must specify exactly one of rate and log_rate.

interpolate_nondiscrete  Logical. When FALSE, log_prob returns -inf (and prob returns 0) for non-integer inputs. When TRUE, log_prob evaluates the continuous function \( k \times \log_{\text{rate}} - \log\Gamma(k+1) - \text{rate} \), which matches the Poisson pmf at integer arguments \( k \) (note that this function is not itself a normalized probability log-density). Default value: TRUE.

validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic's batch members are undefined.

name  name prefixed to Ops created by this class.
Details

Mathematical Details

The probability mass function (pmf) is,

$$pmf(k; \lambda, k \geq 0) = \frac{\lambda^k}{k!} / Z$$

$$Z = \exp(\lambda).$$

where rate = \lambda and Z is the normalizing constant.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Description

The PoissonLogNormalQuadratureCompound is an approximation to a Poisson-LogNormal compound distribution, i.e.,

\[ p(k | \text{loc}, \text{scale}) = \int_{\mathbb{R}_+} dl \ \text{LogNormal}(l | \text{loc}, \text{scale}) \ \text{Poisson}(k | l) \approx \sum \{ \text{prob}[d] \ \text{Poisson}(k | \lambda(\text{grid}[d])) : d=0, \ldots, \text{deg}-1 \} \]

Usage

tfd_poisson_log_normal_quadrature_compound(
    loc,
    scale,
    quadrature_size = 8,
    quadrature_fn = tfp$distributions$quadrature_scheme_lognormal_quantiles,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "PoissonLogNormalQuadratureCompound"
)

Arguments

loc float-like (batch of) scalar Tensor; the location parameter of the LogNormal prior.

scale float-like (batch of) scalar Tensor; the scale parameter of the LogNormal prior.

quadrature_size integer scalar representing the number of quadrature points.

quadrature_fn Function taking loc, scale, quadrature_size, validate_args and returning tuple(grid,probs) representing the LogNormal grid and corresponding normalized weight. Default value: quadrature_scheme_lognormal_quantiles.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

By default, the grid is chosen as quantiles of the LogNormal distribution parameterized by loc, scale and the prob vector is \([1. / \text{quadrature_size}] \times \text{quadrature_size}\).

In the non-approximation case, a draw from the LogNormal prior represents the Poisson rate parameter. Unfortunately, the non-approximate distribution lacks an analytical probability density function (pdf). Therefore the PoissonLogNormalQuadratureCompound class implements an approximation based on quadrature. Note: although the PoissonLogNormalQuadratureCompound is
approximately the Poisson-LogNormal compound distribution, it is itself a valid distribution. Viz., it possesses a sample, log_prob, mean, variance, etc. which are all mutually consistent.

Mathematical Details

The PoissonLogNormalQuadratureCompound approximates a Poisson-LogNormal compound distribution. Using variable-substitution and numerical quadrature (default: based on LogNormal quantiles) we can redefine the distribution to be a parameter-less convex combination of deg different Poisson samples. That is, defined over positive integers, this distribution is parameterized by a (batch of) loc and scale scalars.

The probability density function (pdf) is,

\[
\text{pdf}(k \mid \text{loc}, \text{scale}, \text{deg}) = \sum\{ \text{prob}[d] \text{ Poisson}(k \mid \lambda=\exp(\text{grid}[d])) : d=0, \ldots, \text{deg}-1 \}
\]

Note: probs returned by (optional) quadrature_fn are presumed to be either a length-quadrature_size vector or a batch of vectors in 1-to-1 correspondence with the returned grid. (I.e., broadcasting is only partially supported.)

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(), tfd_dirichlet_multinomial(), tfd_empirical(), tfd_exp半个小时() , tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_mixing_distribution(), tfd_multinomial(), tfd_multivariate_normal_diagonal(), tfd_multivariate_normal_diagonal_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal(), tfd_multivariate_normal_density_logistic(), tfd_multivariate_student_t_linear_operator(), tfd_negexp(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relpex(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_diag(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
The Power Spherical distribution over unit vectors on $S^{n-1}$.

Description

The Power Spherical distribution is a distribution over vectors on the unit hypersphere $S^{n-1}$ embedded in $n$ dimensions ($\mathbb{R}^n$). It serves as an alternative to the von Mises-Fisher distribution with a simpler (faster) log_prob calculation, as well as a reparameterizable sampler. In contrast, the Power Spherical distribution does have -mean_direction as a point with zero density (and hence a neighborhood around that having arbitrarily small density), in contrast with the von Mises-Fisher distribution which has non-zero density everywhere. NOTE: mean_direction is not in general the mean of the distribution. For spherical distributions, the mean is generally not in the support of the distribution.

Usage

tfd_power_spherical(
    mean_direction,
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "PowerSpherical"
)

Arguments

mean_direction Floating-point Tensor with shape [B1, ..., Bn, N]. A unit vector indicating the mode of the distribution, or the unit-normalized direction of the mean.

concentration Floating-point Tensor having batch shape [B1, ..., Bn] broadcastable with mean_direction. The level of concentration of samples around the mean_direction. concentration=0 indicates a uniform distribution over the unit hypersphere, and concentration=+inf indicates a Deterministic distribution (delta function) at mean_direction.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical details
The probability density function (pdf) is,
pdf(x; mu, kappa) = C(kappa) \ (1 + mu^T x) ** k
where,
\[ C(kappa) = 2^{(a + b)} \pi^{b} \Gamma(a) / \Gamma(a + b) \]
a = (n - 1) / 2. + k
b = (n - 1) / 2.

where

- mean_direction = mu; a unit vector in R^n
- concentration = kappa; scalar real >= 0, concentration of samples around mean_direction,
  where 0 pertains to the uniform distribution on the hypersphere, and 'inf' indicates a delta function at mean_direction.

Value
a distribution instance.

See Also
For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(),
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(),
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag_plus_low_rank(),
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(),
tfd_poisson(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(),
tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(),
tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(),
tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_uniform(), tfd_variational_gaussian_process(),
tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(),
tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(),
tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(),
tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()
**tfd_prob**  
*Probability density/mass function.*

**Description**

Probability density/mass function.

**Usage**

```
tfd_prob(distribution, value, ...)
```

**Arguments**

- `distribution` The distribution being used.
- `value` float or double Tensor.
- `...` Additional parameters passed to Python.

**Value**

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

**See Also**

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()`

**Examples**

```r
  d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
  x <- d %>% tfd_sample()
  d %>% tfd_prob(x)
```

---

**tfd_probit_bernoulli**  
*ProbitBernoulli distribution.*

**Description**

The ProbitBernoulli distribution with `probs` parameter, i.e., the probability of a 1 outcome (vs a 0 outcome). Unlike a regular Bernoulli distribution, which uses the logistic (aka 'sigmoid') function to go from the un-constrained parameters to probabilities, this distribution uses the CDF of the standard normal distribution:

\[
p(x=1; \text{probits}) = 0.5 \times (1 + \text{erf}(\text{probits} / \sqrt{2}))
\]

\[
p(x=0; \text{probits}) = 1 - p(x=1; \text{probits})
\]

Where \text{erf} is the error function. A typical application of this distribution is in probit regression.
tfd_probit_bernoulli

Usage
tfd_probit_bernoulli(
  probits = NULL,
  probs = NULL,
  dtype = tf$int32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "ProbitBernoulli"
)

Arguments

probits An N-D Tensor representing the probit-odds of a 1 event. Each entry in the
  Tensor parameterizes an independent ProbitBernoulli distribution where the
  probability of an event is normal_cdf(probits). Only one of probits or probs
  should be passed in.

probs An N-D Tensor representing the probability of a 1 event. Each entry in the
  Tensor parameterizes an independent ProbitBernoulli distribution. Only one of
  probits or probs should be passed in.

dtype The type of the event samples. Default: int32.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
  validity despite possibly degrading runtime performance. When FALSE invalid
  inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
  the value NaN to indicate the result is undefined. When FALSE, an exception is
  raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
  tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
  tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial,
  tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
  tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
  tfd_generated_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
  tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
  tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
  tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
  tfd_logistic().
```
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(),
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(),
tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(),
tfd_pareto(), tfd_pixel_cnn(), tfd_poisson(),
tfd_power_spherical(), tfd_quantized(),
tfd_quantile(), tfd_quantized(),
tfd_relaxed_bernoulli(),
tfd_neg_binomial(),
tfd_pareto(),
tfd_pixel_cnn(),
tfd_poisson_log_normal_quadrature_compound(),
tfd_skellam(),
tfd_student_t_process(),
tfd_transformed_distribution(),
tfd_triangular(),
tfd_truncated_cauchy(),
tfd_truncated_normal(),
tfd_uniform(),
tfd_von_mises_fisher(),
tfd_von_mises(),
tfd_weibull(),
tfd_wishart_linear_operator(),
tfd_zipf()
```

```
tfd_quantile

Quantile function. Aka "inverse cdf" or "percent point function".

Description

Given random variable X and p in [0, 1], the quantile is: tfd_quantile(p) := x such that P[X <= x] == p

Usage

```
tfd_quantile(distribution, value, ...)
```

Arguments

- `distribution` The distribution being used.
- `value` float or double Tensor.
- `...` Additional parameters passed to Python.

Value

a Tensor of shape `sample_shape(x) + self$batch_shape` with values of type `self$dtype`.

See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`, `tfd_variance()```

Examples

```
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_quantile(0.5)
```
tfd_quantized  Distribution representing the quantization $Y = \text{ceiling}(X)$

Description

Definition in Terms of Sampling

Usage

tfd_quantized(
    distribution,
    low = NULL,
    high = NULL,
    validate_args = FALSE,
    name = "QuantizedDistribution"
)

Arguments

distribution  The base distribution class to transform. Typically an instance of Distribution.
low  Tensor with same dtype as this distribution and shape able to be added to samples. Should be a whole number. Default NULL. If provided, base distribution’s prob should be defined at low.
high  Tensor with same dtype as this distribution and shape able to be added to samples. Should be a whole number. Default NULL. If provided, base distribution’s prob should be defined at high - 1. high must be strictly greater than low.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
name  name prefixed to Ops created by this class.

Details

1. Draw $X$
2. Set $Y \leftarrow \text{ceiling}(X)$
3. If $Y < \text{low}$, reset $Y \leftarrow \text{low}$
4. If $Y > \text{high}$, reset $Y \leftarrow \text{high}$
5. Return $Y$

Definition in Terms of the Probability Mass Function

Given scalar random variable $X$, we define a discrete random variable $Y$ supported on the integers as follows:

$$P[Y = j] := P[X \leq \text{low}], \text{ if } j = \text{low},$$
$$:= P[X > \text{high} - 1], \text{ if } j = \text{high},$$
$$:= 0, \text{ if } j < \text{low} \text{ or } j > \text{high},$$
$$:= P[j - 1 < X \leq j], \text{ all other } j.$$
Conceptually, without cutoffs, the quantization process partitions the real line $\mathbb{R}$ into half open intervals, and identifies an integer $j$ with the right endpoints:

$$R = \ldots (-\infty, -2](-2, -1](-1, 0](0, 1](1, 2](2, 3](3, 4] \ldots$$

$$j = \ldots -1 \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad \ldots$$

$P[Y = j]$ is the mass of $X$ within the $j$th interval. If $\text{low} = 0$, and $\text{high} = 2$, then the intervals are redrawn and $j$ is re-assigned:

$$R = (-\infty, 0](0, 1](1, \infty)$$

$$j = 0 \quad 1 \quad 2$$

$P[Y = j]$ is still the mass of $X$ within the $j$th interval.

@section References:


Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chichi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet(), tfd_dirichlet_multinomial(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exp_inverse_gamma(), tfd_finite(), tfd_from_single_event(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_gaussian(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logisitic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_c covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_distribution(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_ exponential_linear_operator(), tfd_vector_normal_diag(), tfd_vector_ normal_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_vector_tril().
tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

tfd_relaxed_bernoulli  
RelaxedBernoulli distribution with temperature and logits parameters

Description

The RelaxedBernoulli is a distribution over the unit interval (0,1), which continuously approximates a Bernoulli. The degree of approximation is controlled by a temperature: as the temperature goes to 0 the RelaxedBernoulli becomes discrete with a distribution described by the logits or probs parameters, as the temperature goes to infinity the RelaxedBernoulli becomes the constant distribution that is identically 0.5.

Usage

tfd_relaxed_bernoulli(
    temperature,
    logits = NULL,
    probs = NULL,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "RelaxedBernoulli"
)

Arguments

- **temperature**: An 0-D Tensor, representing the temperature of a set of RelaxedBernoulli distributions. The temperature should be positive.
- **logits**: An N-D Tensor representing the log-odds of a positive event. Each entry in the Tensor parametrizes an independent RelaxedBernoulli distribution where the probability of an event is sigmoid(logits). Only one of logits or probs should be passed in.
- **probs**: A N-D Tensor representing the probability of a positive event. Each entry in the Tensor parameterizes an independent Bernoulli distribution. Only one of logits or probs should be passed in.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
Details

The RelaxedBernoulli distribution is a reparameterized continuous distribution that is the binary special case of the RelaxedOneHotCategorical distribution (Maddison et al., 2016; Jang et al., 2016). For details on the binary special case see the appendix of Maddison et al. (2016) where it is referred to as BinConcrete. If you use this distribution, please cite both papers.

Some care needs to be taken for loss functions that depend on the log-probability of RelaxedBernoullis, because computing log-probabilities of the RelaxedBernoulli can suffer from underflow issues. In many case loss functions such as these are invariant under invertible transformations of the random variables. The KL divergence, found in the variational autoencoder loss, is an example. Because RelaxedBernoullis are sampled by a Logistic random variable followed by a `tf.sigmoid` op, one solution is to treat the Logistic as the random variable and `tf.sigmoid` as downstream. The KL divergences of two Logitics, which are always followed by a `tf.sigmoid` op, is equivalent to evaluating KL divergences of RelaxedBernoulli samples. See Maddison et al., 2016 for more details where this distribution is called the BinConcrete. An alternative approach is to evaluate Bernoulli log probability or KL directly on relaxed samples, as done in Jang et al., 2016. In this case, guarantees on the loss are usually violated. For instance, using a Bernoulli KL in a relaxed ELBO is no longer a lower bound on the log marginal probability of the observation. Thus care and early stopping are important.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`. Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisquare()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_expovariate()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_lognormal()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_categorical()`, `tfd_mixture_normal_diag_plus_low_rank()`, `tfd_mixture_normal_diag()`, `tfd_mixture_normal_full_copula()`, `tfd_mixture_normal_full_contingency()`, `tfd_mixture_normal_full_distribution()`, `tfd_mixture_normal_full_linear_operator()`, `tfd_mixture_normal_tri_l()`, `tfd_mixture_normal_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_quantized()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`,
tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(),
tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

---

**tfd_relaxed_one_hot_categorical**

*RelaxedOneHotCategorical distribution with temperature and logits*

Description

The RelaxedOneHotCategorical is a distribution over random probability vectors, vectors of positive real values that sum to one, which continuously approximates a OneHotCategorical. The degree of approximation is controlled by a temperature: as the temperature goes to 0 the RelaxedOneHotCategorical becomes discrete with a distribution described by the logits or probs parameters, as the temperature goes to infinity the RelaxedOneHotCategorical becomes the constant distribution that is identically the constant vector of \((1/\text{event\_size}, ..., 1/\text{event\_size})\). The RelaxedOneHotCategorical distribution was concurrently introduced as the Gumbel-Softmax (Jang et al., 2016) and Concrete (Maddison et al., 2016) distributions for use as a reparameterized continuous approximation to the Categorical one-hot distribution. If you use this distribution, please cite both papers.

Usage

```python
  tfd_relaxed_one_hot_categorical(
      temperature,
      logits = NULL,
      probs = NULL,
      validate_args = FALSE,
      allow_nan_stats = TRUE,
      name = "RelaxedOneHotCategorical"
  )
```

Arguments

- **temperature**: An 0-D Tensor, representing the temperature of a set of RelaxedOneHotCategorical distributions. The temperature should be positive.
- **logits**: An N-D Tensor, N >= 1, representing the log probabilities of a set of RelaxedOneHotCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of logits for each class. Only one of logits or probs should be passed in.
- **probs**: An N-D Tensor, N >= 1, representing the probabilities of a set of RelaxedOneHotCategorical distributions. The first N - 1 dimensions index into a batch of independent distributions and the last dimension represents a vector of probabilities for each class. Only one of logits or probs should be passed in.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Value
a distribution instance.

References
• Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax.
  2016.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd Autoregressive()`, `tfd BatchReshape()`, `tfd Bates()`, `tfd Bernoulli()`,
`tfd BetaBinomial()`, `tfd Beta()`, `tfd Binomial()`, `tfd Categorical()`, `tfd Cauchy()`, `tfd Chi2()`,
`tfd Chi()`, `tfd CholeskyLKJ()`, `tfd ContinuousBernoulli()`, `tfd Deterministic()`, `tfd DirichletMultinomial()`,
`tfd Dirichlet()`, `tfd Empirical()`, `tfd ExpGamma()`, `tfd ExpInverseGamma()`, `tfd Exponential()`,
`tfd GammaGamma()`, `tfd Gamma()`, `tfd GeneralizedNormal()`, `tfd Geometric()`, `tfd Gumbel()`, `tfd HalfCauchy()`, `tfd HalfNormal()`,
`tfd HiddenMarkovModel()`, `tfd Horseshoe()`, `tfd Independent()`, `tfd InverseGamma()`,
`tfd InverseGaussian()`, `tfd JohnsonSU()`, `tfd JointDistributionNamedAutoBatched()`,
`tfd JointDistributionNamed()`, `tfd JointDistributionSequentialAutoBatched()`,
`tfd JointDistributionSequential()`, `tfd Kumaraswamy()`, `tfd Laplace()`, `tfd LinearGaussianStateSpaceModel()`,
`tfd LKJ()`, `tfd Logistic()`, `tfd LogNormal()`, `tfd Logistic()`, `tfd MixtureSameFamily()`,
`tfd Mixture()`, `tfd Multinomial()`, `tfd MultivariateNormalDiagPlusLowRank()`, `tfd MultivariateNormalFullCovariance()`,
`tfd MultivariateNormalLinearOperator()`, `tfd MultivariateNormalTriL()`, `tfd MultivariateStudentTLinearOperator()`, `tfd NegativeBinomial()`,
`tfd Normal()`, `tfd OneHotCategorical()`, `tfd Pareto()`, `tfd PixelCNN()`, `tfd PoissonLogNormalQuadratureApprox()`,
`tfd Poisson()`, `tfd PowerSpherical()`, `tfd ProbitBernoulli()`, `tfd Quantized()`, `tfd RelaxedBernoulli()`,
`tfd SampleDistribution()`, `tfd SinhArcsinv()`, `tfd Skellam()`, `tfd SphericalUniform()`,
`tfd StudentTProcess()`, `tfd StudentT()`, `tfd TransformedDistribution()`, `tfd Triangular()`,
`tfd TruncatedCauchy()`, `tfd TruncatedNormal()`, `tfd Uniform()`, `tfd VariationalGaussianProcess()`,
`tfd VectorDiffeomixture()`, `tfd VectorExponentialDiag()`, `tfd VectorExponentialLinearOperator()`,
`tfd VectorLaplaceDiag()`, `tfd VectorLaplaceLinearOperator()`, `tfd VectorSinhArcsinvDiag()`,
`tfd VonMisesFisher()`, `tfd VonMises()`, `tfd Weibull()`, `tfd WishartLinearOperator()`,
`tfd WishartTriL()`, `tfd Wishart()`, `tfd Zipf()`
tfd_sample

Generate samples of the specified shape.

Description

Note that a call to tfd_sample() without arguments will generate a single sample.

Usage

tfd_sample(distribution, sample_shape = list(), ...)

Arguments

distribution  The distribution being used.
sample_shape  0D or 1D int32 Tensor. Shape of the generated samples.
...           Additional parameters passed to Python.

Value

a Tensor with prepended dimensions sample_shape.

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(),
tfd_mode(), tfd_prob(), tfd_quantile(), tfd_stddev(), tfd_survival_function(), tfd_variance()

Examples

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_sample()
```

tfd_sample_distribution

Sample distribution via independent draws.

Description

This distribution is useful for reducing over a collection of independent, identical draws. It is
otherwise identical to the input distribution.
tfd_sample_distribution

Usage

tfd_sample_distribution(
distribution,
sample_shape = list(),
validate_args = FALSE,
name = NULL
)

Arguments

distribution  The base distribution instance to transform. Typically an instance of Distribution.
sample_shape  integer scalar or vector Tensor representing the shape of a single sample.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
name  The name for ops managed by the distribution. Default value: NULL (i.e., 'Sample' + distribution$name).

Details

Mathematical Details The probability function is,

\[ p(x) = \prod_{i = 0, \ldots, (n - 1)} p(x[i]) \]

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space(), tfd_lkj(), tfd_log_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform().
The SinhArcsinh transformation of a distribution on (-inf, inf)

Description
This distribution models a random variable, making use of a SinhArcsinh transformation (which has adjustable tailweight and skew), a rescaling, and a shift. The SinhArcsinh transformation of the Normal is described in great depth in Sinh-arcsinh distributions. Here we use a slightly different parameterization, in terms of tailweight and skewness. Additionally we allow for distributions other than Normal, and control over scale as well as a "shift" parameter \( \text{loc} \).

Usage
```r
tfd_sinh_arcsinh(
  loc,
  scale,
  skewness = NULL,
  tailweight = NULL,
  distribution = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "SinhArcsinh"
)
```

Arguments
- **loc**: Floating-point Tensor.
- **scale**: Tensor of same dtype as loc.
- **skewness**: Skewness parameter. Default is \( 0.0 \) (no skew).
- **tailweight**: Tailweight parameter. Default is \( 1.0 \) (unchanged tailweight).
- **distribution**: `tf$distributions$Distribution-like instance. Distribution that is transformed to produce this distribution. Default is `tfd_normal(0,1)`. Must be a scalar-batch, scalar-event distribution. Typically `distribution$reparameterization_type = FULLY_REPARAMETERIZED` or it is a function of non-trainable parameters. WARNING: If you backprop through a SinhArcsinh sample and `distribution` is not FULLY_REPARAMETERIZED yet is a function of trainable variables, then the gradient will be incorrect!
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

Given random variable $Z$, we define the SinhArcsinh transformation of $Z$, $Y$, parameterized by $(\text{loc}, \text{scale}, \text{skewness}, \text{tailweight})$, via the relation:

$$Y := \text{loc} + \text{scale} \times \text{F}(Z) \times \left(2 / \text{F}_0(2)\right)$$

$$\text{F}(Z) := \text{Sinh}\left( (\text{Arcsinh}(Z) + \text{skewness}) \times \text{tailweight} \right)$$

$$\text{F}_0(Z) := \text{Sinh}\left( \text{Arcsinh}(Z) \times \text{tailweight} \right)$$

This distribution is similar to the location-scale transformation $L(Z) := \text{loc} + \text{scale} \times Z$ in the following ways:

- If $\text{skewness} = 0$ and $\text{tailweight} = 1$ (the defaults), $\text{F}(Z) = Z$, and then $Y = L(Z)$ exactly.
- $\text{loc}$ is used in both to shift the result by a constant factor.
- The multiplication of $\text{scale}$ by $2 / \text{F}_0(2)$ ensures that if $\text{skewness} = 0$, $\text{P}[Y - \text{loc} <= 2 \times \text{scale}] = \text{P}[L(Z) - \text{loc} <= 2 \times \text{scale}]$. Thus it can be said that the weights in the tails of $Y$ and $L(Z)$ beyond $\text{loc} + 2 \times \text{scale}$ are the same.

This distribution is different than $\text{loc} + \text{scale} \times Z$ due to the reshaping done by $\text{F}$:

- Positive (negative) skewness leads to positive (negative) skew.
- Positive skew means, the mode of $\text{F}(Z)$ is "tilted" to the right.
- Positive skew means positive values of $\text{F}(Z)$ become more likely, and negative values become less likely.
- Larger (smaller) tailweight leads to fatter (thinner) tails.
- Fatter tails mean larger values of $|\text{F}(Z)|$ become more likely.
- Tailweight $< 1$ leads to a distribution that is "flat" around $Y = \text{loc}$, and a very steep drop-off in the tails.
- Tailweight $> 1$ leads to a distribution more peaked at the mode with heavier tails.

To see the argument about the tails, note that for $|Z| >> 1$ and $|Z| >> (|\text{skewness}| \times \text{tailweight})^{\text{tailweight}}$, we have $Y \approx 0.5 Z^{\text{tailweight}} e^{(\text{sign}(Z) \times \text{skewness} \times \text{tailweight})}$.

To see the argument regarding multiplying $\text{scale}$ by $2 / \text{F}_0(2)$,

$$\text{P}[\frac{(Y - \text{loc})}{\text{scale}} <= 2] = \text{P}[\text{F}(Z) \times (2 / \text{F}_0(2)) <= 2]$$

$$= \text{P}[\text{F}(Z) <= \text{F}_0(2)]$$

$$= \text{P}[Z <= 2] \quad \text{(if } F = F_0).$$
Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Usage

tfd_skellam(
    rate1 = NULL,
    rate2 = NULL,
    log_rate1 = NULL,
    log_rate2 = NULL,
    force_probs_to_zero_outside_support = FALSE,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Skellam"
)

Arguments

rate1  Floating point tensor, the first rate parameter. rate1 must be positive. Must specify exactly one of rate1 and log_rate1.
rate2  Floating point tensor, the second rate parameter. rate must be positive. Must specify exactly one of rate2 and log_rate2.
log_rate1 Floating point tensor, the log of the first rate parameter. Must specify exactly one of rate1 and log_rate1.
log_rate2 Floating point tensor, the log of the second rate parameter. Must specify exactly one of rate2 and log_rate2.
force_probs_to_zero_outside_support logical. When TRUE, log_prob returns -inf (and prob returns 0) for non-integer inputs. When FALSE, log_prob evaluates the Skellam pmf as a continuous function (note that this function is not itself a normalized probability log-density). Default value: FALSE.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical Details The probability mass function (pmf) is,

\[
pmf(k; l_1, l_2) = (l_1 / l_2)^{k/2} \cdot I_k(2 \cdot \sqrt{l_1 \cdot l_2}) / Z
\]

\[
Z = \exp(l_1 + l_2).
\]

where rate1 = l1, rate2 = l2, Z is the normalizing constant and I_k is the modified bessel function of the first kind.
The uniform distribution over unit vectors on $S^{n-1}$.

**Description**

The uniform distribution on the unit hypersphere $S^{n-1}$ embedded in $n$ dimensions ($\mathbb{R}^n$).

**Usage**

```r
tfd_spherical_uniform(
  dimension,
  batch_shape = list(),
  dtype = tf$float32,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "SphericalUniform"
)```

Arguments

dimension integer. The dimension of the embedded space where the sphere resides.
batch_shape Positive integer-like vector-shaped Tensor representing the new shape of the
batch dimensions. Default value: [].
dtype dtype of the generated samples.
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for
validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.
name name prefixed to Ops created by this class.

Details

Mathematical details
The probability density function (pdf) is,

$$pdf(x; n) = \frac{1.}{A(n)}$$
where,

$$A(n) = 2 \times \pi^{n/2} / \Gamma(n/2),$$

$$\Gamma$$ being the Gamma function.

where n = dimension; corresponds to $$S^{n-1}$$ embedded in $$\mathbb{R}^n$$.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().
Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(),
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(),
tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(),
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(),
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial()
Standard deviation.

Description

Standard deviation is defined as, \( \text{stddev} = \sqrt{\text{Var}(X)} = \sqrt{E[(X - \mu)^2]} \) where \( X \) is the random variable associated with this distribution, \( E \) denotes expectation, and \( \text{Var}(\text{shape}) = \text{batch_shape} + \text{event_shape} \).

Usage

tfd_stddev(distribution, ...)

Arguments

distribution  The distribution being used.
...

Additional parameters passed to Python.

Value

a Tensor of shape \( \text{sample_shape}(x) + \text{self$batch_shape} \) with values of type \( \text{self$dtype} \).

See Also

Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(), tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(), tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_survival_function(), tfd_variance()

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_stddev()
tfd_student_t  Student’s t-distribution

Description
This distribution has parameters: degree of freedom df, location loc, and scale.

Usage
tfd_student_t(
  df,
  loc,
  scale,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "StudentT"
)

Arguments
df  Floating-point Tensor. The degrees of freedom of the distribution(s). df must contain only positive values.
loc  Floating-point Tensor. The mean(s) of the distribution(s).
scale  Floating-point Tensor. The scaling factor(s) for the distribution(s). Note that scale is not technically the standard deviation of this distribution but has semantics more similar to standard deviation than variance.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name  name prefixed to Ops created by this class.

Details
Mathematical details
The probability density function (pdf) is,

\[
pdf(x; \text{df, mu, sigma}) = \frac{(1 + y^2 / \text{df})^{-0.5 (\text{df} + 1)}}{Z}
\]

where,

\[
y = (x - \text{mu}) / \text{sigma}
\]

\[
Z = \text{abs}(\text{sigma}) \sqrt{\text{df} \pi} \frac{\Gamma(0.5 \text{df})}{\Gamma(0.5 (\text{df} + 1))}
\]

where:
• \( \text{loc} = \mu \),
• \( \text{scale} = \sigma \), and,
• \( Z \) is the normalization constant, and,
• \( \Gamma \) is the gamma function. The StudentT distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X \sim \text{StudentT}(df, \text{loc}=0, \text{scale}=1) \\
Y = \text{loc} + \text{scale} \times X
\]

Notice that scale has semantics more similar to standard deviation than variance. However it is not actually the std. deviation; the Student's t-distribution std. dev. is \( \text{scale} \sqrt{df / (df - 2)} \) when \( df > 2 \).

Samples of this distribution are reparameterized (pathwise differentiable). The derivatives are computed using the approach described in the paper *Michael Figurnov, Shakir Mohamed, Andriy Mnih. Implicit Reparameterization Gradients, 2018*

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

**tfd_student_t_process**  *Marginal distribution of a Student’s T process at finitely many points*

**Description**

A Student’s T process (TP) is an indexed collection of random variables, any finite collection of which are jointly Multivariate Student’s T. While this definition applies to finite index sets, it is typically implicit that the index set is infinite; in applications, it is often some finite dimensional real or complex vector space. In such cases, the TP may be thought of as a distribution over (real- or complex-valued) functions defined over the index set.

**Usage**

```r
  tfd_student_t_process(
    df,
    kernel,
    index_points,
    mean_fn = NULL,
    jitter = 1e-06,
    validate_args = FALSE,
    allow_nan_stats = FALSE,
    name = "StudentTProcess"
  )
```

**Arguments**

- **df**  Positive Floating-point Tensor representing the degrees of freedom. Must be greater than 2.
- **kernel**  PositiveSemidefiniteKernel-like instance representing the TP’s covariance function.
- **index_points**  float Tensor representing finite (batch of) vector(s) of points in the index set over which the TP is defined. Shape has the form `[b1,...,bB,e,f1,...,fF]` where `F` is the number of feature dimensions and must equal `kernel.feature_ndims` and `e` is the number (size) of index points in each batch. Ultimately this distribution corresponds to a `e`-dimensional multivariate Student’s T. The batch shape must be broadcastable with `kernel.batch_shape` and any batch dims yielded by `mean_fn`.
- **mean_fn**  Function that acts on `index_points` to produce a (batch of) vector(s) of mean values at `index_points`. Takes a Tensor of shape `[b1,...,bB,f1,...,fF]` and returns a Tensor whose shape is broadcastable with `[b1,...,bB]`. Default value: `NULL` implies constant zero function.
- **jitter**  float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: `1e-6`.
- **validate_args**  Logical, default `FALSE`. When `TRUE` distribution parameters are checked for validity despite possibly degrading runtime performance. When `FALSE` invalid inputs may silently render incorrect outputs. Default value: `FALSE`.

allow_nan_stats

Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name

name prefixed to Ops created by this class.

Details

Just as Student’s T distributions are fully specified by their degrees of freedom, location and scale, a Student’s T process can be completely specified by a degrees of freedom parameter, mean function and covariance function.

Let $S$ denote the index set and $K$ the space in which each indexed random variable takes its values (again, often $\mathbb{R}$ or $\mathbb{C}$). The mean function is then a map $m: S \rightarrow K$, and the covariance function, or kernel, is a positive-definite function $k: (S \times S) \rightarrow K$. The properties of functions drawn from a TP are entirely dictated (up to translation) by the form of the kernel function.

This distribution represents the marginal joint distribution over function values at a given finite collection of points $[x[1], ..., x[N]]$ from the index set $S$. By definition, this marginal distribution is just a multivariate Student’s T distribution, whose mean is given by the vector $[m(x[1]), ..., m(x[N])]$ and whose covariance matrix is constructed from pairwise applications of the kernel function to the given inputs:

$$
\begin{bmatrix}
    k(x[1], x[1]) & k(x[1], x[2]) & \cdots & k(x[1], x[N]) \\
    k(x[2], x[1]) & k(x[2], x[2]) & \cdots & k(x[2], x[N]) \\
    \vdots & \vdots & \ddots & \vdots \\
    k(x[N], x[1]) & k(x[N], x[2]) & \cdots & k(x[N], x[N])
\end{bmatrix}
$$

For this to be a valid covariance matrix, it must be symmetric and positive definite; hence the requirement that $k$ be a positive definite function (which, by definition, says that the above procedure will yield PD matrices). Note also we use a parameterization as suggested in Shat et al. (2014), which requires $df$ to be greater than 2. This allows for the covariance for any finite dimensional marginal of the TP (a multivariate Student’s T distribution) to just be the PD matrix generated by the kernel.

Mathematical Details

The probability density function (pdf) is a multivariate Student’s T whose parameters are derived from the TP’s properties:

$$\text{pdf}(x; df, index\_points, mean\_fn, kernel) = \text{MultivariateStudentT}(df, \text{loc}, K)$$

$$K = (df - 2) / df \times (kernel.matrix(index\_points, index\_points) + \text{jitter} \times \text{eye}(N))$$

$$\text{loc} = (x - mean\_fn(index\_points))^T @ K @ (x - mean\_fn(index\_points))$$

where:

- $df$ is the degrees of freedom parameter for the TP.
- $\text{index\_points}$ are points in the index set over which the TP is defined.
- $\text{mean\_fn}$ is a callable mapping the index set to the TP’s mean values.
- $\text{kernel}$ is $\text{PositiveSemidefiniteKernel}$-like and represents the covariance function of the TP.
• jitter is added to the diagonal to ensure positive definiteness up to machine precision (otherwise Cholesky-decomposition is prone to failure),
• \( \text{eye}(N) \) is an N-by-N identity matrix.

Value

a distribution instance.

References


See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


tfd_survival_function  Survival function.

description

Given random variable \( X \), the survival function is defined: \( \text{tfd_survival_function}(x) = P[X > x] = 1 - P[X < x] = 1 - \text{cdf}(x) \).
tfd_transformed_distribution

Usage
tfd_survival_function(distribution, value, ...)

Arguments
distribution The distribution being used.
value float or double Tensor.
... Additional parameters passed to Python.

Value
a Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.

See Also
Other distribution methods: tfd_cdf(), tfd_covariance(), tfd_cross_entropy(), tfd_entropy(),
tfd_kl_divergence(), tfd_log_cdf(), tfd_log_prob(), tfd_log_survival_function(), tfd_mean(),
tfd_mode(), tfd_prob(), tfd_quantile(), tfd_sample(), tfd_stddev(), tfd_variance()

Examples

d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
x <- d %>% tfd_sample()
d %>% tfd_survival_function(x)

---

tfd_transformed_distribution

A Transformed Distribution

Description
A TransformedDistribution models p(y) given a base distribution p(x), and a deterministic, invertible, differentiable transform, Y = g(X). The transform is typically an instance of the Bijector class and the base distribution is typically an instance of the Distribution class.

Usage
tfd_transformed_distribution(
  distribution,
  bijector,
  batch_shape = NULL,
  event_shape = NULL,
  kwargs_split_fn = NULL,
  validate_args = FALSE,
  parameters = NULL,
  name = NULL
)
**Arguments**

- **distribution**
  The base distribution instance to transform. Typically an instance of Distribution.

- **bijector**
  The object responsible for calculating the transformation. Typically an instance of Bijector.

- **batch_shape**
  integer vector Tensor which overrides distribution batch_shape; valid only if distribution.is_scalar_batch().

- **event_shape**
  integer vector Tensor which overrides distribution event_shape; valid only if distribution.is_scalar_event().

- **kwargs_split_fn**
  Python callable which takes a kwargs dict and returns a tuple of kwargs dicts for each of the distribution and bijector parameters respectively. Default value: _default_kwargs_split_fn (i.e., lambda kwargs: (kwargs.get('distribution_kwargs', {}), kwargs.get('bijector_kwargs', {}))).

- **validate_args**
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

- **parameters**
  Locals dict captured by subclass constructor, to be used for copy/slice re-instantiation operations.

- **name**
  The name for ops managed by the distribution. Default value: bijector.name + distribution.name.

**Details**

A Bijector is expected to implement the following functions:

- **forward**
- **inverse**
- **inverse_log_det_jacobian**

The semantics of these functions are outlined in the Bijector documentation.

We now describe how a TransformedDistribution alters the input/outputs of a Distribution associated with a random variable (rv) $X$. Write $\text{cdf}(Y=y)$ for an absolutely continuous cumulative distribution function of random variable $Y$; write the probability density function $\text{pdf}(Y=y):= d^k / (dy_1,...,dy_k) \text{cdf}(Y=y)$ for its derivative wrt to $Y$ evaluated at $y$. Assume that $Y = g(X)$ where $g$ is a deterministic diffeomorphism, i.e., a non-random, continuous, differentiable, and invertible function. Write the inverse of $g$ as $X = g^{-1}(Y)$ and $(J o g)(x)$ for the Jacobian of $g$ evaluated at $x$.

A TransformedDistribution implements the following operations:

- **sample** Mathematically: $Y = g(X)$ Programmatically: bijector.forward(distribution.sample(...))
- **log_prob** Mathematically: $(\log o \text{pdf})(Y=y) = (\log o \text{pdf} o g^\{-1\})(y) + (\log o \text{abs o det o J o g^\{-1\}})(y)$ Programmatically: $(\text{distribution}.\text{log_prob(bijector.inverse(y))) + bijector.\text{inverse_log_det_jacobian(y))}$
- **log_cdf** Mathematically: $(\log o \text{cdf})(Y=y) = (\log o \text{cdf} o g^\{-1\})(y)$ Programmatically: $(\text{distribution}.\text{log_cdf(bijector.inverse(x)))}$
- and similarly for: cdf, prob, log_survival_function, survival_function.
tfd_triangular

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


### tfd_triangular

**Triangular distribution with low, high and peak parameters**

**Description**

The parameters low, high and peak must be shaped in a way that supports broadcasting (e.g., high - low is a valid operation).

**Usage**

```python
tfd_triangular(
    low = 0,
    high = 1,
    peak = 0.5,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Triangular"
)
```
Arguments

low Floating point tensor, lower boundary of the output interval. Must have low < high. Default value: 0.

high Floating point tensor, upper boundary of the output interval. Must have low < high. Default value: 1.

peak Floating point tensor, mode of the output interval. Must have low <= peak and peak <= high. Default value: 0.5.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nans Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

The Truncated Cauchy distribution.

Description

The truncated Cauchy is a Cauchy distribution bounded between low and high (the pdf is 0 outside these bounds and renormalized). Samples from this distribution are differentiable with respect to loc and scale, but not with respect to the bounds low and high.

Usage

tfd_truncated_cauchy(
  loc,
  scale,
  low,
  high,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "TruncatedCauchy"
)

Arguments

loc Floating point tensor; the modes of the corresponding non-truncated Cauchy distribution(s).

scale Floating point tensor; the scales of the distribution(s). Must contain only positive values.

low float Tensor representing lower bound of the distribution’s support. Must be such that low < high.

high float Tensor representing upper bound of the distribution’s support. Must be such that low < high.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical Details

The probability density function (pdf) of this distribution is:
pdf(x; loc, scale, low, high) =
    { 1 / (pi * scale * (1 + z**2) * A) for low <= x <= high
    { 0 otherwise

where
    z = (x - loc) / scale
    A = CauchyCDF((high - loc) / scale) - CauchyCDF((low - loc) / scale)

where CauchyCDF is the cumulative density function of the Cauchy distribution with 0 mean and unit variance. This is a scalar distribution so the event shape is always scalar and the dimensions of the parameters define the batch_shape.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`. Other distributions: `tfd_autoregressive()`, `tfd_batch reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma_t()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_mixture()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_mixture()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_mixture()`, `tfd_neg_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformation_distri()`, `tfd_triangular()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_process()`, `tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`, `tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_sinh_arcsinh_diag()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart_trl()`, `tfd_wishart()`, `tfd_zipf()`
**Description**

The truncated normal is a normal distribution bounded between low and high (the pdf is 0 outside these bounds and renormalized). Samples from this distribution are differentiable with respect to loc, scale as well as the bounds, low and high, i.e., this implementation is fully reparameterizable. For more details, see here.

**Usage**

```python
tfd_truncated_normal(
    loc,
    scale,
    low,
    high,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "TruncatedNormal"
)
```

**Arguments**

- `loc` : Floating point tensor; the means of the distribution(s).
- `scale` : Floating point tensor; the stddevs of the distribution(s). Must contain only positive values.
- `low` : float Tensor representing lower bound of the distribution’s support. Must be such that low < high.
- `high` : float Tensor representing upper bound of the distribution’s support. Must be such that low < high.
- `validate_args` : Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` : Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` : name prefixed to Ops created by this class.

**Details**

Mathematical Details

The probability density function (pdf) of this distribution is:

\[
\text{pdf}(x; \text{loc}, \text{scale}, \text{low}, \text{high}) = \\
\{ (2 \pi)^{0.5} \exp(-0.5 y^2) / (\text{scale} * z) \text{ for } \text{low} \leq x \leq \text{high} \\
\{ 0 \text{ otherwise} \\
\]

\[
y = (x - \text{loc}) / \text{scale} \\
z = \text{NormalCDF}(\text{high} - \text{loc} / \text{scale}) - \text{NormalCDF}(\text{lower} - \text{loc} / \text{scale})
\]
where:

- NormalCDF is the cumulative density function of the Normal distribution with 0 mean and unit variance.

This is a scalar distribution so the event shape is always scalar and the dimensions of the parameters defined the batch_shape.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.


Description

Uniform distribution with low and high parameters

Mathematical Details
Usage

```r
tfd_uniform(
    low = 0,
    high = 1,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Uniform"
)
```

Arguments

- **low**: Floating point tensor, lower boundary of the output interval. Must have \(\text{low} < \text{high}\).
- **high**: Floating point tensor, upper boundary of the output interval. Must have \(\text{low} < \text{high}\).
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.

Details

The probability density function (pdf) is,

\[
pdf(x; a, b) = I[a <= x < b] / Z
\]

\[
Z = b - a
\]

where

- \(\text{low} = a\),
- \(\text{high} = b\),
- \(Z\) is the normalizing constant, and
- \(I[predicate]\) is the indicator function for predicate.

The parameters \(\text{low}\) and \(\text{high}\) must be shaped in a way that supports broadcasting (e.g., \(\text{high} - \text{low}\) is a valid operation).

Value

- a distribution instance.
See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_mixture(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature_compound(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distributions triangular_arcsinh(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diag(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(), tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(), tfd_wishart(), tfd_zipf()

tfd_variance

Description

Variance is defined as, \( \text{Var} = E[(X - E[X])**2] \) where \( X \) is the random variable associated with this distribution, \( E \) denotes expectation, and \( \text{Var} = \text{batch_shape} + \text{event_shape} \).

Usage

tfd_variance(distribution, ...)

Arguments

distribution The distribution being used.
...

Additional parameters passed to Python.

Value

a Tensor of shape sample_shape(x) + self$batch_shape with values of type self$dtype.
See Also

Other distribution methods: `tfd_cdf()`, `tfd_covariance()`, `tfd_cross_entropy()`, `tfd_entropy()`, `tfd_kl_divergence()`, `tfd_log_cdf()`, `tfd_log_prob()`, `tfd_log_survival_function()`, `tfd_mean()`, `tfd_mode()`, `tfd_prob()`, `tfd_quantile()`, `tfd_sample()`, `tfd_stddev()`, `tfd_survival_function()`

Examples

```r
d <- tfd_normal(loc = c(1, 2), scale = c(1, 0.5))
d %>% tfd_variance()
```

---

`tfd_variational_gaussian_process`

*Posterior predictive of a variational Gaussian process*

Description

This distribution implements the variational Gaussian process (VGP), as described in Titsias (2009) and Hensman (2013). The VGP is an inducing point-based approximation of an exact GP posterior. Ultimately, this Distribution class represents a marginal distribution over function values at a collection of `index_points`. It is parameterized by

- a kernel function,
- a mean function,
- the (scalar) observation noise variance of the normal likelihood,
- a set of index points,
- a set of inducing index points, and
- the parameters of the (full-rank, Gaussian) variational posterior distribution over function values at the inducing points, conditional on some observations.

Usage

```r
tfd_variational_gaussian_process(
  kernel,
  index_points,
  inducing_index_points,
  variational_inducing_observations_loc,
  variational_inducing_observations_scale,
  mean_fn = NULL,
  observation_noise_variance = 0,
  predictive_noise_variance = 0,
  jitter = 1e-06,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "VariationalGaussianProcess"
)
```
Arguments

kernel PositiveSemidefiniteKernel-like instance representing the GP’s covariance function.

index_points float Tensor representing finite (batch of) vector(s) of points in the index set over which the VGP is defined. Shape has the form \([b_1, ..., b_B, e_1, f_1, ..., f_F]\) where \(F\) is the number of feature dimensions and must equal `kernel.feature_ndims` and \(e_1\) is the number (size) of index points in each batch (we denote it \(e_1\) to distinguish it from the number of inducing index points, denoted \(e_2\) below). Ultimately the VariationalGaussianProcess distribution corresponds to an \(e_1\)-dimensional multivariate normal. The batch shape must be broadcastable with `kernel.batch_shape`, the batch shape of `inducing_index_points`, and any batch dims yielded by `mean_fn`.

inducing_index_points float Tensor of locations of inducing points in the index set. Shape has the form \([b_1, ..., b_B, e_2, f_1, ..., f_F]\), just like `index_points`. The batch shape components needn’t be identical to those of `index_points`, but must be broadcast compatible with them.

variational_inducing_observations_loc float Tensor; the mean of the (full-rank Gaussian) variational posterior over function values at the inducing points, conditional on observed data. Shape has the form \([b_1, ..., b_B, e_2]\), where \(b_1, ..., b_B\) is broadcast compatible with other parameters’ batch shapes, and \(e_2\) is the number of inducing points.

variational_inducing_observations_scale float Tensor; the scale matrix of the (full-rank Gaussian) variational posterior over function values at the inducing points, conditional on observed data. Shape has the form \([b_1, ..., b_B, e_2, e_2]\), where \(b_1, ..., b_B\) is broadcast compatible with other parameters and \(e_2\) is the number of inducing points.

mean_fn function that acts on index points to produce a (batch of) vector(s) of mean values at those index points. Takes a Tensor of shape \([b_1, ..., b_B, f_1, ..., f_F]\) and returns a Tensor whose shape is (broadcastable with) \([b_1, ..., b_B]\). Default value: NULL implies constant zero function.

observation_noise_variance float Tensor representing the variance of the noise in the Normal likelihood distribution of the model. May be batched, in which case the batch shape must be broadcastable with the shapes of all other batched parameters (kernel.batch_shape, index_points, etc.). Default value: 0.

predictive_noise_variance float Tensor representing additional variance in the posterior predictive model. If NULL, we simply re-use observation_noise_variance for the posterior predictive noise. If set explicitly, however, we use the given value. This allows us, for example, to omit predictive noise variance (by setting this to zero) to obtain noiseless posterior predictions of function values, conditioned on noisy observations.

jitter float scalar Tensor added to the diagonal of the covariance matrix to ensure positive definiteness of the covariance matrix. Default value: 1e-6.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name  name prefixed to Ops created by this class.

Details

A VGP is “trained” by selecting any kernel parameters, the locations of the inducing index points, and the variational parameters. Titsias (2009) and Hensman (2013) describe a variational lower bound on the marginal log likelihood of observed data, which this class offers through the `variational_loss` method (this is the negative lower bound, for convenience when plugging into a TF Optimizer’s `minimize` function). Training may be done in minibatches.

Titsias (2009) describes a closed form for the optimal variational parameters, in the case of sufficiently small observational data (i.e., small enough to fit in memory but big enough to warrant approximating the GP posterior). A method to compute these optimal parameters in terms of the full observational data set is provided as a static method, `optimal_variational_posterior`. It returns a `MultivariateNormalLinearOperator` instance with optimal location and scale parameters.

Mathematical Details

Notation We will in general be concerned about three collections of index points, and it’ll be good to give them names:

- \( x[1], ..., x[N] \): observation index points – locations of our observed data.
- \( z[1], ..., z[M] \): inducing index points – locations of the “summarizing” inducing points
- \( t[1], ..., t[P] \): predictive index points – locations where we are making posterior predictions based on observations and the variational parameters.

To lighten notation, we’ll use \( X, Z, T \) to denote the above collections. Similarly, we’ll denote by \( f(X) \) the collection of function values at each of the \( x[i] \), and by \( Y \), the collection of (noisy) observed data at each \( x[i] \). We’ll denote kernel matrices generated from pairs of index points as \( K_{tt}, K_{xt}, K_{tz}, \text{etc.} \)

\[
K_{tz} = 
\begin{vmatrix}
  k(t[1], z[1]) & k(t[1], z[2]) & \cdots & k(t[1], z[M]) \\
  k(t[2], z[1]) & k(t[2], z[2]) & \cdots & k(t[2], z[M]) \\
  \vdots & \vdots & \ddots & \vdots \\
  k(t[P], z[1]) & k(t[P], z[2]) & \cdots & k(t[P], z[M])
\end{vmatrix}
\]

Preliminaries A Gaussian process is an indexed collection of random variables, any finite collection of which are jointly Gaussian. Typically, the index set is some finite-dimensional, real vector space, and indeed we make this assumption in what follows. The GP may then be thought of as a distribution over functions on the index set. Samples from the GP are functions on the whole index set; these can’t be represented in finite compute memory, so one typically works with the marginals at a
finite collection of index points. The properties of the GP are entirely determined by its mean function \( m \) and covariance function \( k \). The generative process, assuming a mean-zero normal likelihood with stddev \( \sigma \), is

\[
f \sim \text{GP}(m, k) \quad Y \mid f(X) \sim \text{Normal}(f(X), \sigma), \quad i = 1, \ldots, N
\]

In finite terms (i.e., marginalizing out all but a finite number of \( f(X), \sigma \)), we can write

\[
f(X) \sim \text{MVN}(\text{loc}=m(X), \text{cov}=K_{xx}) \\
Y \mid f(X) \sim \text{Normal}(f(X), \sigma), \quad i = 1, \ldots, N
\]

Posterior inference is possible in analytical closed form but becomes intractible as data sizes get large. See Rasmussen (2006) for details.

The VGP

The VGP is an inducing point-based approximation of an exact GP posterior, where two approximating assumptions have been made:

1. function values at non-inducing points are mutually independent conditioned on function values at the inducing points,
2. the (expensive) posterior over function values at inducing points conditional on observations is replaced with an arbitrary (learnable) full-rank Gaussian distribution,

\[
q(f(Z)) = \text{MVN}(\text{loc}=m, \text{scale}=S),
\]

where \( m \) and \( S \) are parameters to be chosen by optimizing an evidence lower bound (ELBO). The posterior predictive distribution becomes

\[
q(f(T)) = \int df(Z) \, p(f(T) \mid f(Z)) \, q(f(Z)) = \text{MVN}(\text{loc} = A \odot m, \text{scale} = B^{1/2})
\]

where

\[
A = K_{tz} \odot K_{zz}^{-1} \\
B = K_{tt} - A \odot (K_{zz} - S \, S^T) \, A^T
\]

The approximate posterior predictive distribution \( q(f(T)) \) is what the VariationalGaussianProcess class represents.

Model selection in this framework entails choosing the kernel parameters, inducing point locations, and variational parameters. We do this by optimizing a variational lower bound on the marginal log likelihood of observed data. The lower bound takes the following form (see Titsias (2009) and Hensman (2013) for details on the derivation):

\[
L(Z, m, s, Y) = \text{MVN}(\text{loc} = (K_{zx} @ K_{zz}^{-1}) \odot m, \text{scale diag} = \sigma).\log\_prob(Y) - \\
(\text{Tr}(K_{xx} - K_{zx} @ K_{zz}^{-1} @ K_{xz}) + \\
\text{Tr}(S @ S^T @ K_{zz}^{-1} @ K_{zx} @ K_{xz} @ K_{zz}^{-1})) / (2 * \sigma^2) - \\
\text{KL}(q(f(Z)) \mid \mid p(f(Z)))
\]
where in the final KL term, \( p(f(Z)) \) is the GP prior on inducing point function values. This variational lower bound can be computed on minibatches of the full data set \((X, Y)\). A method to compute the negative variational lower bound is implemented as \texttt{VariationalGaussianProcess$variational_loss}.

Optimal variational parameters

As described in Titsias (2009), a closed form optimum for the variational location and scale parameters, \( m \) and \( S \), can be computed when the observational data are not prohibitively voluminous. The \texttt{optimal_variational_posterior} function to computes the optimal variational posterior distribution over inducing point function values in terms of the GP parameters (mean and kernel functions), inducing point locations, observation index points, and observations. Note that the inducing index point locations must still be optimized even when these parameters are known functions of the inducing index points. The optimal parameters are computed as follows:

\[
C = \sigma^{-2} (K_{zz} + K_{zx} @ K_{xz})^{-1}
\]

optimal Gaussian covariance: \( K_{zz} @ C @ K_{zz} \)

optimal Gaussian location: \( \sigma^{-2} K_{zz} @ C @ K_{zx} @ Y \)

Value

a distribution instance.

References


See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exp_gamma()}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_generalized_normal()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space()}, \texttt{tfd_lkj()}, \texttt{tfd_log_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_full_covariance()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature()}, \texttt{tfd_poisson()}, \texttt{tfd_power_spherical()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_relaxed_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_skellam()}, \texttt{tfd_spherical_uniform()}, \texttt{tfd_student_t_process()}, \texttt{tfd_student_t()}, \texttt{tfd_transformed_distribution()}, \texttt{tfd_triangular()}, \texttt{tfd_truncated_cauchy()}, \texttt{tfd_truncated_normal()}, \texttt{tfd_uniform()}, \texttt{tfd_vector_diffeomixture()}.
**tfd_vector_deterministic**

*Vector Deterministic Distribution*

**Description**

The VectorDeterministic distribution is parameterized by a batch point `loc` in $\mathbb{R}^k$. The distribution is supported at this point only, and corresponds to a random variable that is constant, equal to `loc`.

**Usage**

```r
tfd_vector_deterministic(
  loc,
  atol = NULL,
  rtol = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VectorDeterministic"
)
```

**Arguments**

- `loc` (Numeric Tensor of shape `[B1, ..., Bb, k]`, with `b >= 0, k >= 0`) The point (or batch of points) on which this distribution is supported.
- `atol` (Non-negative Tensor of same dtype as `loc` and broadcastable shape) The absolute tolerance for comparing closeness to `loc`. Default is 0.
- `rtol` (Non-negative Tensor of same dtype as `loc` and broadcastable shape) The relative tolerance for comparing closeness to `loc`. Default is 0.
- `validate_args` (Logical, default FALSE) When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` (Logical, default TRUE) When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- `name` (name prefixed to Ops created by this class)

**Details**

See [Degenerate rv](#).
Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

---

tfd_vector_diffeomixture

VectorDiffeomixture distribution

Description

A vector diffeomixture (VDM) is a distribution parameterized by a convex combination of \( K \) component loc vectors, \( loc[k], k = 0, ..., K-1 \), and \( K \) scale matrices \( scale[k], k = 0, ..., K-1 \). It approximates the following compound distribution:

\[
p(x) = \int p(x | z) p(z) \, dz,
\]

where \( z \) is in the \( K \)-simplex, and

\[
p(x | z) := p(x | \text{loc} = \sum_k z[k] \cdot \text{loc}[k], \text{scale} = \sum_k z[k] \cdot \text{scale}[k])
\]

Usage

```r
tfd_vector_diffeomixture(
  mix_loc, temperature, distribution, loc = NULL, scale = NULL,
  quadrature_size = 8, quadrature_fn = tfp$distributions$quadrature_scheme_softmaxnormal_quantiles,
  validate_args = FALSE, allow_nan_stats = TRUE, name = "VectorDiffeomixture"
)
```

Arguments

- `mix_loc` float-like Tensor with shape \([b_1, ..., b_B, K-1]\). In terms of samples, larger `mix_loc[...[,k]] \Rightarrow Z` is more likely to put more weight on its \( k \)th component.
- `temperature` float-like Tensor. Broadcastable with `mix_loc`. In terms of samples, smaller temperature means one component is more likely to dominate. I.e., smaller temperature makes the VDM look more like a standard mixture of \( K \) components.
- `distribution` tfp$distributions$Distribution-like instance. Distribution from which iid samples are used as input to the selected affine transformation. Must be a scalar-batch, scalar-event distribution. Typically `distribution$reparameterization_type` = `FULLY_REPARAMETERIZED` or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorDiffeomixture sample and the distribution
is not FULLY_REPARAMETERIZED yet is a function of trainable variables, then the gradient will be incorrect!

**loc**
Length-K list of float-type Tensors. The k-th element represents the shift used for the k-th affine transformation. If the k-th item is NULL, loc is implicitly \(0\). When specified, must have shape \([B_1, \ldots, B_b, d]\) where \(b \geq 0\) and \(d\) is the event size.

**scale**
Length-K list of LinearOperators. Each should be positive-definite and operate on a d-dimensional vector space. The k-th element represents the scale used for the k-th affine transformation. LinearOperators must have shape \([B_1, \ldots, B_b, d, d]\), \(b \geq 0\), i.e., characterizes b-batches of d x d matrices.

**quadrature_size**
integer scalar representing number of quadrature points. Larger quadrature_size means \(q_N(x)\) better approximates \(p(x)\).

**quadrature_fn**
Function taking \(normal_loc, normal_scale, quadrature_size, validate_args\) and returning tuple(grid,probs) representing the SoftmaxNormal grid and corresponding normalized weight. normalized) weight. Default value: quadrature_scheme_softmaxnormal_quantiles.

**validate_args**
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

**allow_nan_stats**
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

**name**
name prefixed to Ops created by this class.

**Details**

The integral \(\int p(x \mid z) p(z) \, dz\) is approximated with a quadrature scheme adapted to the mixture density \(p(z)\). The \(N\) quadrature points \(z_{N, n}\) and weights \(w_{N, n}\) (which are non-negative and sum to 1) are chosen such that \(q_N(x) := \sum_{n=1}^N w_{N, n} \, p(x \mid z_{N, n}) \rightarrow p(x)\) as \(N \rightarrow \infty\).

Since \(q_N(x)\) is in fact a mixture (of \(N\) points), we may sample from \(q_N\) exactly. It is important to note that the VDM is defined as \(q_N\) above, and not \(p(x)\). Therefore, sampling and pdf may be implemented as exact (up to floating point error) methods.

A common choice for the conditional \(p(x \mid z)\) is a multivariate Normal. The implemented marginal \(p(z)\) is the SoftmaxNormal, which is a \(K\)-1 dimensional Normal transformed by a SoftmaxCentered bijector, making it a density on the \(K\)-simplex. That is, \(Z = \text{SoftmaxCentered}(X), X = \text{Normal}(\text{mix_loc} / \text{temperature}, 1 / \text{temperature})\)

The default quadrature scheme chooses \(z_{N, n}\) as \(N\) midpoints of the quantiles of \(p(z)\) (generalized quantiles if \(K > 2\)). See Dillon and Langmore (2018) for more details.

About Vector distributions in TensorFlow.

The \texttt{VectorDiffeomixture} is a non-standard distribution that has properties particularly useful in variational Bayesian methods. Conditioned on a draw from the SoftmaxNormal, \(x \mid z\) is a vector whose components are linear combinations of affine transformations, thus is itself an affine transformation.
Note: The marginals $X_{1|v}, \ldots, X_{d|v}$ are not generally identical to some parameterization of distribution. This is due to the fact that the sum of draws from distribution are not generally itself the same distribution.

About Diffeomixtures and reparameterization.

The VectorDiffeomixture is designed to be reparameterized, i.e., its parameters are only used to transform samples from a distribution which has no trainable parameters. This property is important because backprop stops at sources of stochasticity. That is, as long as the parameters are used after the underlying source of stochasticity, the computed gradient is accurate. Reparametrization means that we can use gradient-descent (via backprop) to optimize Monte-Carlo objectives. Such objectives are a finite-sample approximation of an expectation and arise throughout scientific computing.

WARNING: If you backprop through a VectorDiffeomixture sample and the "base" distribution is both: not FULLY_REPARAMETERIZED and a function of trainable variables, then the gradient is not guaranteed correct!

Value

a distribution instance.

References


See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gaussian_process(), tfd_gaussian_process_regression_model(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_log_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(), tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(), tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(), tfd_variational_gaussian_process(), tfd_vector_exponential_diag(), tfd_vector_exponential_linear_operator(), tfd_vector_laplace_diagonal(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diagonal(), tfd_von_mises_fisher().
The vector exponential distribution is defined over a subset of $\mathbb{R}^k$, and parameterized by a (batch of) length-$k$ loc vector and a (batch of) $k \times k$ scale matrix: \[ \text{covariance} = \text{scale} @ \text{scale}^\top, \] where @ denotes matrix-multiplication.

**Usage**

```
tfd_vector_exponential_diag(
  loc = NULL,
  scale_diag = NULL,
  scale_identity_multiplier = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VectorExponentialDiag"
)
```

**Arguments**

- **loc**: Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape [B1, ..., Bb, k] where b >= 0 and k is the event size.
- **scale_diag**: Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape [B1, ..., Bb, k], b >= 0, and characterizes b-batches of $k \times k$ diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.
- **scale_identity_multiplier**: Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape [B1, ..., Bb], b >= 0, and characterizes b-batches of scaled $k \times k$ identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.
- **validate_args**: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats**: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name**: name prefixed to Ops created by this class.
Details

Mathematical Details The probability density function (pdf) is defined over the image of the scale matrix + loc, applied to the positive half-space: \( \text{Supp} = \{ \text{loc} + \text{scale} \odot x : x \in \mathbb{R}^k, x_1 > 0, \ldots, x_k > 0 \} \).

On this set,

\[
\text{pdf}(y; \text{loc}, \text{scale}) = \exp(-||x||_1) / Z, \text{ for } y \in \text{Supp}
\]

\[
x = \text{inv}(\text{scale}) \odot (y - \text{loc}),
\]

\[
Z = |\text{det}(\text{scale})|,
\]

where:

- \( \text{loc} \) is a vector in \( \mathbb{R}^k \),
- \( \text{scale} \) is a linear operator in \( \mathbb{R}^{k \times k} \), \( \text{cov} = \text{scale} \odot \text{scale}.T \),
- \( Z \) denotes the normalization constant, and,
- \( ||x||_1 \) denotes the 11 norm of \( x \), \( \text{sum}_i |x_i| \). The VectorExponential distribution is a member of the location-scale family, i.e., it can be constructed as,

\[
X = (X_1, \ldots, X_k), \text{ each } X_i \sim \text{Exponential}(rate=1)
\]

\[
Y = (Y_1, \ldots, Y_k) = \text{scale} \odot X + \text{loc}
\]

About VectorExponential and Vector distributions in TensorFlow.

The VectorExponential is a non-standard distribution that has useful properties. The marginals \( Y_1, \ldots, Y_k \) are \textit{not} Exponential random variables, due to the fact that the sum of Exponential random variables is not Exponential. Instead, \( Y \) is a vector whose components are linear combinations of Exponential random variables. Thus, \( Y \) lives in the vector space generated by vectors of Exponential distributions. This allows the user to decide the mean and covariance (by setting \( \text{loc} \) and \( \text{scale} \)), while preserving some properties of the Exponential distribution. In particular, the tails of \( Y_i \) will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of \( Y_i \) is the convolution of the pdf of \( k \) independent Exponential random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

The batch_shape is the broadcast shape between \( \text{loc} \) and \( \text{scale} \) arguments. The event_shape is given by last dimension of the matrix implied by \( \text{scale} \). The last dimension of \( \text{loc} \) (if provided) must broadcast with this. Recall that \( \text{covariance} = 2 \odot \text{scale} \odot \text{scale}.T \). Additional leading dimensions (if any) will index batches. If both \( \text{scale}_\text{diag} \) and \( \text{scale}_\text{identity\_multiplier} \) are NULL, then \( \text{scale} \) is the Identity matrix.

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd\_sample()}, \texttt{tfd\_log\_prob()}, \texttt{tfd\_mean()}.

Other distributions: \texttt{tfd\_autoregressive()}, \texttt{tfd\_batch\_reshape()}, \texttt{tfd\_bates()}, \texttt{tfd\_bernoulli()}, \texttt{tfd\_beta\_binomial()}, \texttt{tfd\_beta()}, \texttt{tfd\_binomial()}, \texttt{tfd\_categorical()}, \texttt{tfd\_cauchy()}, \texttt{tfd\_chi2()}, \texttt{tfd\_chi()}, \texttt{tfd\_cholesky\_lkj()}, \texttt{tfd\_continuous\_bernoulli()}, \texttt{tfd\_deterministic()}, \texttt{tfd\_dirichlet\_multinomial}.
The vectorization of the Exponential distribution on $\mathbb{R}^k$

**Description**

The vector exponential distribution is defined over a subset of $\mathbb{R}^k$, and parameterized by a (batch of) length-$k$ loc vector and a (batch of) $k \times k$ scale matrix: covariance = scale @ scale.T, where @ denotes matrix-multiplication.

**Usage**

```python
tfd_vector_exponential_linear_operator(  
    loc = NULL,  
    scale = NULL,  
    validate_args = FALSE,  
    allow_nan_stats = TRUE,  
    name = "VectorExponentialLinearOperator"
)
```

**Arguments**

- **loc**: Floating point tensor; the means of the distribution(s).
- **scale**: Instance of LinearOperator with same dtype as loc and shape [B1, ..., Bb, k, k].
Mathematical Details

The probability density function (pdf) is

$$
pdf(y; \text{loc}, \text{scale}) = \exp(-||x||_1) / Z, \text{ for } y \in S(\text{loc}, \text{scale}),
$$

$$
x = \text{inv}(\text{scale}) @ (y - \text{loc}),
$$

$$
Z = |\text{det}(\text{scale})|,
$$

where:

- \text{loc} is a vector in \(\mathbb{R}^k\),
- \text{scale} is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.\text{T}\),
- \(S = \{\text{loc} + \text{scale} @ x : x \in \mathbb{R}^k, x_1 > 0, \ldots, x_k > 0\}\), is an image of the positive half-space,
- \(||x||_1\) denotes the \(l_1\) norm of \(x\), \(\sum_i |x_i|\),
- \(Z\) denotes the normalization constant.

The \text{VectorExponential} distribution is a member of the \text{location-scale family}, i.e., it can be constructed as,

$$
X = (X_1, \ldots, X_k), \text{ each } X_i \sim \text{Exponential}(rate=1)
$$

$$
Y = (Y_1, \ldots, Y_k) = \text{scale} @ X + \text{loc}
$$

About \text{VectorExponential} and \text{Vector} distributions in TensorFlow.

The \text{VectorExponential} is a non-standard distribution that has useful properties. The marginals \(Y_1, \ldots, Y_k\) are \textit{not} \text{Exponential} random variables, due to the fact that the sum of \text{Exponential} random variables is not \text{Exponential}. Instead, \(Y\) is a vector whose components are linear combinations of \text{Exponential} random variables. Thus, \(Y\) lives in the vector space generated by vectors of \text{Exponential} distributions. This allows the user to decide the mean and covariance (by setting \text{loc} and \text{scale}), while preserving some properties of the \text{Exponential} distribution. In particular, the tails of \(Y_i\) will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of \(Y_i\) is the convolution of the pdf of \(k\) independent \text{Exponential} random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

The \text{batch_shape} is the broadcast shape between \text{loc} and \text{scale} arguments. The \text{event_shape} is given by last dimension of the matrix implied by \text{scale}. The last dimension of \text{loc} (if provided) must broadcast with this. Recall that \(\text{covariance} = 2 * \text{scale} @ \text{scale}.\text{T}\). Additional leading dimensions (if any) will index batches.

* @param loc Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape \([B1, \ldots, Bb, k]\) where \(b \geq 0\) and \(k\) is the event size.
tfd_vector_laplace_diag

Value
a distribution instance.

See Also
For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet()`, `tfd_dirichlet_multinomial()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gamma_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature_compound()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart_tril()`, `tfd_wishart()`, `tfd_zipf()`

---

tfd_vector_laplace_diag

The vectorization of the Laplace distribution on \(\mathbb{R}^k\)

Description
The vector laplace distribution is defined over \(\mathbb{R}^k\), and parameterized by a (batch of) length-k loc vector (the means) and a (batch of) k x k scale matrix: covariance = 2 * scale @ scale.T, where @ denotes matrix-multiplication.

Usage
```python
tfd_vector_laplace_diag(
    loc = NULL,
    scale_diag = NULL,
    scale_identity_multiplier = NULL,
)```

validate_args = FALSE,
allow_nan_stats = TRUE,
name = "VectorLaplaceDiag"
)

Arguments

loc
Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified,
may have shape [B1,...,Bb,k] where b >= 0 and k is the event size.

scale_diag
Non-zero, floating-point Tensor representing a diagonal matrix added to scale.
May have shape [B1,...,Bb,k], b >= 0, and characterizes b-batches of k x k diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

scale_identity_multiplier
Non-zero, floating-point Tensor representing a scaled-identity-matrix added to scale. May have shape [B1,...,Bb,k], b >= 0, and characterizes b-batches of scaled k x k identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

validate_args
Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details The probability density function (pdf) is,

\[
pdf(x; \text{loc}, \text{scale}) = \exp(-||y||_1) / Z
\]
\[
y = \text{inv}(\text{scale}) @ (x - \text{loc})
\]
\[
Z = 2^{*k} |\text{det}(\text{scale})|
\]

where:

- loc is a vector in \(\mathbb{R}^k\),
- scale is a linear operator in \(\mathbb{R}^{k \times k}\), \(\text{cov} = \text{scale} @ \text{scale}.T\),
- \(Z\) denotes the normalization constant, and,
- \(||y||_1\) denotes the \(1\) norm of \(y\), \('\sum_i |y_i|\).

A (non-batch) scale matrix is:

\[
\text{scale} = \text{diag}(\text{scale}_\text{diag} + \text{scale}_\text{identity}_\text{multiplier} * \text{ones}(k))
\]

where:
• scale_diag.shape = [k], and,
• scale_identity_multiplier.shape = []. Additional leading dimensions (if any) will index batches.
If both scale_diag and scale_identity_multiplier are NULL, then scale is the Identity matrix.

About VectorLaplace and Vector distributions in TensorFlow

The VectorLaplace is a non-standard distribution that has useful properties. The marginals Y_1, ..., Y_k are not Laplace random variables, due to the fact that the sum of Laplace random variables is not Laplace. Instead, Y is a vector whose components are linear combinations of Laplace random variables. Thus, Y lives in the vector space generated by vectors of Laplace distributions. This allows the user to decide the mean and covariance (by setting loc and scale), while preserving some properties of the Laplace distribution. In particular, the tails of Y_i will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of Y_i is the convolution of the pdf of k independent Laplace random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(),
tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(),
tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial,
tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma_gamma(), tfd_gaussian_process(), tfd_gaussian_process_regression_model(),
tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(),
tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space,
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(),
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(),
tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature,
tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(), tfd_sample_distribution(), tfd_sinh_arcsinh(),
tfd_skellam(), tfd_spherical_uniform(), tfd_student_t_process(), tfd_student_t(),
tfd_transformed_distribution(), tfd_triangular(), tfd_truncated_cauchy(), tfd_truncated_normal(), tfd_uniform(),
tfd_variational_gaussian_process(), tfd_vector_diffeomixture(), tfd_vector_exponential_diag(),
tfd_vector_exponential_linear_operator(), tfd_vector_laplace_linear_operator(), tfd_vector_sinh_arcsinh_diag(),
tfd_von_mises_fisher(), tfd_von_mises(), tfd_weibull(), tfd_wishart_linear_operator(), tfd_wishart_tri_l(),
tfd_wishart(), tfd_zipf()
The vectorization of the Laplace distribution on $\mathbb{R}^k$

**Description**

The vector laplace distribution is defined over $\mathbb{R}^k$, and parameterized by a (batch of) length-$k$ loc vector (the means) and a (batch of) $k \times k$ scale matrix: $\text{covariance} = 2 \ast \text{scale} \otimes \text{scale}.^T$, where $\otimes$ denotes matrix-multiplication.

**Usage**

```python
  tfd_vector_laplace_linear_operator(  
    loc = NULL,  
    scale = NULL,  
    validate_args = FALSE,  
    allow_nan_stats = TRUE,  
    name = "VectorLaplaceLinearOperator"
  )
```

**Arguments**

- **loc** Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape $[B_1, ..., B_b, k]$ where $b \geq 0$ and $k$ is the event size.
- **scale** Instance of LinearOperator with same dtype as loc and shape $[B_1, ..., B_b, k, k]$.
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats** Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name** name prefixed to Ops created by this class.

**Details**

Mathematical Details The probability density function (pdf) is,

$$
\text{pdf}(x; \text{loc}, \text{scale}) = \exp(-||y||_1) / Z, \\
y = \text{inv}(\text{scale}) \otimes (x - \text{loc}), \\
Z = 2^{**k} |\text{det}(\text{scale})|,
$$

where:

- **loc** is a vector in $\mathbb{R}^k$,
- **scale** is a linear operator in $\mathbb{R}^{[k \times k]}$, $\text{cov} = \text{scale} \otimes \text{scale}.^T$,
The VectorLaplace distribution is a member of the location-scale family, i.e., it can be constructed as,

\[ X = (X_1, \ldots, X_k), \text{ each } X_i \sim \text{Laplace}(\text{loc}=0, \text{scale}=1) \]

\[ Y = (Y_1, \ldots, Y_k) = \text{scale} \odot X + \text{loc} \]

About VectorLaplace and Vector distributions in TensorFlow

The VectorLaplace is a non-standard distribution that has useful properties. The marginals \( Y_1, \ldots, Y_k \) are not Laplace random variables, due to the fact that the sum of Laplace random variables is not Laplace. Instead, \( Y \) is a vector whose components are linear combinations of Laplace random variables. Thus, \( Y \) lives in the vector space generated by vectors of Laplace distributions. This allows the user to decide the mean and covariance (by setting loc and scale), while preserving some properties of the Laplace distribution. In particular, the tails of \( Y_i \) will be (up to polynomial factors) exponentially decaying. To see this last statement, note that the pdf of \( Y_i \) is the convolution of the pdf of \( k \) independent Laplace random variables. One can then show by induction that distributions with exponential (up to polynomial factors) tails are closed under convolution.

The batch_shape is the broadcast shape between loc and scale arguments. The event_shape is given by last dimension of the matrix implied by scale. The last dimension of loc (if provided) must broadcast with this. Recall that \[ \text{covariance} = 2 \times \text{scale} \odot \text{scale}^T. \] Additional leading dimensions (if any) will index batches.

Value

a distribution instance.

See Also

For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chi2()}, \texttt{tfd_chi()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exp_gamma()}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma_gamma()}, \texttt{tfd_gamma()}, \texttt{tfd_gaussian_process_regression_model()}, \texttt{tfd_gaussian_process()}, \texttt{tfd_generalized_normal()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space()}, \texttt{tfd_lkj()}, \texttt{tfd_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal()}, \texttt{tfd_multivariate_normal_full_c covariance()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_normal_tril()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature()}, \texttt{tfd_poisson()}, \texttt{tfd_power_spherical()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_relaxed_one_hot_categorical()}, \texttt{tfd_sample_distribution()}, \texttt{tfd_sinh_arcsinh()}, \texttt{tfd_student_t_population_stddev()}.
The (diagonal) SinhArcsinh transformation of a distribution on $\mathbb{R}^k$

**Description**

This distribution models a random vector $Y = (Y_1, ..., Y_k)$, making use of a SinhArcsinh transformation (which has adjustable tailweight and skew), a rescaling, and a shift. The SinhArcsinh transformation of the Normal is described in great depth in Sinh-arcsinh distributions. Here we use a slightly different parameterization, in terms of tailweight and skewness. Additionally we allow for distributions other than Normal, and control over scale as well as a "shift" parameter loc.

**Usage**

```r
tfd_vector_sinh_arcsinh_diag(
  loc = NULL,
  scale_diag = NULL,
  scale_identity_multiplier = NULL,
  skewness = NULL,
  tailweight = NULL,
  distribution = NULL,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VectorSinhArcsinhDiag"
)
```

**Arguments**

- **loc**
  Floating-point Tensor. If this is set to NULL, loc is implicitly 0. When specified, may have shape $[B_1, ..., B_b, k]$ where $b \geq 0$ and $k$ is the event size.

- **scale_diag**
  Non-zero, floating-point Tensor representing a diagonal matrix added to scale. May have shape $[B_1, ..., B_b, k]$, $b \geq 0$, and characterizes $b$-batches of $k \times k$ diagonal matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.

- **scale_identity_multiplier**
  Non-zero, floating-point Tensor representing a scale-identity-matrix added to scale. May have shape $[B_1, ..., B_b]$, $b \geq 0$, and characterizes $b$-batches of scale $k \times k$ identity matrices added to scale. When both scale_identity_multiplier and scale_diag are NULL then scale is the Identity.
skewness  Skewness parameter. floating-point Tensor with shape broadcastable with event_shape.
tailweight  Tailweight parameter. floating-point Tensor with shape broadcastable with event_shape.
distribution  tf$\textit{distributions}$Distribution-like instance. Distribution from which k iid samples are used as input to transformation F. Default is tfd\_normal(loc = 0, scale = 1). Must be a scalar-batch, scalar-event distribution. Typically distribution$reparameterization\_type = \text{FULLY\_REPARAMETERIZED}$ or it is a function of non-trainable parameters. WARNING: If you backprop through a VectorSinhArcsinhDiag sample and distribution is not FULLY\_REPARAMETERIZED yet is a function of trainable variables, then the gradient will be incorrect!
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name  name prefixed to Ops created by this class.

Details

Mathematical Details

Given iid random vector $Z = (Z_1, \ldots, Z_k)$, we define the VectorSinhArcsinhDiag transformation of $Z$, $Y$, parameterized by $(\text{loc}, \text{scale}, \text{skewness}, \text{tailweight})$, via the relation (with @ denoting matrix multiplication):

\[
Y := \text{loc} + \text{scale} @ F(Z) * (2 / F_0(2))
\]

\[
F(Z) := \text{Sinh}( (\text{Arcsinh}(Z) + \text{skewness}) * \text{tailweight} )
\]

\[
F_0(Z) := \text{Sinh}( \text{Arcsinh}(Z) * \text{tailweight} )
\]

This distribution is similar to the location-scale transformation $L(Z) := \text{loc} + \text{scale} @ Z$ in the following ways:

- If skewness = 0 and tailweight = 1 (the defaults), $F(Z) = Z$, and then $Y = L(Z)$ exactly.
- loc is used in both to shift the result by a constant factor.
- The multiplication of scale by $2 / F_0(Z)$ ensures that if skewness = 0 $P[Y - \text{loc} \leq 2 * \text{scale}] = P[L(Z) - \text{loc} \leq 2 * \text{scale}]$. Thus it can be said that the weights in the tails of $Y$ and $L(Z)$ beyond $\text{loc} + 2 * \text{scale}$ are the same. This distribution is different than $\text{loc} + \text{scale} @ Z$ due to the reshaping done by $F$:
  - Positive (negative) skewness leads to positive (negative) skew.
  - positive skew means, the mode of $F(Z)$ is "tilted" to the right.
  - positive skew means positive values of $F(Z)$ become more likely, and negative values become less likely.
  - Larger (smaller) tailweight leads to fatter (thinner) tails.
  - Fatter tails mean larger values of $|F(Z)|$ become more likely.
  - tailweight < 1 leads to a distribution that is "flat" around $Y = \text{loc}$, and a very steep drop-off in the tails.
- tailweight > 1 leads to a distribution more peaked at the mode with heavier tails. To see the argument about the tails, note that for |Z| >> 1 and |Z| >> (|skewness| * tailweight)**tailweight, we have Y approx 0.5 Z**tailweight e**(sign(Z) skewness * tailweight). To see the argument regarding multiplying scale by 2 / F_0(2),

\[
P[(Y - loc) / scale <= 2] = P[F(Z) * (2 / F_0(2)) <= 2]
= P[F(Z) <= F_0(2)]
= P[Z <= 2] (if F = F_0).
\]

**Value**

a distribution instance.

**See Also**

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisq()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`, `tfd_gamma()`, `tfd_gammalognormal()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`, `tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`, `tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gaussian()`, `tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`, `tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`, `tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space_model()`, `tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`, `tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal()`, `tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`, `tfd_multivariate_normal_tri_l()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`, `tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature()`, `tfd_poisson()`, `tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`, `tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`, `tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t_process()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_von_mises_fisher()`, `tfd_von_mises()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart()`, `tfd_zipf()`
Description

The von Mises distribution is a univariate directional distribution. Similarly to Normal distribution, it is a maximum entropy distribution. The samples of this distribution are angles, measured in radians. They are 2 pi-periodic: x = 0 and x = 2pi are equivalent. This means that the density is also 2 pi-periodic. The generated samples, however, are guaranteed to be in [-pi, pi) range. When concentration = 0, this distribution becomes a Uniform distribution on the [-pi, pi) domain.

Usage

tfd_von_mises(
  loc,
  concentration,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "VonMises"
)

Arguments

loc
  Floating point tensor, the circular means of the distribution(s).

concentration
  Floating point tensor, the level of concentration of the distribution(s) around loc. Must take non-negative values. concentration = 0 defines a Uniform distribution, while concentration = +inf indicates a Deterministic distribution at loc.

validate_args
  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats
  Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name
  name prefixed to Ops created by this class.

Details

The von Mises distribution is a special case of von Mises-Fisher distribution for n=2. However, the TFP’s VonMisesFisher implementation represents the samples and location as (x, y) points on a circle, while VonMises represents them as scalar angles.

Mathematical details The probability density function (pdf) of this distribution is,

pdf(x; loc, concentration) = exp(concentration cos(x - loc)) / Z
Z = 2 * pi * I_0 (concentration)

where:

• I_0 (concentration) is the modified Bessel function of order zero;
• loc the circular mean of the distribution, a scalar. It can take arbitrary values, but it is 2pi-periodic: loc and loc + 2pi result in the same distribution.
• concentration >= 0 parameter is the concentration parameter. When concentration = 0,
this distribution becomes a Uniform distribution on [-pi, pi).

The parameters loc and concentration must be shaped in a way that supports broadcasting (e.g. loc + concentration is a valid operation).

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`,
`tfd_beta_binomial()`, `tfd_beta()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chi2()`,
`tfd_chol()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_deterministic()`, `tfd_dirichlet_multinomial()`,
`tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exponential()`,
`tfd_gamma_gamma()`, `tfd_gamma()`, `tfd_gaussian_process_regression_model()`, `tfd_gaussian_process()`,
`tfd_generalized_normal()`, `tfd_geometric()`, `tfd_gumbel()`, `tfd_half_cauchy()`, `tfd_half_normal()`,
`tfd_hidden_markov_model()`, `tfd_horseshoe()`, `tfd_independent()`, `tfd_inverse_gamma()`,
`tfd_inverse_gaussian()`, `tfd_johnson_s_u()`, `tfd_joint_distribution_named_auto_batched()`,
`tfd_joint_distribution_named()`, `tfd_joint_distribution_sequential_auto_batched()`,
`tfd_joint_distribution_sequential()`, `tfd_kumaraswamy()`, `tfd_laplace()`, `tfd_linear_gaussian_state_space()`,
`tfd_lkj()`, `tfd_logistic()`, `tfd_log_normal()`, `tfd_logistic()`, `tfd_mixture_same_family()`,
`tfd_mixture()`, `tfd_multinomial()`, `tfd_multivariate_normal_diag_plus_low_rank()`, `tfd_multivariate_normal_diag()`,
`tfd_multivariate_normal_full_covariance()`, `tfd_multivariate_normal_linear_operator()`,
`tfd_multivariate_normal_tril()`, `tfd_multivariate_student_t_linear_operator()`, `tfd_negative_binomial()`,
`tfd_normal()`, `tfd_one_hot_categorical()`, `tfd_pareto()`, `tfd_pixel_cnn()`, `tfd_poisson_log_normal_quadrature()`,
`tfd_power_spherical()`, `tfd_probit_bernoulli()`, `tfd_quantized()`, `tfd_relaxed_bernoulli()`,
`tfd_relaxed_one_hot_categorical()`, `tfd_sample_distribution()`, `tfd_sinh_arcsinh()`,
`tfd_skellam()`, `tfd_spherical_uniform()`, `tfd_student_t()`, `tfd_transformed_distribution()`,
`tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_variational_gaussian_bernoulli()`,
`tfd_vector_diffeomixture()`, `tfd_vector_exponential_diag()`, `tfd_vector_exponential_linear_operator()`,
`tfd_vector_laplace_diag()`, `tfd_vector_laplace_linear_operator()`, `tfd_vector_sinh_arcsinh_diag()`,
`tfd_von_mises_fisher()`, `tfd_weibull()`, `tfd_wishart_linear_operator()`, `tfd_wishart_tri_l()`,
`tfd_wishart()`, `tfd_zipf()`

**tfd_von_mises_fisher**

The von Mises-Fisher distribution over unit vectors on $S^{n-1}$

Description

The von Mises-Fisher distribution is a directional distribution over vectors on the unit hypersphere $S^{n-1}$ embedded in n dimensions ($R^n$).
Usage

tfd_von_mises_fisher(
    mean_direction,
    concentration,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "VonMisesFisher"
)

Arguments

mean_direction Floating-point Tensor with shape [B1, ... Bn, D]. A unit vector indicating the mode of the distribution, or the unit-normalized direction of the mean. (This is not in general the mean of the distribution; the mean is not generally in the support of the distribution.) NOTE: D is currently restricted to <= 5.

concentration Floating-point Tensor having batch shape [B1, ... Bn] broadcastable with mean_direction. The level of concentration of samples around the mean_direction. concentration=0 indicates a uniform distribution over the unit hypersphere, and concentration=+inf indicates a Deterministic distribution (delta function) at mean_direction.

validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE an exception is raised if one or more of the statistic’s batch members are undefined.

name name prefixed to Ops created by this class.

Details

Mathematical details The probability density function (pdf) is,

\[
pdf(x; \mu, \kappa) = C(\kappa) \exp(\kappa \cdot \mu^T x)
\]

where,

\[
C(\kappa) = (2 \pi)^{-n/2} \kappa^{n/2-1} / I_{n/2-1}(\kappa),
\]

\[I_n(z)\] being the modified Bessel function of the first kind of order \(v\)

where:

- mean_direction = \(\mu\); a unit vector in \(\mathbb{R}^k\),
- concentration = \(\kappa\); scalar real >= 0, concentration of samples around mean_direction, where 0 pertains to the uniform distribution on the hypersphere, and inf indicates a delta function at mean_direction.

NOTE: Currently only n in 2, 3, 4, 5 are supported. For n=5 some numerical instability can occur for low concentrations (<.01).
Value

A distribution instance.

See Also


---

tfd_weibull

The Weibull distribution with 'concentration' and scale parameters.

Description

The probability density function (pdf) of this distribution is,

\[ \text{pdf}(x; \lambda, k) = k / \lambda \times (x / \lambda)^{k-1} \times \exp(-(x / \lambda)^k) \]

where concentration = k and scale = \lambda. The cumulative density function of this distribution is,

\[ \text{cdf}(x; \lambda, k) = 1 - \exp(-(x / \lambda)^k) \]

The Weibull distribution includes the Exponential and Rayleigh distributions as special cases:

Exponential(rate) = Weibull(concentration=1., 1. / rate)

Rayleigh(scale) = Weibull(concentration=2., sqrt(2.) * scale)
Usage

tfd_weibull(
    concentration,
    scale,
    validate_args = FALSE,
    allow_nan_stats = TRUE,
    name = "Weibull"
)

Arguments

concentration  Positive Float-type Tensor, the concentration param of the distribution. Must contain only positive values.
scale          Positive Float-type Tensor, the scale param of the distribution. Must contain only positive values.
validate_args  Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
name           name prefixed to Ops created by this class.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chi2(), tfd_chi(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(), tfd_gamma(), tfd_gammas(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(), tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(), tfd_inverse_gaussian(), tfd_johnson_s_u(), tfd_joint_distribution_named_auto_batched(), tfd_joint_distribution_named(), tfd_joint_distribution_sequential_auto_batched(), tfd_joint_distribution_sequential(), tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(), tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(), tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(), tfd_multivariate_normal_diag(), tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(), tfd_multivariate_normal_tri_l(), tfd_multivariate_student_t_linear_operator(), tfd_negative_binomial(), tfd_normal(), tfd_one_hot_categorical(), tfd_pareto(), tfd_pixel_cnn(), tfd_poisson_log_normal_quadrature(), tfd_poisson(), tfd_power_spherical(), tfd_probit_bernoulli(), tfd_quantized(), tfd_relaxed_bernoulli(), tfd_student_t_distribution(), tfd_transformed_distribution(), tfd_truncated_normal(), tfd_uniform(), tfd_vector_bernoulli(), tfd_von_mises_fisher(), tfd_wishart().
The matrix Wishart distribution on positive definite matrices

This distribution is defined by a scalar number of degrees of freedom df and an instance of LinearOperator, which provides matrix-free access to a symmetric positive definite operator, which defines the scale matrix.

Usage

tfd_wishart(
  df,
  scale = NULL,
  scale_tril = NULL,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "Wishart"
)

Arguments

df float or double tensor, the degrees of freedom of the distribution(s). df must be greater than or equal to k.

scale float or double Tensor. The symmetric positive definite scale matrix of the distribution. Exactly one of scale and 'scale_tril must be passed.

scale_tril float or double Tensor. The Cholesky factorization of the symmetric positive definite scale matrix of the distribution. Exactly one of scale and 'scale_tril must be passed.

input_output_cholesky Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and stddev methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
validate_args: Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.

allow_nan_stats: Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.

name: name prefixed to Ops created by this class.

Details:
Mathematical Details
The probability density function (pdf) is,

$$pdf(X; df, scale) = \frac{\det(X)^{0.5(df-k-1)} \exp(-0.5 \text{tr}[\text{inv}(scale) X])}{Z}$$

$$Z = 2^{0.5 df k} |\det(scale)|^{0.5 df} \Gamma_k(0.5 df)$$

where:
- \(df \geq k\) denotes the degrees of freedom,
- \(scale\) is a symmetric, positive definite, \(k \times k\) matrix,
- \(Z\) is the normalizing constant, and,
- \(\Gamma_k\) is the \text{multivariate Gamma function}.

Value:
a distribution instance.

See Also:
For usage examples see e.g. \texttt{tfd_sample()}, \texttt{tfd_log_prob()}, \texttt{tfd_mean()}.

Other distributions: \texttt{tfd_autoregressive()}, \texttt{tfd_batch_reshape()}, \texttt{tfd_bates()}, \texttt{tfd_bernoulli()}, \texttt{tfd_beta_binomial()}, \texttt{tfd_beta()}, \texttt{tfd_binomial()}, \texttt{tfd_categorical()}, \texttt{tfd_cauchy()}, \texttt{tfd_chisquare()}, \texttt{tfd_cholesky_lkj()}, \texttt{tfd_continuous_bernoulli()}, \texttt{tfd_deterministic()}, \texttt{tfd_dirichlet_multinomial()}, \texttt{tfd_dirichlet()}, \texttt{tfd_empirical()}, \texttt{tfd_exp}, \texttt{tfd_exp_inverse_gamma()}, \texttt{tfd_exponential()}, \texttt{tfd_gamma()}, \texttt{tfd_bernoulli()}, \texttt{tfd_generalized_normal()}, \texttt{tfd_geometric()}, \texttt{tfd_gumbel()}, \texttt{tfd_half_cauchy()}, \texttt{tfd_half_normal()}, \texttt{tfd_hidden_markov_model()}, \texttt{tfd_horseshoe()}, \texttt{tfd_independent()}, \texttt{tfd_inverse_gamma()}, \texttt{tfd_inverse_gaussian()}, \texttt{tfd_johnson_s_u()}, \texttt{tfd_joint_distribution_named_auto_batched()}, \texttt{tfd_joint_distribution_named()}, \texttt{tfd_joint_distribution_sequential_auto_batched()}, \texttt{tfd_joint_distribution_sequential()}, \texttt{tfd_kumaraswamy()}, \texttt{tfd_laplace()}, \texttt{tfd_linear_gaussian_state_space_model()}, \texttt{tfd_lkj()}, \texttt{tfd_logistic()}, \texttt{tfd_log_normal()}, \texttt{tfd_logistic()}, \texttt{tfd_mixture_same_family()}, \texttt{tfd_mixture()}, \texttt{tfd_multinomial()}, \texttt{tfd_multivariate_normal_diag_plus_low_rank()}, \texttt{tfd_multivariate_normal_full_covariance()}, \texttt{tfd_multivariate_normal_linear_operator()}, \texttt{tfd_multivariate_normal_tri_l()}, \texttt{tfd_multivariate_student_t_linear_operator()}, \texttt{tfd_negative_binomial()}, \texttt{tfd_normal()}, \texttt{tfd_one_hot_categorical()}, \texttt{tfd_pareto()}, \texttt{tfd_pixel_cnn()}, \texttt{tfd_poisson_log_normal_quadrature_compound()}, \texttt{tfd_power_spherical()}, \texttt{tfd_probit_bernoulli()}, \texttt{tfd_quantized()}, \texttt{tfd_relaxed_bernoulli()}, \texttt{tfd_sigmoid()}, \texttt{tfd_student_t()}. \texttt{tfd_transformed_distribution()}. \texttt{tfd_tril()}. \texttt{tfd_uniform()}. \texttt{tfd_von_mises_fisher()}. \texttt{tfd_von_mises()}.
tfd_wishart_linear_operator

The matrix Wishart distribution on positive definite matrices

Description

This distribution is defined by a scalar number of degrees of freedom df and an instance of LinearOperator, which provides matrix-free access to a symmetric positive definite operator, which defines the scale matrix.

Usage

tfd_wishart_linear_operator(
  df,
  scale,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "WishartLinearOperator"
)

Arguments

df float or double tensor, the degrees of freedom of the distribution(s). df must be greater than or equal to k.
scale float or double instance of LinearOperator.
input_output_cholesky Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and stddev methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
validate_args Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
allow_nan_stats
Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use
the value NaN to indicate the result is undefined. When FALSE, an exception is
raised if one or more of the statistic’s batch members are undefined.

name
name prefixed to Ops created by this class.

Details
Mathematical Details
The probability density function (pdf) is,

\[ p(X; df, scale) = \frac{\det(X)^{0.5(df-k-1)}}{Z} \exp\left(-0.5 \ tr[\text{inv}(\text{scale}) \ X]\right) \]
\[ Z = 2^{0.5(df \times k)} |\det(\text{scale})|^{0.5(df)} \Gamma_k(0.5(df)) \]

where:
- \( df \geq k \) denotes the degrees of freedom,
- \( \text{scale} \) is a symmetric, positive definite, \( k \times k \) matrix,
- \( Z \) is the normalizing constant, and,
- \( \Gamma_k \) is the multivariate Gamma function.

Value
a distribution instance.

See Also
For usage examples see e.g. \( \text{tfd}\_\text{sample}, \text{tfd}\_\text{log}\_\text{prob}, \text{tfd}\_\text{mean} \).

Other distributions: \( \text{tfd}\_\text{autoregressive}, \text{tfd}\_\text{batch}\_\text{reshape}, \text{tfd}\_\text{bates}, \text{tfd}\_\text{bernoulli}, \text{tfd}\_\text{beta}, \text{tfd}\_\text{binomial}, \text{tfd}\_\text{categorical}, \text{tfd}\_\text{cauchy}, \text{tfd}\_\text{chi}, \text{tfd}\_\text{cholesky}\_\text{lkj}, \text{tfd}\_\text{continuous}\_\text{bernoulli}, \text{tfd}\_\text{deterministic}, \text{tfd}\_\text{dirichlet}, \text{tfd}\_\text{dirichlet}\_\text{multinomial}, \text{tfd}\_\text{dirichlet}\_\text{logit}, \text{tfd}\_\text{empirical}, \text{tfd}\_\text{exp}\_\text{gamma}, \text{tfd}\_\text{exp}\_\text{inverse}\_\text{gamma}, \text{tfd}\_\text{exponential}, \text{tfd}\_\text{gamma}\_\text{gamma}, \text{tfd}\_\text{gamma}, \text{tfd}\_\text{gaussian}\_\text{process}\_\text{regression}\_\text{model}, \text{tfd}\_\text{gaussian}\_\text{process}, \text{tfd}\_\text{generalized}\_\text{normal}, \text{tfd}\_\text{geometric}, \text{tfd}\_\text{gumbel}, \text{tfd}\_\text{half}\_\text{cauchy}, \text{tfd}\_\text{half}\_\text{normal}, \text{tfd}\_\text{hidden}\_\text{markov}\_\text{model}, \text{tfd}\_\text{horseshoe}, \text{tfd}\_\text{independent}, \text{tfd}\_\text{inverse}\_\text{gamma}, \text{tfd}\_\text{inverse}\_\text{gaussian}, \text{tfd}\_\text{johnson}\_\text{s}\_\text{u}, \text{tfd}\_\text{joint}\_\text{distribution}\_\text{named}\_\text{auto}\_\text{batched}, \text{tfd}\_\text{joint}\_\text{distribution}\_\text{named}, \text{tfd}\_\text{joint}\_\text{distribution}\_\text{sequential}\_\text{auto}\_\text{batched}, \text{tfd}\_\text{joint}\_\text{distribution}\_\text{sequential}, \text{tfd}\_\text{kumaraswamy}, \text{tfd}\_\text{laplace}, \text{tfd}\_\text{linear}\_\text{gaussian}\_\text{state}\_\text{space}, \text{tfd}\_\text{lkj}, \text{tfd}\_\text{log}\_\text{logistic}, \text{tfd}\_\text{log}\_\text{normal}, \text{tfd}\_\text{logistic}, \text{tfd}\_\text{mixture}\_\text{same}\_\text{family}, \text{tfd}\_\text{mixture}, \text{tfd}\_\text{multinomial}, \text{tfd}\_\text{multivariate}\_\text{normal}\_\text{diag}\_\text{plus}\_\text{low}\_\text{rank}, \text{tfd}\_\text{multivariate}\_\text{normal}, \text{tfd}\_\text{multivariate}\_\text{normal}\_\text{full}\_\text{covariance}, \text{tfd}\_\text{multivariate}\_\text{normal}\_\text{linear}\_\text{operator}, \text{tfd}\_\text{multivariate}\_\text{normal}\_\text{tri}l, \text{tfd}\_\text{multivariate}\_\text{student}\_t\_\text{linear}\_\text{operator}, \text{tfd}\_\text{negative}\_\text{binomial}, \text{tfd}\_\text{normal}, \text{tfd}\_\text{one}\_\text{hot}\_\text{categorical}, \text{tfd}\_\text{pareto}, \text{tfd}\_\text{pixel}\_\text{cnn}, \text{tfd}\_\text{poisson}\_\text{log}\_\text{normal}\_\text{quadrature}, \text{tfd}\_\text{poisson}, \text{tfd}\_\text{power}\_\text{categorical}, \text{tfd}\_\text{power}\_\text{sigmoid}, \text{tfd}\_\text{quantize}, \text{tfd}\_\text{relaxed}\_\text{bernoulli}, \text{tfd}\_\text{relaxed}\_\text{one}\_\text{hot}\_\text{categorical}, \text{tfd}\_\text{sample}\_\text{distribution}, \text{tfd}\_\text{sinh}\_\text{ arcsinh}, \text{tfd}\_\text{skellam}, \text{tfd}\_\text{ spherical}\_\text{uniform}, \text{tfd}\_\text{student}\_t\_\text{ process}, \text{tfd}\_\text{student}\_t, \text{tfd}\_\text{transformed}\_\text{distribution}, \text{tfd}\_\text{trapezoidal}, \text{tfd}\_\text{truncated}\_\text{cauchy}, \text{tfd}\_\text{truncated}\_\text{normal}, \text{tfd}\_\text{uniform}, \text{tfd}\_\text{variational}\_\text{gamma}, \text{tfd}\_\text{vector}\_\text{diffeomixture}, \text{tfd}\_\text{vector}\_\text{exponential}\_\text{diag}, \text{tfd}\_\text{vector}\_\text{exponential}\_\text{linear}\_\text{operator}, \text{tfd}\_\text{vector}\_\text{linear}\_\text{operator}, \text{tfd}\_\text{ weibull}, \text{tfd}\_\text{wishart}, \text{tfd}\_\text{wishart}\_\text{linear}\_\text{operator}, \text{tfd}\_\text{wulf}\_\text{linear}\_\text{operator}. \]
**tfd_wishart_tri_l**

The **matrix Wishart distribution parameterized with Cholesky factors**.

---

**Description**

This distribution is defined by a scalar degrees of freedom df and a scale matrix, expressed as a lower triangular Cholesky factor.

**Usage**

```python

tfd_wishart_tri_l(
  df,
  scale_tril,
  input_output_cholesky = FALSE,
  validate_args = FALSE,
  allow_nan_stats = TRUE,
  name = "WishartTriL"
)
```

**Arguments**

- **df** float or double tensor, the degrees of freedom of the distribution(s). df must be greater than or equal to k.
- **scale_tril** float or double Tensor. The Cholesky factorization of the symmetric positive definite scale matrix of the distribution.
- **input_output_cholesky** Logical. If TRUE, functions whose input or output have the semantics of samples assume inputs are in Cholesky form and return outputs in Cholesky form. In particular, if this flag is TRUE, input to log_prob is presumed of Cholesky form and output from sample, mean, and mode are of Cholesky form. Setting this argument to TRUE is purely a computational optimization and does not change the underlying distribution; for instance, mean returns the Cholesky of the mean, not the mean of Cholesky factors. The variance and stddev methods are unaffected by this flag. Default value: FALSE (i.e., input/output does not have Cholesky semantics).
- **validate_args** Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- **allow_nan_stats** Logical, default TRUE. When TRUE, statistics (e.g., mean, mode, variance) use the value NaN to indicate the result is undefined. When FALSE, an exception is raised if one or more of the statistic’s batch members are undefined.
- **name** name prefixed to Ops created by this class.
Details

Mathematical Details

The probability density function (pdf) is,

$$pdf(X; \text{df}, \text{scale}) = \det(X)^{(0.5 \times (\text{df}-k-1))} \exp(-0.5 \ \text{tr}[\text{inv}(\text{scale}) \times X]) / Z$$

$$Z = 2^{(0.5 \ \text{df} \ k)} \ |\det(\text{scale})|^{(0.5 \ \text{df})} \ \Gamma_k(0.5 \ \text{df})$$

where:

- $\text{df} \geq k$ denotes the degrees of freedom,
- $\text{scale}$ is a symmetric, positive definite, $k \times k$ matrix,
- $Z$ is the normalizing constant, and,
- $\Gamma_k$ is the multivariate Gamma function.

Value

a distribution instance.

See Also

For usage examples see e.g. `tfd_sample()`, `tfd_log_prob()`, `tfd_mean()`.

Other distributions: `tfd_autoregressive()`, `tfd_batch_reshape()`, `tfd_bates()`, `tfd_bernoulli()`, `tfd_binomial()`, `tfd_categorical()`, `tfd_cauchy()`, `tfd_chisquare()`, `tfd_cholesky_lkj()`, `tfd_continuous_bernoulli()`, `tfd_dirichlet_multinomial()`, `tfd_dirichlet()`, `tfd_empirical()`, `tfd_exp_gamma()`, `tfd_exp_inverse_gamma()`, `tfd_exp_negative_binomial()`, `tfd_exp_normal()`, `tfd_normal()`, `tfd_student_t()`, `tfd_transformed_distribution()`, `tfd_triangular()`, `tfd_truncated_cauchy()`, `tfd_truncated_normal()`, `tfd_uniform()`, `tfd_wishart()`, `tfd_zipf()`, `tfd_wishart_linear_operator()`,
The Zipf distribution is parameterized by a power parameter.

**Usage**

```r
tfd_zipf(
  power,
  dtype = tf$int32,
  interpolate_nondiscrete = TRUE,
  sample_maximum_iterations = 100,
  validate_args = FALSE,
  allow_nan_stats = FALSE,
  name = "Zipf"
)
```

**Arguments**

- `power` Float like Tensor representing the power parameter. Must be strictly greater than 1.
- `dtype` The dtype of Tensor returned by sample. Default value: tf$int32.
- `interpolate_nondiscrete` Logical. When FALSE, log_prob returns -inf (and prob returns 0) for non-integer inputs. When TRUE, log_prob evaluates the continuous function -power log(k) - log(zeta(power)), which matches the Zipf pmf at integer arguments k (note that this function is not itself a normalized probability log-density). Default value: TRUE.
- `sample_maximum_iterations` Maximum number of iterations of allowable iterations in sample. When validate_args=TRUE, samples which fail to reach convergence (subject to this cap) are masked out with self$dtype$min or nan depending on self$dtype$is_integer. Default value: 100.
- `validate_args` Logical, default FALSE. When TRUE distribution parameters are checked for validity despite possibly degrading runtime performance. When FALSE invalid inputs may silently render incorrect outputs. Default value: FALSE.
- `allow_nan_stats` Default value: FALSE.
- `name` name prefixed to Ops created by this class.

**Details**

Mathematical Details The probability mass function (pmf) is,
pmf(k; α, k ≥ 0) = (k^(-α)) / Z
Z = ζ(α).

where power = α and Z is the normalization constant. ζ is the Riemann zeta function. Note that gradients with respect to the power parameter are not supported in the current implementation.

Value

a distribution instance.

See Also

For usage examples see e.g. tfd_sample(), tfd_log_prob(), tfd_mean().

Other distributions: tfd_autoregressive(), tfd_batch_reshape(), tfd_bates(), tfd_bernoulli(), tfd_beta_binomial(), tfd_beta(), tfd_binomial(), tfd_categorical(), tfd_cauchy(), tfd_chisq(), tfd_cholesky_lkj(), tfd_continuous_bernoulli(), tfd_deterministic(), tfd_dirichlet_multinomial(), tfd_dirichlet(), tfd_empirical(), tfd_exp_gamma(), tfd_exp_inverse_gamma(), tfd_exponential(),
tfd_gamma(), tfd_gamma(), tfd_gaussian_process_regression_model(), tfd_gaussian_process(), tfd_generalized_normal(), tfd_geometric(), tfd_gumbel(), tfd_half_cauchy(), tfd_half_normal(),
tfd_hidden_markov_model(), tfd_horseshoe(), tfd_independent(), tfd_inverse_gamma(),
tfd_inverse_gaussian(), tfd_johnson_s(), tfd_joint_distribution_named_auto_batched(),
tfd_joint_distribution_named(),
tfd_joint_distribution_sequential_auto_batched(),
tfd_joint_distribution_sequential(),
tfd_kumaraswamy(), tfd_laplace(), tfd_linear_gaussian_state_space_model(),
tfd_lkj(), tfd_logistic(), tfd_log_normal(), tfd_logistic(), tfd_mixture_same_family(),
tfd_mixture(), tfd_multinomial(), tfd_multivariate_normal_diag_plus_low_rank(),
tfd_multivariate_normal(),
tfd_multivariate_normal_full_covariance(), tfd_multivariate_normal_linear_operator(),
tfd_multivariate_normal_tril(),
tfd_multivariate_student_t_linear_operator(),
tfd_negative_binomial(),
tfd_normal(),
tfd_one_hot_categorical(),
tfd_pareto(),
tfd_pixel_cnn(),
tfd_poisson_log_normal_quadrature_compound(),
tfd_poisson(),
tfd_power_spherical(),
tfd_probit_bernoulli(),
tfd_quantized(),
tfd_relaxed_bernoulli(),
tfd_relaxed_one_hot_categorical(),
tfd_sample_distribution(),
tfd_sinh_arcsinh(),
tfd_skellam(),
tfd_spherical_uniform(),
tfd_student_t_process(),
tfd_student_t(),
tfd_transformed_distribution(),
tfd_triangular(),
tfd_truncated_cauchy(),
tfd_truncated_normal(),
tfd_uniform(),
tfd_variational_gaussian_process(),
tfd_vector_diffeomixture(),
tfd_vector_exponential_diag(),
tfd_vector_exponential_linear_operator(),
tfd_vector_laplace_diag(),
tfd_vector_laplace_linear_operator(),
tfd_vector_sinh_arcsinh_diag(),
tfd_von_mises_fisher(),
tfd_von_mises(),
tfd_weibull(),
tfd_wishart_linear_operator(),
tfd_wishart_tril(),
tfd_wishart()
Format

An object of class python.builtin.module (inherits from python.builtin.object) of length 6.

Value

Module(tensorflow_probability)

---

tfp_version   TensorFlow Probability Version

Description

TensorFlow Probability Version

Usage

tfp_version()

Value

the Python TFP version

---

vi_amari_alpha   The Amari-alpha Csiszar-function in log-space

Description

A Csiszar-function is a member of $F = \{ f: R_+ \to R : f \text{ convex } \}$.

Usage

vi_amari_alpha(logu, alpha = 1, self_normalized = FALSE, name = NULL)

Arguments

logu           float-like Tensor representing $\log(u)$ from above.
alpha          float-like scalar.
self_normalized logical indicating whether $f(u=1)=0$. When $f(u=1)=0$ the implied Csiszar f-Divergence remains non-negative even when $p, q$ are unnormalized measures.
name           name prefixed to Ops created by this function.
When `self_normalized = TRUE`, the Amari-alpha Csiszar-function is:

\[
    f(u) = \begin{cases} 
        -\log(u) + (u - 1), & \text{alpha} = 0 \\
        u \log(u) - (u - 1), & \text{alpha} = 1 \\
        \left( (u^{\alpha} - 1) - \alpha \frac{(u - 1)}{(\alpha (\alpha - 1))} \right), & \text{otherwise}
    \end{cases}
\]

When `self_normalized = FALSE` the `(u - 1)` terms are omitted.

Warning: when `alpha != 0` and/or `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for `|\log u| >> 0`.

**Value**

`amari_alpha_of_u` float-like Tensor of the Csiszar-function evaluated at \(u = \exp(\log u)\).

**References**


**See Also**

Other `vi`-functions: `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
**Details**

When `self_normalized = True` the Arithmetic-Geometric Csiszar-function is:

\[ f(u) = (1 + u) \log\left( \frac{1 + u}{\sqrt{u}} \right) - (1 + u) \log(2) \]

When `self_normalized = False` the \((1 + u) \log(2)\) term is omitted.

Observe that as an f-Divergence, this Csiszar-function implies:

\[
D_f[p, q] = KL[m, p] + KL[m, q]
\]

\[
m(x) = 0.5 \ p(x) + 0.5 \ q(x)
\]

In a sense, this divergence is the "reverse" of the Jensen-Shannon f-Divergence. This Csiszar-function induces a symmetric f-Divergence, i.e., \(D_f[p, q] = D_f[q, p]\).

Warning: when `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for \(\log u \gg 0\).

**Value**

`arithmetic_geometric_of_u`: float-like Tensor of the Csiszar-function evaluated at \(u = \exp(\log u)\).

**See Also**


---

**Description**

A Csiszar-function is a member of \(F = \{ f:R_+ \to R : f \text{ convex} \}\).

**Usage**

`vi_chi_square(logu, name = NULL)`

**Arguments**

- `logu`: float-like Tensor representing \(\log(u)\) from above.
- `name`: name prefixed to Ops created by this function.
Details

The Chi-square Csiszar-function is:

\[ f(u) = u^2 - 1 \]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| >> 0 \).

Value

\texttt{chi\_square\_of\_u}: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: \texttt{vi\_amari\_alpha()}, \texttt{vi\_arithmetic\_geometric()}, \texttt{vi\_csiszar\_vimco()}, \texttt{vi\_dual\_csiszar\_function()}, \texttt{vi\_fit\_surrogate\_posterior()}, \texttt{vi\_jeffreys()}, \texttt{vi\_jensen\_shannon()}, \texttt{vi\_kl\_forward()}, \texttt{vi\_kl\_reverse()}, \texttt{vi\_log1p\_abs()}, \texttt{vi\_modified\_gan()}, \texttt{vi\_monte\_carlo\_variational\_loss()}, \texttt{vi\_pearson()}, \texttt{vi\_squared\_hellinger()}, \texttt{vi\_symmetrized\_csiszar\_function()}

\begin{Verbatim}
vi\_csiszar\_vimco
\end{Verbatim}

Use VIMCO to lower the variance of the gradient of \( \text{csiszar\_function}(\text{Avg}(\log u)) \)

Description

This function generalizes VIMCO (Mnih and Rezende, 2016) to Csiszar f-Divergences.

Usage

\begin{verbatim}
vi_csiszar_vimco(f, p_log_prob, q, num_draws, num_batch_draws = 1, seed = NULL, name = NULL)
\end{verbatim}

Arguments

- \texttt{f}: function representing a Csiszar-function in log-space.
- \texttt{p\_log\_prob}: function representing the natural-log of the probability under distribution \( p \). (In variational inference \( p \) is the joint distribution.)
- \texttt{q}: \texttt{tfd\$Distribution-like} instance; must implement: \texttt{sample(n, seed)}, and \texttt{log\_prob(x)}. (In variational inference \( q \) is the approximate posterior distribution.)
**Details**

Note: if `q.reparameterization_type = tfd.FULLY_REPARAMETERIZED`, consider using `monte_carlo_csiszar_f_divergence`.

The VIMCO loss is:

\[
vimco = f(\text{Avg}\{\log u[i] : i=0,\ldots,m-1\})
\]

where,

\[
\log u[i] = \log( p(x, h[i]) / q(h[i] | x) )
\]

\[h[i] \text{iid} q(H | x)\]

Interestingly, the VIMCO gradient is not the naive gradient of `vimco`. Rather, it is characterized by:

\[
\text{grad}[vimco] - \text{variance\_reducing\_term}
\]

where,

\[
\text{variance\_reducing\_term} = \text{Sum}\{ \text{grad}[\log q(h[i] | x)] \ast (vimco - f(\log \text{Avg}\{h[j;i] : j=0,\ldots,m-1\})) \} : i=0 \text{ to } m
\]

\[
h[j;i] = u[j] \text{ for } j\neq i, \text{ GeometricAverage}\{ u[k] : k\neq i \} \text{ for } j=i
\]

(We omitted `stop_gradient` for brevity. See implementation for more details.) The `Avg\{h[j;i] : j\}` term is a kind of "swap-out average" where the `i`-th element has been replaced by the leave-`i`-out Geometric-average.

This implementation prefers numerical precision over efficiency, i.e., \(O(\text{num\_draws} \ast \text{num\_batch\_draws} \ast \text{prod}(\text{batch\_shape}) \ast \text{prod}(\text{event\_shape}))\). (The constant may be fairly large, perhaps around 12.)

**Value**

- `vimco` The Csiszar f-Divergence generalized VIMCO objective

**References**


**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_dual_csiszar_function

Calculates the dual Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex} \} \).

Usage

\[
\text{vi_dual_csiszar_function}(\text{logu}, \text{csiszar\_function}, \text{name = NULL})
\]

Arguments

- **logu** float-like Tensor representing \( \log(u) \) from above.
- **csiszar\_function** function representing a Csiszar-function over log-domain.
- **name** name prefixed to Ops created by this function.

Details

The Csiszar-dual is defined as:

\[
f^*(u) = u f(1 / u)
\]

where \( f \) is some other Csiszar-function. For example, the dual of \( \text{kl\_reverse} \) is \( \text{kl\_forward} \), i.e.,

\[
f(u) = -\log(u) \\
f^*(u) = u f(1 / u) = -u \log(1 / u) = u \log(u)
\]

The dual of the dual is the original function:

\[
f^{**}(u) = (u f(1/u))^*(u) = u (1/u) f(1/(1/u)) = f(u)
\]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \text{logu} \gg 0 \).

Value

\( \text{dual\_f\_of\_u} \) float-like Tensor of the result of calculating the dual of \( f \) at \( u = \exp(\text{logu}) \).

See Also

Other vi-functions: \( \text{vi\_amari\_alpha()}, \text{vi\_arithmetic\_geometric()}, \text{vi\_chi\_square()}, \text{vi\_csiszar\_vimco()}, \text{vi\_fit\_surrogate\_posterior()}, \text{vi\_jeffreys()}, \text{vi\_jensen\_shannon()}, \text{vi\_kl\_forward()}, \text{vi\_kl\_reverse()}, \text{vi\_log1p\_abs()}, \text{vi\_modified\_gan()}, \text{vi\_monte\_carlo\_variational\_loss()}, \text{vi\_pearson()}, \text{vi\_squared\_hellinger()}, \text{vi\_symmetrized\_csiszar\_function()} \)
vi_fit_surrogate_posterior

Fit a surrogate posterior to a target (unnormalized) log density

Description

The default behavior constructs and minimizes the negative variational evidence lower bound (ELBO), given by

```python
q_samples <- surrogate_posterior$sample(num_draws)
elbo_loss <- -tf$reduce_mean(target_log_prob_fn(q_samples) - surrogate_posterior$log_prob(q_samples))
```

Usage

```r
vi_fit_surrogate_posterior(
  target_log_prob_fn,
  surrogate_posterior,
  optimizer,
  num_steps,
  convergence_criterion = NULL,
  trace_fn = tfp$vi$optimization$trace_loss,
  variational_loss_fn = tfp$vi$optimization$reparameterized_elbo,
  sample_size = 1,
  trainable_variables = NULL,
  seed = NULL,
  name = "fit_surrogate_posterior"
)
```

Arguments

- **target_log_prob_fn**: function that takes a set of Tensor arguments and returns a Tensor log-density. Given `q_samples <- surrogate_posterior$sample(sample_size)`, this will be (in Python) called as `target_log_prob_fn(q_sample)` if `q_sample` is a list or a tuple, `target_log_prob_fn(**q_sample)` if `q_sample` is a dictionary, or `target_log_prob_fn(q_sample)` if `q_sample` is a Tensor. It should support batched evaluation, i.e., should return a result of shape `[sample_size]`.

- **surrogate_posterior**: A `tfp$distributions$Distribution` instance defining a variational posterior (could be a `tfp$distributions$JointDistribution`). Crucially, the distribution's `log_prob` and (if reparameterized) `sample` methods must directly invoke all ops that generate gradients to the underlying variables. One way to ensure this is to use `tfp$util$DeferredTensor` to represent any parameters defined as transformations of unconstrained variables, so that the transformations execute at runtime instead of at distribution creation.
optimizer
Optimizer instance to use. This may be a TF1-style `tf$train$Optimizer`, TF2-style `tf$optimizers$Optimizer`, or any Python-compatible object that implements `optimizer$apply_gradients(grads_and_vars)`.

num_steps
integer number of steps to run the optimizer.

corvergence_criterion
Optional instance of `tfp$optimizer$convergence_criteria$ConvergenceCriterion` representing a criterion for detecting convergence. If NULL, the optimization will run for `num_steps` steps, otherwise, it will run for at most `num_steps` steps, as determined by the provided criterion. Default value: NULL.

trace_fn
function with signature `state = trace_fn(loss,grads,variables)`, where `state` may be a Tensor or nested structure of Tensors. The state values are accumulated (by `tf$scan`) and returned. The default `trace_fn` simply returns the loss, but in general can depend on the gradients and variables (if `trainable_variables` is not NULL then `variables==trainable_variables`; otherwise it is the list of all variables accessed during execution of `loss_fn()`), as well as any other quantities captured in the closure of `trace_fn`, for example, statistics of a variational distribution. Default value: function(loss,grads,variables) loss.

variational_loss_fn
function with signature `loss <- variational_loss_fn(target_log_prob_fn,surrogate_posterior,sample_size,seed)` defining a variational loss function. The default is a Monte Carlo approximation to the standard evidence lower bound (ELBO), equivalent to minimizing the 'reverse' KL[q\|p] divergence between the surrogate q and true posterior p. Default value: `functools.partial(tfp.vi.monte_carlo_variational_loss,discrepancy_fn=tfp.vi.kl_reverse,use_reparameterization=True)`.

sample_size
integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

trainable_variables
Optional list of `tf$Variable` instances to optimize with respect to. If NULL, defaults to the set of all variables accessed during the computation of the variational bound, i.e., those defining `surrogate_posterior` and the model `target_log_prob_fn`. Default value: NULL.

seed
integer to seed the random number generator.

name
name prefixed to ops created by this function. Default value: 'fit_surrogate_posterior'.

Details
This corresponds to minimizing the 'reverse' Kullback-Liebler divergence (KL[q\|p]) between the variational distribution and the unnormalized `target_log_prob_fn`, and defines a lower bound on the marginal log likelihood, log p(x) >= -elbo_loss.

More generally, this function supports fitting variational distributions that minimize any Csiszar f-divergence.

Value
`results` Tensor or nested structure of Tensors, according to the return type of `result_fn`. Each Tensor has an added leading dimension of size `num_steps`, packing the trajectory of the result over the course of the optimization.
vi_jeffreys

The Jeffreys Csiszar-function in log-space

Description
A Csiszar-function is a member of $F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \}$.

Usage
vi_jeffreys(logu, name = NULL)

Arguments
logu float-like Tensor representing $\log(u)$ from above.
name name prefixed to Ops created by this function.

Details
The Jeffreys Csiszar-function is:

\[ f(u) = 0.5 \left( u \log(u) - \log(u) \right) \]
\[ = 0.5 \text{kl}_{\text{forward}} + 0.5 \text{kl}_{\text{reverse}} \]
\[ = \text{symmetrized_csiszar_function}(\text{kl}_{\text{reverse}}) \]
\[ = \text{symmetrized_csiszar_function}(\text{kl}_{\text{forward}}) \]

This Csiszar-function induces a symmetric $f$-Divergence, i.e., $D_f[p,q] = D_f[q,p]$.
Warning: this function makes non-log-space calculations and may therefore be numerically unstable for $|\log u| > 0$.

Value
jeffreys_of_u: float-like Tensor of the Csiszar-function evaluated at $u = \exp(\log u)$.

See Also
Other vi-functions: vi_amari_alpha(), vi_arithmetic_geometric(), vi_chi_square(), vi_csiszar_vimco(), vi_dual_csiszar_function(), vi_jeffreys(), vi_jensen_shannon(), vi_kl_forward(), vi_kl_reverse(), vi_log1p_abs(), vi_modified_gan(), vi_monte_carlo_variational_loss(), vi_pearson(), vi_squared_hellinger(), vi_symmetrized_csiszar_function()
vi_jensen_shannon

The Jensen-Shannon Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

Usage

\[
\text{vi}_\text{jensen}_\text{shannon}(\logu, \text{self\_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]

Arguments

- \( \logu \): float-like Tensor representing \( \log(u) \) from above.
- \( \text{self\_normalized} \): logical indicating whether \( f'(u=1)=0 \). When \( f'(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- \( \text{name} \): name prefixed to Ops created by this function.

Details

When \( \text{self\_normalized} = \text{True} \), the Jensen-Shannon Csiszar-function is:

\[
f(u) = u \log(u) - (1 + u) \log(1 + u) + (u + 1) \log(2)
\]

When \( \text{self\_normalized} = \text{False} \) the \( (u + 1) \log(2) \) term is omitted.

Observe that as an f-Divergence, this Csiszar-function implies:

\[
\text{D}_f[p, q] = \text{KL}[p, m] + \text{KL}[q, m]
\]

\[
m(x) = 0.5 \ p(x) + 0.5 \ q(x)
\]

In a sense, this divergence is the "reverse" of the Arithmetic-Geometric f-Divergence.

This Csiszar-function induces a symmetric f-Divergence, i.e., \( \text{D}_f[p, q] = \text{D}_f[q, p] \).

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \logu \gg 0 \).

Value

\( \text{jensen\_shannon\_of\_u} \), float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\logu) \).

References

See Also

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function`

---

**vi_kl_forward**

*The forward Kullback-Leibler Csiszar-function in log-space*

**Description**

A Csiszar-function is a member of $F = \{ f: R_+ \to R : f \text{ convex} \}$.

**Usage**

`vi_kl_forward(logu, self_normalized = FALSE, name = NULL)`

**Arguments**

- `logu` float-like Tensor representing $\log(u)$ from above.
- `self_normalized` logical indicating whether $f'(u=1)=0$. When $f'(u=1)=0$ the implied Csiszar f-Divergence remains non-negative even when $p$, $q$ are unnormalized measures.
- `name` name prefixed to Ops created by this function.

**Details**

When `self_normalized = TRUE`, the KL-reverse Csiszar-function is $f(u) = u \log(u) - (u - 1)$. When `self_normalized = FALSE` the $(u -1)$ term is omitted. Observe that as an f-Divergence, this Csiszar-function implies: ${\mathbb{D}}_f[p,q] = \text{KL}[q,p]$

The KL is "forward" because in maximum likelihood we think of minimizing $q$ as in $\text{KL}[p,q]$.

Warning: when `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for $\log(u) \gg 0$.

**Value**

`kl_forward_of_u` float-like Tensor of the Csiszar-function evaluated at $u = \exp(logu)$.

**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_kl_reverse
The reverse Kullback-Leibler Csiszar-function in log-space

Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \rightarrow \mathbb{R} : f \text{ convex} \} \).

Usage

\[
\text{vi_kl_reverse}(\text{logu}, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})
\]

Arguments

- \( \text{logu} \) float-like Tensor representing \( \log(u) \) from above.
- \( \text{self_normalized} \) logical indicating whether \( f(u=1)=0 \). When \( f(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- \( \text{name} \) name prefixed to Ops created by this function.

Details

When \( \text{self_normalized} = \text{TRUE} \), the KL-reverse Csiszar-function is \( f(u) = -\log(u) + (u - 1) \). When \( \text{self_normalized} = \text{FALSE} \) the \((u - 1)\) term is omitted. Observe that as an f-Divergence, this Csiszar-function implies: \( D_f[p,q] = \text{KL}[q,p] \)

The KL is “reverse” because in maximum likelihood we think of minimizing \( q \) as in \( \text{KL}[p,q] \).

Warning: when \( \text{self_normalized} = \text{TRUE} \) this function makes non-log-space calculations and may therefore be numerically unstable for \( |\logu| \gg 0 \).

Value

\( \text{kl_reverse_of_u} \) float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\text{logu}) \).

See Also

Other vi-functions: \text{vi_amari_alpha()}, \text{vi_arithmetic_geometric()}, \text{vi_chi_square()}, \text{vi_csiszar_vimco()}, \text{vi_dual_csiszar_function()}, \text{vi_fit_surrogate_posterior()}, \text{vi_jeffreys()}, \text{vi_jensen_shannon()}, \text{vi_kl_forward()}, \text{vi_log1p_abs()}, \text{vi_modified_gan()}, \text{vi_monte_carlo_variational_loss()}, \text{vi_pearson()}, \text{vi_squared_hellinger()}, \text{vi_symmetrized_csiszar_function}()
**vi_log1p_abs**

*The log1p-abs Csiszar-function in log-space*

### Description

A Csiszar-function is a member of \( F = \{ f: \mathbb{R}_+ \rightarrow \mathbb{R} : f \text{ convex} \} \).

### Usage

```r
vi_log1p_abs(logu, name = NULL)
```

### Arguments

- **logu**: float-like Tensor representing \( \log(u) \) from above.
- **name**: name prefixed to Ops created by this function.

### Details

The Log1p-Abs Csiszar-function is:

\[
 f(u) = u^{\text{sign}(u-1)} - 1
\]

This function is so-named because it was invented from the following recipe. Choose a convex function \( g \) such that \( g(0)=0 \) and solve for \( f \):

\[
 \log(1 + f(u)) = g(\log(u)). \\
\iff \\
 f(u) = \exp(g(\log(u))) - 1
\]

That is, the graph is identically \( g \) when y-axis is log1p-domain and x-axis is log-domain.

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| \gg 0 \).

### Value

\( \log1p\_abs\_of\_u \): float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

### See Also

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_modified_gan  

The Modified-GAN Csiszar-function in log-space

Description

A Csiszar-function is a member of $F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex } \}$.

Usage

$$\text{vi_modified_gan}(\log u, \text{self_normalized} = \text{FALSE}, \text{name} = \text{NULL})$$

Arguments

- $\log u$ float-like Tensor representing $\log(u)$ from above.
- $\text{self_normalized}$ logical indicating whether $f'(u=1)=0$. When $f'(u=1)=0$ the implied Csiszar $f$-
  Divergence remains non-negative even when $p, q$ are unnormalized measures.
- $\text{name}$ name prefixed to Ops created by this function.

Details

When $\text{self_normalized} = \text{True}$ the modified-GAN (Generative/Adversarial Network) Csiszar-function
is:

$$f(u) = \log(1 + u) - \log(u) + 0.5 \ (u - 1)$$

When $\text{self_normalized} = \text{False}$ the $0.5 \ (u - 1)$ is omitted.

The unmodified GAN Csiszar-function is identical to Jensen-Shannon (with $\text{self_normalized} = \text{False}$).

Warning: this function makes non-log-space calculations and may therefore be numerically unstable
for $|\log u| >> 0$.

Value

$$\text{jensen_shannon_of_u}, \text{float-like Tensor of the Csiszar-function evaluated at } u = \exp(\log u).$$

See Also

Other vi-functions: $\text{vi_amari_alpha()}, \text{vi_arithmetic_geometric()}, \text{vi_chi_square()}, \text{vi_csiszar_vimco()}, \text{vi_dual_csiszar_function()}, \text{vi_fit_surrogate_posterior()}, \text{vi_jeffreys()}, \text{vi_jensen_shannon()}, \text{vi_kl_forward()}, \text{vi_kl_reverse()}, \text{vi_log1p_abs()}, \text{vi_monte_carlo_variational_loss()}, \text{vi_pearson()}, \text{vi_squared_hellinger()}, \text{vi_symmetrized_csiszar_function()}$
Monte-Carlo approximation of an \( f \)-Divergence variational loss

Description

Variational losses measure the divergence between an unnormalized target distribution \( p \) (provided via `target_log_prob_fn`) and a surrogate distribution \( q \) (provided as `surrogate_posterior`). When the target distribution is an unnormalized posterior from conditioning a model on data, minimizing the loss with respect to the parameters of `surrogate_posterior` performs approximate posterior inference.

Usage

```r
vi_monte_carlo_variational_loss(
  target_log_prob_fn,
  surrogate_posterior,
  sample_size = 1,
  discrepancy_fn = vi_kl_reverse,
  use_reparametrization = NULL,
  seed = NULL,
  name = NULL
)
```

Arguments

- `target_log_prob_fn`: function that takes a set of Tensor arguments and returns a Tensor log-density. Given \( q\_sample \leftarrow \text{surrogate}\_posterior\$sample(sample\_size) \), this will be (in Python) called as `target_log_prob_fn(q\_sample)` if `q\_sample` is a list or a tuple, `target_log_prob_fn(**q\_sample)` if `q\_sample` is a dictionary, or `target_log_prob_fn(q\_sample)` if `q\_sample` is a Tensor. It should support batched evaluation, i.e., should return a result of shape `[sample\_size]`.

- `surrogate_posterior`: A `tfp\$distributions\$Distribution` instance defining a variational posterior (could be a `tfp\$distributions\$JointDistribution`). Crucially, the distribution’s `log\_prob` and (if reparameterized) `sample` methods must directly invoke all ops that generate gradients to the underlying variables. One way to ensure this is to use `tfp\$util\$DeferredTensor` to represent any parameters defined as transformations of unconstrained variables, so that the transformations execute at runtime instead of at distribution creation.

- `sample_size`: integer number of Monte Carlo samples to use in estimating the variational divergence. Larger values may stabilize the optimization, but at higher cost per step in time and memory. Default value: 1.

- `discrepancy_fn`: function representing a Csiszar \( f \) function in log-space. That is, `discrepancy\_fn(log(u)) = f(u)`, where \( f \) is convex in \( u \). Default value: `vi_kl_reverse`.
use_reparametrization

logical. When NULL (the default), automatically set to: surrogate_posterior.reparameterization_type == tfp$distributions$FULLY_REPARAMETERIZED. When TRUE uses the standard Monte-Carlo average. When FALSE uses the score-gradient trick. (See above for details.) When FALSE, consider using csiszar_vimco.

seed

integer seed for surrogate_posterior$sample.

name

name prefixed to Ops created by this function.

Details

This function defines divergences of the form $E_{q}[\text{discrepancy}_\text{fn}(\log p(z) - \log q(z))]$, sometimes known as f-divergences.

In the special case \(\text{discrepancy}_\text{fn}(\log u) = -\log u\) (the default \(\text{vi_kl_reverse}\)), this is the reverse Kullback-Liebler divergence \(KL[q||p]\), whose negation applied to an unnormalized \(p\) is the widely-used evidence lower bound (ELBO). Other cases of interest available under tfp$vi include the forward \(KL[p||q]\) (given by \(\text{vi_kl_forward}(\log u) = \exp(\log u) \times \log u\)), total variation distance, Amari alpha-divergences, and more.

Csiszar f-divergences

A Csiszar function \(f\) is a convex function from \(\mathbb{R}^+\) (the positive reals) to \(\mathbb{R}\). The Csiszar f-Divergence is given by:

\[
D_f[p(X), q(X)] := E_{q(X)}[ f( p(X) / q(X) ) ]
\]

\[
= m^{-1} \sum_{j=1}^{m} f( p(x_j) / q(x_j) )
\]

where \(x_j \sim \text{iid} q(X)\).

For example, \(f = \lambda u: -\log(u)\) recovers \(KL[q||p]\), while \(f = \lambda u: u \times \log(u)\) recovers the forward \(KL[p||q]\). These and other functions are available in tfp$vi.

Tricks: Reparameterization and Score-Gradient

When \(q\) is "reparameterized", i.e., a diffeomorphic transformation of a parameterless distribution (e.g., Normal(Y; m, s) \(\leftrightarrow\) Y = sX + m, X \sim Normal(0,1)), we can swap gradient and expectation, i.e., \(\text{grad}[\text{Avg}\{ s_i : i=1...n \}] = \text{Avg}\{ \text{grad}[s_i] : i=1...n \}\) where \(S_n=\text{Avg}\{s_i\}\) and \(s_i = f(x_i), x_i \sim \text{iid} q(X)\).

However, if \(q\) is not reparameterized, TensorFlow's gradient will be incorrect since the chain-rule stops at samples of unreparameterized distributions. In this circumstance using the Score-Gradient trick results in an unbiased gradient, i.e.,

\[
\text{grad}[ E_q[f(X)] ]
\]

\[
= \text{grad}[ \int dx \ q(x) \ f(x) ]
\]

\[
= \int dx \ \text{grad}[ \ q(x) \ f(x) ]
\]

\[
= \int dx \ [ \ q'(x) \ f(x) + q(x) \ f'(x) ]
\]

\[
= \text{int dx} \ q(x) \ [ q'(x) / q(x) \ f(x) + f'(x) ]
\]

\[
= \int dx \ q(x) \ \text{grad}[ f(x) \ q(x) / \text{stop_grad}[q(x)] ]
\]

\[
= E_q[ \ \text{grad}[ f(x) \ q(x) / \text{stop_grad}[q(x)] ] ]
\]

Unless \(q\.reparameterization_type != tfd.FULLY_REPARAMETERIZED\) it is usually preferable to set use_reparametrization = True.

Example Application: The Csiszar f-Divergence is a useful framework for variational inference. I.e., observe that,
\[ f(p(x)) = f\left( E_{q(Z \mid x)}\left[ \frac{p(x, Z)}{q(Z \mid x)} \right] \right) \leq E_{q(Z \mid x)}\left[ f\left( \frac{p(x, Z)}{q(Z \mid x)} \right) \right] := D_f[p(x, Z), q(Z \mid x)] \]

The inequality follows from the fact that the "perspective" of \( f \), i.e., \( (s, t) \mapsto t f(s / t) \), is convex in \( (s, t) \) when \( s/t \) in domain(\( f \)) and \( t \) is a real. Since the above framework includes the popular Evidence Lower BOund (ELBO) as a special case, i.e., \( f(u) = -\log(u) \), we call this framework "Evidence Divergence Bound Optimization" (EDBO).

**Value**

```
monte_carlo_variational_loss float-like Tensor
  Monte Carlo approximation of the Csiszar f-Divergence.
```

**References**


**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_pearson()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`

---

**vi_pearson**

*The Pearson Csiszar-function in log-space*

**Description**

A Csiszar-function is a member of \( F = \{ f:R_+ \rightarrow R : f \text{ convex } \} \).

**Usage**

```
vi_pearson(logu, name = NULL)
```

**Arguments**

- `logu` float-like Tensor representing \( \log(u) \) from above.
- `name` name prefixed to Ops created by this function.

**Details**

The Pearson Csiszar-function is:

\[ f(u) = (u - 1)^2 \]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \log(u) \gg 0 \).
The Squared-Hellinger Csiszar-function in log-space

**Description**

A Csiszar-function is a member of \( F = \{ f : \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \} \).

**Usage**

```r
v_squared_hellinger(logu, name = NULL)
```

**Arguments**

- `logu`: float-like Tensor representing \( \log(u) \) from above.
- `name`: name prefixed to Ops created by this function.

**Details**

The Squared-Hellinger Csiszar-function is:

\[
f(u) = (\sqrt{u} - 1)^2
\]

This Csiszar-function induces a symmetric f-Divergence, i.e., \( D_f[p,q] = D_f[q,p] \).

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( |\log u| >> 0 \).

**Value**

Squared-Hellinger_of_u: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

**See Also**

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chisquare()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_loglp_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_squared_hellinger()`, `vi_symmetrized_csiszar_function()`
vi_symmetrized_csiszar_function

Symmetrizes a Csiszar-function in log-space

Description

A Csiszar-function is a member of $F = \{ f: \mathbb{R}_+ \to \mathbb{R} : f \text{ convex} \}$. 

Usage

```r
vi_symmetrized_csiszar_function(logu, csiszar_function, name = NULL)
```

Arguments

- `logu` float-like Tensor representing log($u$) from above.
- `csiszar_function` function representing a Csiszar-function over log-domain.
- `name` name prefixed to Ops created by this function.

Details

The symmetrized Csiszar-function is defined as:

$$f_g(u) = 0.5 \ g(u) + 0.5 \ u \ g \left( \frac{1}{u} \right)$$

where $g$ is some other Csiszar-function. We say the function is "symmetrized" because:

$$D_{f_g}[p, q] = D_{f_g}[q, p]$$

for all $p \ll q$ (i.e., support($p$) = support($q$)).

There exists alternatives for symmetrizing a Csiszar-function. For example,

$$f_g(u) = \max(f(u), f^*(u)),$$

where $f^*$ is the dual Csiszar-function, also implies a symmetric f-Divergence.

Example: When either of the following functions are symmetrized, we obtain the Jensen-Shannon Csiszar-function, i.e.,

$$g(u) = -\log(u) - (1 + u) \ \log((1 + u) / 2) + u - 1$$
$$h(u) = \log(4) + 2 \ u \ \log(u / (1 + u))$$

implies,

$$f_g(u) = f_h(u) = u \ \log(u) - (1 + u) \ \log((1 + u) / 2)$$
$$= jensen_shannon(log(u)).$$

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for $|\log u| \gg 0$. 
Value

`symmetrized_g_of_u`: float-like Tensor of the result of applying the symmetrization of `g` evaluated at `u = exp(logu)`.

See Also

Other vi-functions: `vi_amari_alpha()`, `vi_arithmetic_geometric()`, `vi_chi_square()`, `vi_csiszar_vimco()`, `vi_dual_csiszar_function()`, `vi_fit_surrogate_posterior()`, `vi_jeffreys()`, `vi_jensen_shannon()`, `vi_kl_forward()`, `vi_kl_reverse()`, `vi_log1p_abs()`, `vi_modified_gan()`, `vi_monte_carlo_variational_loss()`, `vi_pearson()`, `vi_squared_hellinger()`

---

**vi_total_variation**

**The Total Variation Csiszar-function in log-space**

**Description**

A Csiszar-function is a member of \( F = \{ f:R_+ \to R : f \text{ convex} \} \).

**Usage**

`vi_total_variation(logu, name = NULL)`

**Arguments**

- `logu` float-like Tensor representing \( \log(u) \) from above.
- `name` name prefixed to Ops created by this function.

**Details**

The Total-Variation Csiszar-function is:

\[
f(u) = 0.5 \ |u - 1|
\]

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for \( \logu \gg 0 \).

**Value**

`total_variation_of_u`: float-like Tensor of the Csiszar-function evaluated at `u = exp(logu)`.

**See Also**

Other vi-functions: `vi_t_power()`, `vi_triangular()`
vi_triangular

vi_triangular
The Triangular Csiszar-function in log-space

Description

The Triangular Csiszar-function is:

Usage

vi_triangular(logu, name = NULL)

Arguments

logu float-like Tensor representing log(u) from above.
name name prefixed to Ops created by this function.

Details

f(u) = (u - 1)**2 / (1 + u)

Warning: this function makes non-log-space calculations and may therefore be numerically unstable for |logu| >> 0.

Value

triangular_of_u: float-like Tensor of the Csiszar-function evaluated at u = exp(logu).

See Also

Other vi-functions#': vi_t_power(), vi_total_variation()
Arguments

- **logu**: float-like Tensor representing log(u) from above.
- **t**: Tensor of same dtype as logu and broadcastable shape.
- **self_normalized**: logical indicating whether \( f(u=1)=0 \). When \( f(u=1)=0 \) the implied Csiszar f-Divergence remains non-negative even when \( p, q \) are unnormalized measures.
- **name**: name prefixed to Ops created by this function.

Details

When `self_normalized = True` the T-Power Csiszar-function is:

\[
  f(u) = s \left[ u^t - 1 - t(u - 1) \right]
\]

\[
  s = \begin{cases} 
  -1 & 0 < t < 1 \\
  +1 & \text{otherwise}
\end{cases}
\]

When `self_normalized = False` the \(-t(u -1)\) term is omitted.

This is similar to the amari_alpha Csiszar-function, with the associated divergence being the same up to factors depending only on \( t \).

Warning: when `self_normalized = True` this function makes non-log-space calculations and may therefore be numerically unstable for \(|\log u| \gg 0\).

Value

- **t_power_of_u**: float-like Tensor of the Csiszar-function evaluated at \( u = \exp(\log u) \).

See Also

Other vi-functions: `vi_total_variation()`, `vi_triangular()`
Index

* bijector_methods
  tfb_forward, 164
  tfb_forward_log_det_jacobian, 165
  tfb_inverse, 175
  tfb_inverse_log_det_jacobian, 176

* bijectors
  tfb_absolute_value, 141
  tfb_affine, 142
  tfb_affine_linear_operator, 144
  tfb_affine_scalar, 145
  tfb_ascending, 146
  tfb_batch_normalization, 147
  tfb_blockwise, 149
  tfb_chain, 150
  tfb_cholesky_outer_product, 151
  tfb_cholesky_to_inv_cholesky, 152
  tfb_correlation_cholesky, 153
  tfb_cumsum, 155
  tfb_discrete_cosine_transform, 156
  tfb_exp, 157
  tfb_expm1, 158
  tfb_ffjord, 159
  tfb_fill_scale_tri_l, 162
  tfb_fill_triangular, 163
  tfb_glow, 166
  tfb_gompertz_cdf, 169
  tfb_gumbel, 171
  tfb_gumbel_cdf, 172
  tfb_identity, 173
  tfb_inhale, 174
  tfb_invert, 177
  tfb_iterated_sigmoid_centered, 178
  tfb_kumaraswamy, 179
  tfb_kumaraswamy_cdf, 180
  tfb_lambert_w_tail, 181
  tfb_masked_autoregressive_default_template, 182
  tfb_masked_autoregressive_flow, 184
  tfb_masked_dense, 187
  tfb_matrix_inverse_tri_l, 189
  tfb_matvec_lu, 190
  tfb_normal_cdf, 191
  tfb_ordered, 192
  tfb_pad, 193
  tfb_permute, 194
  tfb_power_transform, 195
  tfb_rational_quadratic_spline, 196
  tfb_rayleigh_cdf, 198
  tfb_real_nvp, 199
  tfb_real_nvp_default_template, 201
  tfb_reciprocal, 203
  tfb_reshape, 204
  tfb_scale, 205
  tfb_scale_matvec_diag, 206
  tfb_scale_matvec_linear_operator, 207
  tfb_scale_matvec_lu, 208
  tfb_scale_matvec_tri_l, 209
  tfb_scale_tri_l, 211
  tfb_shift, 212
  tfb_shifted_gompertz_cdf, 213
  tfb_sigmoid, 214
  tfb_sinh, 215
  tfb_sinh_arcsinh, 216
  tfb_softmax_centered, 217
  tfb_softplus, 218
  tfb_softsign, 220
  tfb_split, 221
  tfb_square, 222
  tfb_tanh, 223
  tfb_transform_diagonal, 224
  tfb_transpose, 225
  tfb_weibull, 226
  tfb_weibull_cdf, 227

* datasets
  tfp, 434

* distribution_layers
layer_categorical_mixture_of_one_hot_categorical, 17
layer_distribution_lambda, 43
layer_independent_bernoulli, 44
layer_independent_logistic, 45
layer_independent_normal, 46
layer_independent_poisson, 47
layer_kl_divergence_add_loss, 48
layer_kl_divergence_regularizer, 49
layer_multivariate_normal_tri_l, 54
layer_one_hot_categorical, 55

* distributions
  tfd_autoregressive, 229
tfd_batch_reshape, 231
tfd_bates, 232
tfd_bernoulli, 234
tfd_beta, 236
tfd_beta_binomial, 238
tfd_binomial, 240
tfd_categorical, 243
tfd_cauchy, 244
tfd_chi, 247
tfd_chi2, 248
tfd_cholesky_lkj, 250
tfd_continuous_beroulli, 251
tfd_deterministic, 255
tfd_dirichlet, 257
tfd_dirichlet_multinomial, 259
tfd_dirichlet_multinomial, 259
tfd_empirical, 262
  tfd_exp_gamma, 266
tfd_exp_inverse_gamma, 268
  tfd_exponential, 265
tfd_gamma, 273
tfd_gamma, 273
tfd_gamma_gamme, 275
tfd_gaussian_process, 277
tfd_gaussian_process_regression_model, 280
tfd_beta, 236
tfd_beta_binomial, 238
tfd_binomial, 240
tfd_continuous_bernoulli, 251
tfd_deterministic, 255
tfd_multivariate_student_t_linear_operator, 347
  * layers
  layer_autoregressive, 14
  layer_conv_1d_flipout, 18
  layer_conv_1d_reparameterization, 21
  layer_conv_2d_flipout, 23
  layer_conv_2d_reparameterization, 26
  layer_conv_3d_flipout, 29
  layer_conv_3d_reparameterization, 32
  layer_dense_flipout, 34
  layer_dense_local_reparameterization, 37
  layer_dense_reparameterization, 39
  layer_dense_variational, 41
  layer_variable, 56
  * mcmc_kernels
  mcmc_dual_averaging_step_size_adaptation, 58
  mcmc_hamiltonian_monte_carlo, 62
  mcmc_metropolis_adjusted_langevin_algorithm, 64
  mcmc_metropolis_hastings, 65
  mcmc_no_u_turn_sampler, 67
  mcmc_random_walk_metropolis, 71
  mcmc_replica_exchange_mc, 73
  mcmc_simple_step_size_adaptation, 80
  mcmc_slice_sampler, 83
  mcmc_transformed_transition_kernel, 84
  mcmc_uncalibrated_hamiltonian_monte_carlo, 86
  mcmc_uncalibrated_langevin, 87
  mcmc_uncalibrated_random_walk, 88

  * glm_fit
  glm_families, 8
  glm_fit.tensorflow.tensor, 9
  glm_fit_one_step.tensorflow.tensor,
INDEX

layer_conv_3d_reparameterization, 15, 20, 23, 26, 29, 31, 32, 37, 39, 41, 42, 57
layer_dense_flipout, 15, 21, 23, 26, 29, 32, 34, 39, 41, 42, 57
layer_dense_local_reparameterization, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 57
layer_dense_reparameterization, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 57
layer_distribution_lambda, 18, 43, 44–46, 48–54, 56
layer_independent_beroulli, 18, 43, 44, 45, 46, 48–54, 56
layer_independent_logistic, 18, 43, 44, 45, 46, 48–54, 56
layer_independent_normal, 18, 43–45, 46–48, 48–54, 56
layer_independent_normal(), 18, 43–45, 48–54, 56
layer_independent_poisson, 18, 43–46, 47, 49–54, 56
layer_kl_divergence_add_loss, 18, 43–46, 48, 49, 50–54, 56
layer_kl_divergence_regularizer, 18, 43–46, 48, 49, 49, 51–54, 56
layer_mixture_logistic, 18, 43–46, 48–50, 50, 52–54, 56
layer_mixture_normal, 18, 43–46, 48–51, 51, 53, 54, 56
layer_mixture_same_family, 18, 43–46, 48–52, 52, 54, 56
layer_multivariate_normal_tri_l, 18, 43–46, 48–53, 54, 56
layer_one_hot_categorical, 18, 43–46, 48–54, 55
layer_variable, 15, 21, 23, 26, 29, 32, 34, 37, 39, 41, 42, 56
layer_variational_gaussian_process, 57
mcmc_dual_averaging_step_size_adaptation, 58, 64–66, 68, 72, 74, 82, 84, 85, 87–89
mcmc_effective_sample_size, 61, 71, 75, 77, 80
mcmc.hamiltonian.monte.carlo, 61, 62, 65, 66, 68, 72, 74, 82, 84, 85, 87–89
mcmc_metropolis_adjusted_langevin_algorithm, 61, 64, 66, 68, 72, 74, 82, 84, 85, 87–89
mcmc_metropolis_hastings, 61, 64, 65, 65, 68, 72, 74, 82, 84, 85, 87–89
mcmc_no_u_turn_sampler, 61, 64–66, 67, 72, 74, 82, 84, 85, 87–89
mcmc_no_u_turn_sampler(), 61
mcmc_potential_scale_reduction, 62, 70, 75, 77, 80
mcmc_random_walk_metropolis, 61, 64–66, 68, 71, 74, 82, 84, 85, 87–89
mcmc_replica_exchange_mc, 61, 64–66, 68, 72, 73, 82, 84, 85, 87–89
mcmc_sample_annealed_importance_chain, 62, 71, 74, 77, 80
mcmc_sample_chain, 62, 71, 75, 76, 80
mcmc_sample_chain(), 75, 80
mcmc_sample_halton_sequence, 62, 71, 75, 77, 78
mcmc_simple_step_size_adaptation, 61, 64–66, 68, 72, 74, 80, 84, 85, 87–89
mcmc_slice_sampler, 61, 64–66, 68, 72, 74, 82, 83, 85, 87–89
mcmc_transfomred_transition_kernel, 61, 64–66, 68, 72, 84, 84, 87–89
mcmc_uncalibrated_hamiltonian.monte.carlo, 61, 64–66, 68, 72, 74, 82, 84, 85, 86, 88, 89
mcmc_uncalibrated_langevin, 61, 64–66, 68, 72, 74, 82, 84, 85, 87, 87, 89
mcmc_uncalibrated_random_walk, 61, 64–66, 68, 72, 74, 82, 84, 85, 87, 88, 88
params_size_categorical_mixture_of_one_hot_categorical, 89
params_size_independent_beroulli, 90
params_size_independent_logistic, 90
params_size_independent_normal, 91
params_size_independent_poisson, 91
params_size_mixture_logistic, 92
params_size_mixture_normal, 92
params_size_mixture_same_family, 93
params_size_multivariate_normal_tri_l, 93
params_size_one_hot_categorical, 94
sts.additive.state.space.model, 94, 98
sts_autoregressive, 97, 97, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_constrained_seasonal_state_space_model, 97, 98, 101, 103, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_fit_with_hmc, 102, 103, 107, 108, 113, 115, 124, 125
sts_fit_with_hmc(), 98, 109, 117, 118, 121, 127, 131, 135, 140, 141
sts_forecast, 102, 103, 107, 108, 113, 115, 124, 125
sts_forecast(), 98, 109, 117, 118, 121, 127, 131, 135, 140, 141
sts_local_level, 97, 98, 101, 106, 109, 111, 117, 119, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_local_linear_trend, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_local_linear_trend_state_space_model, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_one_step_predictive, 102, 103, 107, 108, 113, 115, 124, 125
sts_sample_uniform_initial_state, 102, 103, 107, 108, 113, 115, 124, 124
sts_seasonal, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 125, 129, 131, 133, 135, 137, 140, 141
sts_seasonal_state_space_model, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_semi_local_linear_trend, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_semi_local_linear_trend_state_space_model, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 140, 141
sts_smooth_linear_regression, 97, 98, 101, 106, 109, 111, 117, 118, 120, 121, 123, 127, 129, 131, 133, 135, 137, 138, 141
INDEX

135, 137, 140, 140
tensorflow::install_tensorflow(), 14
tfb_absolute_value, 141, 144–149,
151–158, 161–163, 169–173, 175,
178, 180–182, 184, 187–192,
194–196, 198, 199, 201–204,
206–212, 214, 215, 217, 218,
220–224, 226–228
tfb_affine, 142, 142, 145–149, 151–158,
161–163, 169–173, 175, 178,
180–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_affine_linear_operator, 142, 144,
144, 146–149, 151–158, 161–163,
169–173, 175, 178, 180–182, 184,
187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_affine_scalar, 142, 144, 145, 145,
147–149, 151–158, 161–163,
169–173, 175, 180–182, 184,
187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_ascending, 142, 144–146, 146, 148, 149,
151–158, 161–163, 169–173, 175,
178, 180–182, 184, 187–192,
194–196, 198, 199, 201–204,
206–212, 214, 215, 217, 218,
220–224, 226–228
tfb_batch_normalization, 142, 144–147,
147, 149, 151–158, 161–163,
169–173, 175, 180–182, 184,
187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_blockwise, 142, 144–148, 149, 151–158,
161–163, 169–173, 175, 178,
180–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_chain, 142, 144–149, 150, 152–158,
161–163, 169–173, 175, 178,
180–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_cholesky_outer_product, 142,
144–149, 151, 151, 153–158,
161–163, 169–173, 175, 178,
180–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_cholesky_to_inv_cholesky, 142,
144–149, 151, 152, 152–154–158,
161–163, 169–173, 175, 178,
180–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_correlation_cholesky, 142, 144–149,
151–153, 153, 155–158, 161–163,
169–173, 175, 178, 180–182, 184,
187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_cumsum, 142, 144–149, 151–154, 155,
156–158, 161–163, 169–173, 175,
178–182, 184, 187–192, 194–196,
198, 199, 201–204, 206–212, 214,
215, 217, 218, 220–224, 226–228
tfb_discrete_cosine_transform, 142,
144–149, 151–155, 156, 157, 158,
161–163, 169–173, 175, 178–182,
184, 187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_exp, 142, 144–149, 151–156, 157, 158,
161–163, 169–173, 175, 178–182,
184, 187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_expml, 142, 144–149, 151–157, 158,
161–163, 169–173, 175, 178–182,
184, 187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_ffjord, 142, 144–149, 151–158, 159,
162, 163, 169–173, 175, 178–182,
184, 187–192, 194–196, 198, 199,
201–204, 206–212, 214, 215, 217,
218, 220–224, 226–228
tfb_fill_scale_tril, 142, 144–149,
151–158, 161, 162, 163, 169–173,
175, 178–182, 184, 187–192,
194–196, 198, 199, 201–203,
205–212, 214–218, 220–224,
| tfb_inverse_log_det_jacobian | 164, 165, 176, 177 |
| tfb_log_det_jacobian | 164, 165, 176, 177 |
| tfb_log_gyrate_inverse_log_det_jacobian | 164, 165, 176, 177 |
INDEX

tfb_masked_autoregressive_flow(), 16

tfb_scale_matvec_linear_operator, 142, 144–148, 150–153, 155, 157, 158,
<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
</table>
INDEX

468

424, 434

tfd_cdf, 246, 254, 255, 264, 313, 324,
328–330, 334, 368, 370, 377, 385, 391, 401

tfd_chisq, 230, 232, 234, 235, 237, 239, 241,
244, 245, 247, 249, 251, 253, 256,
258, 260, 263, 266, 268, 270, 275,
277, 280, 282, 284, 288, 289, 291,
292, 294, 296, 298, 300, 302, 304,
305, 308, 309, 312, 315, 316, 319,
321, 322, 325, 327, 331, 333, 336,
338, 341, 343, 345, 347, 348, 350,
352, 354, 355, 360, 363, 365, 367,
369, 372, 374, 376, 378, 381, 383,
384, 387, 390, 393, 394, 396,
398, 400, 405, 409, 411, 414, 416,
418, 421, 423, 425, 426, 428, 430,
432, 434

tfd_blockwise, 242

tfd_categorical, 230, 232, 234, 235, 237,
239, 241, 243, 245, 248, 249, 251,
253, 256, 258, 260, 263, 266, 268,
270, 275, 277, 280, 282, 284, 288,
289, 291, 292, 294, 296, 298, 300,
302, 304, 305, 308, 309, 312, 315,
316, 319, 321, 322, 325, 327, 331,
333, 336, 338, 341, 343, 345, 347,
348, 350, 352, 354, 355, 360, 363,
365, 367, 369, 372, 374, 376, 378,
381, 383, 384, 387, 390, 393, 394,
396, 398, 400, 405, 409, 411, 414,
416, 418, 421, 423, 425, 426, 428,
430, 432, 434

tfd_cholesky_lkj, 230, 232, 234, 235, 237,
239, 241, 244, 245, 248, 249, 250,
253, 256, 258, 260, 263, 266, 268,
270, 275, 277, 280, 282, 284, 288,
289, 291, 292, 294, 296, 298, 300,
302, 304, 305, 308, 309, 312, 315,
316, 319, 321, 322, 325, 327, 331,
333, 336, 338, 341, 343, 345, 347,
348, 350, 352, 354, 355, 360, 363,
365, 367, 369, 372, 374, 376, 378,
381, 383, 384, 387, 390, 393, 394,
396, 398, 400, 405, 409, 411, 414,
416, 418, 421, 423, 425, 426, 428,
430, 432, 434

tfd_continuous_bernoulli, 230, 232, 234,
235, 237, 239, 241, 244, 245, 248,
tfd_exp_relaxed_one_hot_categorical


tfd_finite_discrete, 272


INDEX


tfd_kl_divergence, 246, 254, 255, 264, 313, 324, 328–330, 334, 368, 370, 377, 385, 391, 401


tfd_log_cdf, 246, 254, 255, 256, 313, 324, 328–330, 334, 368, 370, 377, 385, 391, 401


tfd_logit_normal, 308, 309
280, 266
239
282
329
330, 334, 368, 370, 377, 385, 391, 401
tfd_log_prob, 246, 254, 255, 264, 313, 324,
328, 329, 330, 334, 368, 370, 377, 385, 391, 401
tfd_log_prob(), 230, 232, 234, 235, 237,
239, 241, 242, 244, 245, 248, 249,
251, 253, 256, 258, 260, 262, 263, 266, 268, 270, 271, 273, 275, 277,
280, 282, 284, 286, 288, 291, 292, 294, 296, 297, 300, 302, 304, 305,
308, 309, 312, 315, 316, 319, 321,
322, 324, 325, 327, 331, 333, 336,
338, 341, 343, 345, 347, 348, 350,
352, 354, 355, 357, 360, 362, 363,
365, 367, 369, 372, 374, 376, 378,
381, 383, 384, 387, 390, 393, 394,
396, 398, 400, 405, 407, 409, 411,
414, 416, 418, 421, 423, 425, 426,
428, 430, 432, 434
tfd_logit_survival_function, 246, 254, 255,
264, 313, 324, 328, 328, 330, 334,
368, 370, 377, 385, 391, 401
tfd_logistic, 231, 232, 234, 235, 237, 239,
241, 244, 246, 248, 249, 251, 253,
256, 258, 260, 264, 266, 268, 270,
275, 277, 280, 282, 284, 288, 289,
291, 292, 294, 296, 298, 300, 302,
304, 305, 308, 310, 312, 315, 316,
319, 321, 323, 326, 327, 330, 333,
336, 338, 341, 343, 345, 347, 349,
351, 352, 354, 356, 360, 363, 365,
367, 369, 372, 374, 376, 378, 381,
383, 384, 387, 390, 393, 394, 396,
398, 400, 405, 409, 412, 414, 416,
418, 421, 423, 425, 426, 428, 430,
432, 434
tfd_logit_normal, 323
tfd_mean, 246, 254, 255, 264, 313, 324, 328,
329, 329, 334, 368, 370, 377, 385,
391, 401
tfd_mean(), 230, 232, 234, 235, 237, 239,
241, 242, 244, 245, 248, 249, 251,
253, 256, 258, 260, 262, 263, 266,
268, 270, 271, 273, 275, 277, 280,
282, 284, 286, 288, 291, 292, 294,

tfd_stddev, 246, 254, 255, 264, 313, 324, 328–330, 334, 368, 370, 377, 385, 391, 401


tfd_survival_function, 246, 254, 255, 264, 313, 324, 328–330, 334, 368, 370, 377, 385, 391, 401


<table>
<thead>
<tr>
<th>line</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>tfp, 434</td>
<td></td>
</tr>
<tr>
<td>tfp_version, 435</td>
<td></td>
</tr>
<tr>
<td>vi_arithmetic_geometric, 436, 436, 438–440, 443, 445–448, 451, 452, 454</td>
<td></td>
</tr>
<tr>
<td>vi_chi_square, 436, 437, 437, 439, 440, 443, 445–448, 451, 452, 454</td>
<td></td>
</tr>
<tr>
<td>vi_dual_csisz_var_function, 436–439, 440, 443, 445–448, 451, 452, 454</td>
<td></td>
</tr>
<tr>
<td>vi_fit_surrogate_posterior, 436–440, 441, 443, 445–448, 451, 452, 454</td>
<td></td>
</tr>
<tr>
<td>vi_jeffreys, 436–440, 443, 445–448, 451, 452, 454</td>
<td></td>
</tr>
<tr>
<td>vi_kl_forward, 436–440, 443, 445, 445</td>
<td></td>
</tr>
</tbody>
</table>
vi_kl_reverse, 436–440, 443, 445, 446, 447, 448, 451, 452, 454
vi_log1p_abs, 436–440, 443, 445, 446, 447, 448, 451, 452, 454
vi_monte_carlo_variational_loss, 436–440, 443, 445–448, 449, 452, 454
vi_squared_hellinger, 436–440, 443, 445–448, 451, 452, 454
vi_symmetrized_csiszar_function, 436–440, 443, 445–448, 451, 452, 453
vi_t_power, 454, 455, 455
vi_total_variation, 454, 455, 456
vi_triangular, 454, 455, 456