Package ‘torch’

December 15, 2023

Type Package
Title Tensors and Neural Networks with 'GPU' Acceleration
Version 0.12.0
Description Provides functionality to define and train neural networks similar to 'PyTorch' by Paszke et al (2019) <arXiv:1912.01703> but written entirely in R using the 'libtorch' library. Also supports low-level tensor operations and 'GPU' acceleration.
License MIT + file LICENSE
BugReports https://github.com/mlverse/torch/issues
Encoding UTF-8
SystemRequirements LibTorch (https://pytorch.org/); Only x86_64 platforms are currently supported except for ARM system running macOS.
Config/build/copy-method copy
LinkingTo Rcpp
Imports Rcpp, R6, withr, rlang, methods, utils, stats, bit64, magrittr, tools, coro (>= 1.0.2), callr, cli (>= 3.0.0), glue, ellipsis, desc, safetensors (>= 0.1.1), jsonlite
RoxygenNote 7.2.3
Suggests testthat (>= 3.0.0), covr, knitr (>= 1.36), rmarkdown, palmerpenguins, mvtnorm, numDeriv, katex
VignetteBuilder knitr
'distributions-chi2.R' 'distributions-mixture_same_family.R'
'distributions-multivariate_normal.R' 'distributions-normal.R'
'distributions-poisson.R' 'dtype.R' 'gen-method.R'
'gen-namespace-docs.R' 'gen-namespace-examples.R'
'gen-namespace.R' 'generator.R' 'help.R' 'indexing.R'
'install.R' 'ivalue.R' 'jit-compile.R' 'jit-execute.R'
'jit-ops.R' 'lantern_load.R' 'lantern_sync.R' 'layout.R'
'linear.R' 'memory_format.R' 'utils-data.R' 'nn.R'
'nn-activation.R' 'nn-batchnorm.R' 'nn-conv.R' 'nn-distance.R'
'nn-dropout.R' 'nn-flatten.R' 'nn-init.R' 'nn-linear.R'
'nn-loss.R' 'nn-normalization.R' 'nn-pooling.R' 'nn-rnn.R'
'nn-sparse.R' 'nn-upsampling.R' 'nn-utils-clip-grad.R'
'nn-utils-rnn.R' 'nn-utils-weight-norm.R' 'nn-utils.R'
'nn_adaptive.R' 'nnf-activation.R' 'nnf-batchnorm.R'
'nnf-conv.R' 'nnf-distance.R' 'nnf-dropout.R' 'nnf-embedding.R'
'nnf-fold.R' 'nnf-in斯坦norm.R' 'nnf-linear.R' 'nnf-loss.R'
'nnf-normalization.R' 'nnf-padding.R' 'nnf-pixelshuffle.R'
'nnf-pooling.R' 'nnf-upsampling.R' 'nnf-vision.R' 'operators.R'
'optim.R' 'optim-adadelta.R' 'optim-adagrad.R' 'optim-adam.R'
'optim-adamw.R' 'optim-asgd.R' 'optim-lbfgs.R'
'optim_lr_scheduler.R' 'optim-rmsprop.R' 'optim-rprop.R'
'optim-sgd.R' 'package.R' 'qscheme.R' 'quantization.R'
'reduction.R' 'save.R' 'scalar.R' 'script_module.R' 'stack.R'
'storage.R' 'threads.R' 'trace.R' 'translate.R' 'type-info.R'
'utils-data-collate.R' 'utils-data-dataloader.R'
'utils-data-enum.R' 'utils-data-fetcher.R'
'utils-data-sampler.R' 'utils-pipe.R' 'variable_list.R'
'with-indices.R' 'wrappers.R'

NeedsCompilation yes

Author  Daniel Falbel [aut, cre, cph],
         Javier Luraschi [aut],
         Dmitriy Selivanov [ctb],
         Athos Damiani [ctb],
         Christophe Regouby [ctb],
         Krzysztof Joachimiak [ctb],
         Hamada S. Badr [ctb],
         RStudio [cph]

Maintainer Daniel Falbel <daniel@rstudio.com>

Repository CRAN

Date/Publication 2023-12-15 16:00:02 UTC

R topics documented:

  as_array ................................................................. 17
  AutogradContext ........................................................ 17
  autograd_backward ...................................................... 19
is_torch_qscheme ........................................ 52
is_undefined_tensor ..................................... 52
iterable_dataset .......................................... 53
jit_compile .................................................. 54
jit_load ....................................................... 54
jit_ops ......................................................... 55
jit_save ......................................................... 56
jit_save_for_mobile ....................................... 56
jit_scalar ..................................................... 57
jit_trace ....................................................... 57
jit_trace_module .......................................... 59
jit_tuple ...................................................... 60
linalg_cholesky ............................................. 60
linalg_cholesky_ex ......................................... 61
linalg_cond ..................................................... 62
linalg_det ....................................................... 64
linalg_eig ....................................................... 64
linalg_eigh ..................................................... 66
linalg_eigvals ................................................ 68
linalg_eigvalsh ............................................... 69
linalg_householder_product ................................ 70
linalg_inv ....................................................... 71
linalg_inv_ex ................................................ 72
linalg_lstsq .................................................. 73
linalg_matrix_norm .......................................... 75
linalg_matrix_power ......................................... 77
linalg_matrix_rank .......................................... 78
linalg_multi_dot ............................................ 79
linalg_norm ..................................................... 80
linalg_pinv ..................................................... 82
linalg_qr ......................................................... 83
linalg_slogdet ............................................... 84
linalg_solve ................................................... 85
linalg_svd ....................................................... 86
linalg_svdvals ............................................... 88
linalg_tensorinv ............................................. 89
linalg_tensorsolve .......................................... 90
linalg_vector_norm .......................................... 91
load_state_dict ............................................. 93
local_autocast ............................................. 93
local_device ............................................... 95
lr_cosine_annealing ....................................... 95
lr_lambda ..................................................... 96
lr_multiplicative .......................................... 97
lr_one_cycle ............................................... 97
lr_reduce_on_plateau ..................................... 100
lr_scheduler ................................................ 101
lr_step ........................................................ 102
topics documented:

- nnf_adaptive_avg_pool1d .................................................. 103
- nnf_adaptive_avg_pool2d .................................................. 103
- nnf_adaptive_avg_pool3d .................................................. 104
- nnf_adaptive_max_pool1d .................................................. 104
- nnf_adaptive_max_pool2d .................................................. 105
- nnf_adaptive_max_pool3d .................................................. 105
- nnf_affine_grid ............................................................ 106
- nnf_alpha_dropout .......................................................... 106
- nnf_avg_pool1d ............................................................... 107
- nnf_avg_pool2d ............................................................... 107
- nnf_avg_pool3d ............................................................... 108
- nnf_batch_norm .............................................................. 109
- nnf_bilinear ................................................................. 110
- nnf_binary_cross_entropy .................................................. 110
- nnf_binary_cross_entropy_with_logits .................................. 111
- nnf_celu ....................................................................... 112
- nnf_contrib_sparsemax .................................................... 112
- nnf_conv1d ................................................................. 113
- nnf_conv2d ................................................................. 113
- nnf_conv3d ................................................................. 114
- nnf_conv_tbc ............................................................... 115
- nnf_conv_transpose1d ...................................................... 116
- nnf_conv_transpose2d ...................................................... 117
- nnf_conv_transpose3d ...................................................... 118
- nnf_cosine_embedding_loss .............................................. 119
- nnf_cosine_similarity .................................................... 119
- nnf_cross_entropy .......................................................... 120
- nnf_ctc_loss ............................................................... 121
- nnf_dropout ............................................................... 122
- nnf_dropout2d ............................................................. 122
- nnf_dropout3d ............................................................. 123
- nnf_elu ................................................................. 123
- nnf_embedding ............................................................. 124
- nnf_embedding_bag ......................................................... 125
- nnf_fold ................................................................. 126
- nnf_fractional_max_pool2d .............................................. 127
- nnf_fractional_max_pool3d .............................................. 128
- nnf_gelu .............................................................. 129
- nnf_glu .............................................................. 129
- nnf_grid_sample ......................................................... 130
- nnf_group_norm ............................................................ 131
- nnf_gumbel_softmax .................................................... 132
- nnf_hardshrink ............................................................ 132
- nnf_hardsigmoid .......................................................... 133
- nnf_hardswish ............................................................ 133
- nnf_hardtanh ............................................................. 134
- nnf_hinge_embedding_loss .............................................. 134
- nnf_instance_norm ........................................................ 135
nnf_interpolate .................................................. 135
nnf_kl_div .......................................................... 137
nnf_l1_loss ........................................................... 137
nnf_layer_norm .................................................... 138
nnf_leaky_relu ..................................................... 138
nnf_linear ............................................................ 139
nnf_local_response_norm ......................................... 139
nnf_logsigmoid ..................................................... 140
nnf_log_softmax .................................................... 140
nnf_lp_pool1d ....................................................... 141
nnf_lp_pool2d ....................................................... 141
nnf_margin_ranking_loss .......................................... 142
nnf_max_pool1d ..................................................... 142
nnf_max_pool2d ..................................................... 143
nnf_max_pool3d ..................................................... 144
nnf_max_unpool1d ................................................... 144
nnf_max_unpool2d ................................................... 145
nnf_max_unpool3d ................................................... 146
nnf_mse_loss ........................................................ 146
nnf_multilabel_margin_loss ...................................... 147
nnf_multilabel_soft_margin_loss .................................. 147
nnf_multi_head_attention_forward ................................ 148
nnf_multi_margin_loss ............................................. 150
nnf_nll_loss ........................................................ 151
nnf_normalize ....................................................... 152
nnf_one_hot ........................................................ 152
nnf_pad ............................................................... 153
nnf_pairwise_distance .............................................. 154
nnf_pdist ............................................................ 154
nnf_pixel_shuffle ................................................... 155
nnf_poisson_nll_loss .............................................. 155
nnf_prelu ............................................................. 156
nnf_relu .............................................................. 156
nnf_relu6 ............................................................. 157
nnf_relu ............................................................... 157
nnf_selu ............................................................... 158
nnf_sigmoid ........................................................ 158
nnf_silu .............................................................. 159
nnf_smooth_l1_loss .................................................. 159
nnf_softmax ........................................................ 160
nnf_softmax ........................................................ 160
nnf_softermin ....................................................... 160
nnf_softplus ........................................................ 161
nnf_softshrink ....................................................... 162
nnf_softsign ........................................................ 162
nnf_soft_margin_loss .............................................. 163
nnf_tanhshrink ..................................................... 163
nnf_threshold ......................................................... 164
nnf_triplet_margin_loss ............................................ 164
nnf_triplet_margin_with_distance_loss
nnf_unfold
nn_adaptive_avg_pool1d
nn_adaptive_avg_pool2d
nn_adaptive_avg_pool3d
nn_adaptive_log_softmax_with_loss
nn_adaptive_max_pool1d
nn_adaptive_max_pool2d
nn_adaptive_max_pool3d
nn_avg_pool1d
nn_avg_pool2d
nn_avg_pool3d
nn_batch_norm1d
nn_batch_norm2d
nn_batch_norm3d
nn_bce_loss
nn_bce_with_logits_loss
nn_bilinear
nn_buffer
nn_celu
nn_contrib_sparsemax
nn_conv1d
nn_conv2d
nn_conv3d
nn_conv_transpose1d
nn_conv_transpose2d
nn_conv_transpose3d
nn_cosine_embedding_loss
nn_cross_entropy_loss
nn_ctc_loss
nn_dropout
nn_dropout2d
nn_dropout3d
nn_elu
nn_embedding
nn_embedding_bag
nn_flatten
nn_fractional_max_pool2d
nn_fractional_max_pool3d
nn_gelu
nn_glu
nn_group_norm
nn_gru
nn_hardshrink
nn_hardsigmoid
nn_hardswish
nnhardtanh
nn_hinge_embedding_loss
nn_identity .................................................. 224
nn_init_calculate_gain .................................. 224
nn_init_constant .......................................... 225
nn_init_dirac ............................................. 225
nn_init_eye ............................................... 226
nn_init_kaiming_normal .................................. 226
nn_init_kaiming_uniform ................................ 227
nn_init_normal .......................................... 228
nn_init_ones ............................................. 229
nn_init_orthogonal ...................................... 229
nn_init_sparse ......................................... 230
nn_init_trunc_normal ................................... 230
nn_init_uniform ........................................ 231
nn_init_xavier_normal ................................... 232
nn_init_xavier_uniform .................................. 232
nn_init_zeros .......................................... 233
nn_kl_div_loss .......................................... 233
nn_l1_loss ............................................... 234
nn_layer_norm .......................................... 235
nn_leaky_relu .......................................... 237
nn_linear ................................................ 238
nn_log_sigmoid ......................................... 239
nn_log_softmax ......................................... 239
nn_lp_pool1d ........................................... 240
nn_lp_pool2d ........................................... 241
nn_lstm .................................................. 243
nn_margin_ranking_loss ................................ 245
nn_max_pool1d .......................................... 246
nn_max_pool2d .......................................... 247
nn_max_pool3d .......................................... 248
nn_max_unpool1d ....................................... 250
nn_max_unpool2d ....................................... 251
nn_max_unpool3d ....................................... 252
nn_module ................................................ 253
nn_module_dict ......................................... 255
nn_module_list ......................................... 256
nn_mse_loss ............................................. 257
nn_multihead_attention ................................. 258
nn_multilabel_margin_loss ............................. 260
nn_multilabel_soft_margin_loss ....................... 261
nn_multi_margin_loss .................................. 262
nn_nll_loss ............................................. 263
nn_pairwise_distance ................................... 264
nn_parameter ............................................ 265
nn_poisson_nll_loss ................................... 266
nn_prelu ................................................ 267
nn_prune_head ......................................... 268
nn_relu ................................................. 269
topics documented:

- `nn_relu6` .......................................................... 269
- `nn_rnn` .......................................................... 270
- `nn_relu` .......................................................... 272
- `nn_selu` .......................................................... 273
- `nn_sequential` ..................................................... 274
- `nn_sigmoid` ....................................................... 275
- `nn_silu` .......................................................... 275
- `nn_smooth_l1_loss` ............................................ 276
- `nn_softmax` ...................................................... 277
- `nn_softmax2d` .................................................... 278
- `nn_softmin` ....................................................... 278
- `nn_softplus` ...................................................... 279
- `nn_softshrink` ................................................... 280
- `nn_softsign` ...................................................... 281
- `nn_soft_margin_loss` ......................................... 281
- `nn_tanh` .......................................................... 282
- `nn_tanhshrink` ................................................... 283
- `nn_threshold` ..................................................... 283
- `nn_triplet_margin_loss` ....................................... 284
- `nn_triplet_margin_with_distance_loss` .................. 285
- `nn_unflatten` .................................................... 287
- `nn_upsample` ..................................................... 288
- `nn_utils_clip_grad_norm_` .................................. 289
- `nn_utils_clip_grad_value_` .................................. 290
- `nn_utils_rnn_pack_padded_sequence` ..................... 290
- `nn_utils_rnn_pack_sequence` ................................ 291
- `nn_utils_rnn_pad_packed_sequence` ....................... 292
- `nn_utils_rnn_pad_sequence` ................................ 293
- `nn_utils_weight_norm` ......................................... 294
- `optimizer` ........................................................ 297
- `optim_adadelta` ................................................ 298
- `optim_adagrad` .................................................. 299
- `optim_adam` ...................................................... 301
- `optim_adamw` ..................................................... 302
- `optim_asgd` ...................................................... 302
- `optim_lbfgs` ..................................................... 303
- `optim_required` .................................................. 305
- `optim_rmsprop` ................................................... 305
- `optim_rprop` ...................................................... 307
- `optim_sgd` ........................................................ 308
- `sampler` .......................................................... 309
- `tensor_dataset` .................................................. 310
- `threads` ........................................................... 310
- `torch_abs` ........................................................ 311
- `torch_absolute` .................................................. 311
- `torch_acos` ....................................................... 312
- `torch_acosh` ...................................................... 312
- `torch_adaptive_avg_pool1d` .................................. 313
torch_add  .................................................. 314
torch_addbmm .................................................. 315
torch_adddiv .................................................. 316
torch_addcmul .................................................. 317
torch_addmm .................................................. 318
torch_addmv .................................................. 319
torch_addr ..................................................... 320
torch_allclose ................................................ 321
torch_amax ..................................................... 322
torch_amin ..................................................... 323
torch_angle ..................................................... 324
torch_arange ................................................... 324
torch_arccos ................................................... 326
torch_arccosh .................................................. 326
torch_arcsin ................................................... 327
torch_arcsinh .................................................. 327
torch_arctan ................................................... 328
torch_arctanh .................................................. 328
torch_argmax ................................................... 329
torch_argmin ................................................... 330
torch_argsort .................................................. 331
torch_asin ...................................................... 331
torch_asinh ..................................................... 332
torch_as_strided ............................................... 333
torch_atan ....................................................... 334
torch_atan2 ..................................................... 334
torch_atanh ..................................................... 335
torch_atleast_1d ............................................... 336
torch_atleast_2d ............................................... 337
torch_atleast_3d ............................................... 337
torch_avg_pool1d ............................................... 338
torch_baddbmm ............................................... 339
torch_bartlett_window ......................................... 340
torch_bernoulli ............................................... 341
torch_bincount ............................................... 342
torch_bitwise_and ............................................. 343
torch_bitwise_not ............................................. 343
torch_bitwise_or .............................................. 344
torch_bitwise_xor .............................................. 344
torch_blackman_window ........................................ 345
torch_block_diag ............................................... 346
torch_bmm ....................................................... 346
torch_broadcast_tensors ....................................... 347
torch_bucketize ................................................ 348
torch_can_cast ................................................ 349
torch_cartesian_prod ......................................... 349
torch_cat ......................................................... 350
torch_cdist ...................................................... 351
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch.ceil</td>
<td>351</td>
</tr>
<tr>
<td>torch.celu</td>
<td>352</td>
</tr>
<tr>
<td>torch.celu_</td>
<td>353</td>
</tr>
<tr>
<td>torch.chain_matmul</td>
<td>354</td>
</tr>
<tr>
<td>torch.channel_shuffle</td>
<td>355</td>
</tr>
<tr>
<td>torch.cholesky</td>
<td>356</td>
</tr>
<tr>
<td>torch.cholesky_inverse</td>
<td>357</td>
</tr>
<tr>
<td>torch.chunk</td>
<td>358</td>
</tr>
<tr>
<td>torch.clamp</td>
<td>358</td>
</tr>
<tr>
<td>torch.clip</td>
<td>359</td>
</tr>
<tr>
<td>torch.clone</td>
<td>360</td>
</tr>
<tr>
<td>torch.combinations</td>
<td>361</td>
</tr>
<tr>
<td>torch.complex</td>
<td>361</td>
</tr>
<tr>
<td>torch.conj</td>
<td>362</td>
</tr>
<tr>
<td>torch.conv1d</td>
<td>363</td>
</tr>
<tr>
<td>torch.conv2d</td>
<td>364</td>
</tr>
<tr>
<td>torch.conv3d</td>
<td>365</td>
</tr>
<tr>
<td>torch.conv_tbc</td>
<td>366</td>
</tr>
<tr>
<td>torch.conv_transpose1d</td>
<td>366</td>
</tr>
<tr>
<td>torch.conv_transpose2d</td>
<td>367</td>
</tr>
<tr>
<td>torch.conv_transpose3d</td>
<td>369</td>
</tr>
<tr>
<td>torch.cos</td>
<td>370</td>
</tr>
<tr>
<td>torch.cosh</td>
<td>371</td>
</tr>
<tr>
<td>torch.cosine_similarity</td>
<td>371</td>
</tr>
<tr>
<td>torch.count_nonzero</td>
<td>372</td>
</tr>
<tr>
<td>torch.cross</td>
<td>373</td>
</tr>
<tr>
<td>torch.cummax</td>
<td>374</td>
</tr>
<tr>
<td>torch.cummin</td>
<td>374</td>
</tr>
<tr>
<td>torch.cumprod</td>
<td>375</td>
</tr>
<tr>
<td>torch.cumsum</td>
<td>376</td>
</tr>
<tr>
<td>torch.deg2rad</td>
<td>377</td>
</tr>
<tr>
<td>torch.dequantize</td>
<td>377</td>
</tr>
<tr>
<td>torch.det</td>
<td>378</td>
</tr>
<tr>
<td>torch.device</td>
<td>379</td>
</tr>
<tr>
<td>torch.diag</td>
<td>379</td>
</tr>
<tr>
<td>torch.diagflat</td>
<td>380</td>
</tr>
<tr>
<td>torch.diagonal</td>
<td>381</td>
</tr>
<tr>
<td>torch.diag_embed</td>
<td>382</td>
</tr>
<tr>
<td>torch.diff</td>
<td>383</td>
</tr>
<tr>
<td>torch.digamma</td>
<td>384</td>
</tr>
<tr>
<td>torch.dist</td>
<td>384</td>
</tr>
<tr>
<td>torch.div</td>
<td>385</td>
</tr>
<tr>
<td>torch.divide</td>
<td>386</td>
</tr>
<tr>
<td>torch.dot</td>
<td>387</td>
</tr>
<tr>
<td>torch.dstack</td>
<td>388</td>
</tr>
<tr>
<td>torch.dtype</td>
<td>389</td>
</tr>
<tr>
<td>torch.eig</td>
<td>390</td>
</tr>
</tbody>
</table>
torch_einsum ................................................................. 390
torch_empty ............................................................... 392
torch_empty_like .......................................................... 393
torch_empty_strided ....................................................... 394
torch_eq ................................................................. 395
torch_equal .............................................................. 396
torch_erf ................................................................. 396
torch_erfc ............................................................... 397
torch_erfinv .............................................................. 398
torch_exp ................................................................. 398
torch_exp2 ............................................................... 399
torch_expm1 .............................................................. 400
torch_eye ................................................................. 400
torch_fft_fft ............................................................. 401
torch_fft_fftfreq ........................................................ 402
torch_fft_ifft ............................................................. 403
torch_fft_irfft ........................................................... 404
torch_fft_rfft ............................................................. 405
torch_finfo ............................................................... 406
torch_fix ................................................................. 406
torch_flatten ............................................................. 407
torch_flip ............................................................... 407
torch_fliplr .............................................................. 408
torch_flipud .............................................................. 409
torch_floor ............................................................... 410
torch_floor_divide ........................................................ 410
torch_fmod ............................................................... 411
torch_frac ................................................................. 412
torch_full ............................................................... 412
torch_full_like ........................................................... 413
torch_gather .............................................................. 414
torch_gcd ................................................................. 415
torch_ge ................................................................. 416
torch_generator ............................................................ 417
torch_geqrf ............................................................... 417
torch_ger ................................................................. 418
torch_get_rng_state ..................................................... 419
torch_greater ............................................................. 419
torch_greater_equal ....................................................... 420
torch_gt ................................................................. 420
torch_hamming_window .................................................... 421
torch_hann_window ....................................................... 422
torch_heaviside .......................................................... 423
torch_histc .............................................................. 424
torch_hstack ............................................................. 425
torch_hypot .............................................................. 425
torch_i0 ................................................................. 426
torch_iinfo .............................................................. 427
topics documented:

- torch_imag . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 427
- torch_index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 428
- torch_index_put . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 428
- torch_index_put_ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 429
- torch_index_select . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 429
- torch_install_path . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 430
- torch_inverse . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 430
- torch_isclose . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 431
- torch_isfinite . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 432
- torch_isinf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 433
- torch_isnan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 433
- torch_isneginf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 434
- torch_isposinf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 434
- torch_isreal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 435
- torch_istft . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 436
- torch_is_complex . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 437
- torch_is_floating_point . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 438
- torch_is_installed . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 438
- torch_is_nonzero . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 439
- torch_kaiser_window . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 439
- torch_kron . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 441
- torch_kthvalue . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 441
- torch_layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 442
- torch_lcm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 442
- torch_le . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 443
- torch_lerp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 444
- torch_less . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 445
- torch_less_equal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 445
- torch_lgamma . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 446
- torch_linspace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 446
- torch_load . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 447
- torch_log . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 448
- torch_log10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 448
- torch_log1p . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 449
- torch_log2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 450
- torch_logaddexp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 451
- torch_logaddexp2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 451
- torch_logcumsumexp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 452
- torch_logdet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 453
- torch_logical_and . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 454
- torch_logical_not . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 454
- torch_logical_or . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 455
- torch_logical_xor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 456
- torch_logit . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 456
- torch_logspace . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 457
- torch_logsumexp . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 458
- torch_lstsq . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459
- torch_lt . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 460
torch.lu ................................................................. 461
torch.lu_solve ....................................................... 461
torch.lu_unpack .................................................... 462
torch_manual_seed ................................................... 463
torch_masked_select ................................................ 464
torch_matmul ......................................................... 465
torch_matrix_exp .................................................... 466
torch_matrix_power .................................................. 467
torch_matrix_rank ................................................... 468
torch_max .............................................................. 468
torch_maximum ......................................................... 470
torch_mean ............................................................ 470
torch_median .......................................................... 471
torch_memory_format ................................................ 472
torch_meshgrid ....................................................... 473
torch_min .............................................................. 473
torch_minimum ........................................................ 475
torch_mm ................................................................. 476
torch_mode ............................................................. 476
torchMovedim .......................................................... 477
torch_mul .............................................................. 478
torch_multinomial ..................................................... 479
torch_multiply ........................................................ 480
torch_mv ................................................................. 481
torch_mvlgamma ........................................................ 481
torch_nanquantile ..................................................... 482
torch_nansum ........................................................... 483
torch_narrow ........................................................... 484
torch_ne ................................................................. 485
torch_neg ............................................................... 486
torch_negative ........................................................ 486
torch_nextafter ........................................................ 487
torch_nonzero ........................................................ 487
torch_norm .............................................................. 488
torch_normal .......................................................... 489
torch_not_equal ....................................................... 491
torch_ones .............................................................. 491
torch_ones_like ........................................................ 492
torch_orgqr ............................................................ 493
torch.ormqr ............................................................. 494
torch_outer ............................................................. 494
torch_pdist ............................................................. 495
torch_pinv .............................................................. 496
torch_pixel_shuffle ................................................... 497
torch_poisson .......................................................... 497
torch_polar ............................................................. 498
torch_polygamma ......................................................... 499
torch_pow .............................................................. 500
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>torch_prod</td>
<td>501</td>
</tr>
<tr>
<td>torch_promote_types</td>
<td>502</td>
</tr>
<tr>
<td>torch_qr</td>
<td>503</td>
</tr>
<tr>
<td>torch_qscheme</td>
<td>504</td>
</tr>
<tr>
<td>torch_quatint</td>
<td>504</td>
</tr>
<tr>
<td>torch_quantile</td>
<td>505</td>
</tr>
<tr>
<td>torch_quantize_per_channel</td>
<td>506</td>
</tr>
<tr>
<td>torch_quantize_per_tensor</td>
<td>507</td>
</tr>
<tr>
<td>torch_rad2deg</td>
<td>507</td>
</tr>
<tr>
<td>torch_rand</td>
<td>507</td>
</tr>
<tr>
<td>torch_randint</td>
<td>508</td>
</tr>
<tr>
<td>torch_randint_like</td>
<td>509</td>
</tr>
<tr>
<td>torch_randn</td>
<td>510</td>
</tr>
<tr>
<td>torch_randn_like</td>
<td>512</td>
</tr>
<tr>
<td>torch_randperm</td>
<td>513</td>
</tr>
<tr>
<td>torch_rand_like</td>
<td>514</td>
</tr>
<tr>
<td>torch_range</td>
<td>515</td>
</tr>
<tr>
<td>torch_real</td>
<td>516</td>
</tr>
<tr>
<td>torch_reciprocal</td>
<td>517</td>
</tr>
<tr>
<td>torch_reduction</td>
<td>517</td>
</tr>
<tr>
<td>torch_relu</td>
<td>518</td>
</tr>
<tr>
<td>torch_relu_</td>
<td>518</td>
</tr>
<tr>
<td>torch_remainder</td>
<td>519</td>
</tr>
<tr>
<td>torch_renorm</td>
<td>519</td>
</tr>
<tr>
<td>torch_repeat_interleave</td>
<td>520</td>
</tr>
<tr>
<td>torch_reshape</td>
<td>521</td>
</tr>
<tr>
<td>torch_result_type</td>
<td>522</td>
</tr>
<tr>
<td>torch_roll</td>
<td>523</td>
</tr>
<tr>
<td>torch_rot90</td>
<td>523</td>
</tr>
<tr>
<td>torch_round</td>
<td>524</td>
</tr>
<tr>
<td>torch_rrelu_</td>
<td>525</td>
</tr>
<tr>
<td>torch_rsqrt</td>
<td>526</td>
</tr>
<tr>
<td>torch_save</td>
<td>526</td>
</tr>
<tr>
<td>torch_scalar_tensor</td>
<td>527</td>
</tr>
<tr>
<td>torch_searchsorted</td>
<td>528</td>
</tr>
<tr>
<td>torch_selu</td>
<td>529</td>
</tr>
<tr>
<td>torch_selu_</td>
<td>529</td>
</tr>
<tr>
<td>torch_serialize</td>
<td>530</td>
</tr>
<tr>
<td>torch_set_default_dtype</td>
<td>530</td>
</tr>
<tr>
<td>torch_sgn</td>
<td>531</td>
</tr>
<tr>
<td>torch_sigmoid</td>
<td>531</td>
</tr>
<tr>
<td>torch_sigmoid</td>
<td>532</td>
</tr>
<tr>
<td>torch_signbit</td>
<td>533</td>
</tr>
<tr>
<td>torch_sign</td>
<td>533</td>
</tr>
<tr>
<td>torch_sinh</td>
<td>534</td>
</tr>
<tr>
<td>torch_slogdet</td>
<td>535</td>
</tr>
<tr>
<td>torch_sort</td>
<td>536</td>
</tr>
<tr>
<td>torch_sparse_coo_tensor</td>
<td>536</td>
</tr>
<tr>
<td>torch_split</td>
<td>538</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>torch_sqrt</td>
<td>539</td>
</tr>
<tr>
<td>torch_square</td>
<td>539</td>
</tr>
<tr>
<td>torch_squeeze</td>
<td>540</td>
</tr>
<tr>
<td>torch_stack</td>
<td>541</td>
</tr>
<tr>
<td>torch_std</td>
<td>541</td>
</tr>
<tr>
<td>torch_std_mean</td>
<td>542</td>
</tr>
<tr>
<td>torch_stft</td>
<td>543</td>
</tr>
<tr>
<td>torch_sub</td>
<td>546</td>
</tr>
<tr>
<td>torch_subtract</td>
<td>546</td>
</tr>
<tr>
<td>torch_sum</td>
<td>547</td>
</tr>
<tr>
<td>torch_svd</td>
<td>548</td>
</tr>
<tr>
<td>torch_t</td>
<td>549</td>
</tr>
<tr>
<td>torch_take</td>
<td>550</td>
</tr>
<tr>
<td>torch_tan</td>
<td>551</td>
</tr>
<tr>
<td>torch_tanh</td>
<td>552</td>
</tr>
<tr>
<td>torch_tensor</td>
<td>552</td>
</tr>
<tr>
<td>torch_tenordot</td>
<td>553</td>
</tr>
<tr>
<td>torch_tensor_from_buffer</td>
<td>554</td>
</tr>
<tr>
<td>torch_threshold</td>
<td>554</td>
</tr>
<tr>
<td>torch_topk</td>
<td>555</td>
</tr>
<tr>
<td>torch_trace</td>
<td>556</td>
</tr>
<tr>
<td>torch_transpose</td>
<td>556</td>
</tr>
<tr>
<td>torch_trapz</td>
<td>557</td>
</tr>
<tr>
<td>torch_triangular_solve</td>
<td>558</td>
</tr>
<tr>
<td>torch_tril</td>
<td>559</td>
</tr>
<tr>
<td>torch_tril_indices</td>
<td>560</td>
</tr>
<tr>
<td>torch_triu</td>
<td>561</td>
</tr>
<tr>
<td>torch_triu_indices</td>
<td>562</td>
</tr>
<tr>
<td>torch_true_divide</td>
<td>564</td>
</tr>
<tr>
<td>torch_trunc</td>
<td>564</td>
</tr>
<tr>
<td>torch_unbind</td>
<td>565</td>
</tr>
<tr>
<td>torch_unique_consecutive</td>
<td>566</td>
</tr>
<tr>
<td>torchUnsafeChunk</td>
<td>567</td>
</tr>
<tr>
<td>torchUnsafeSplit</td>
<td>567</td>
</tr>
<tr>
<td>torch_unsqueeze</td>
<td>568</td>
</tr>
<tr>
<td>torch_vander</td>
<td>569</td>
</tr>
<tr>
<td>torch_var</td>
<td>569</td>
</tr>
<tr>
<td>torch_var_mean</td>
<td>570</td>
</tr>
<tr>
<td>torch_vdot</td>
<td>571</td>
</tr>
<tr>
<td>torch_view_as_complex</td>
<td>572</td>
</tr>
<tr>
<td>torch_view_as_real</td>
<td>573</td>
</tr>
<tr>
<td>torch_vstack</td>
<td>574</td>
</tr>
<tr>
<td>torch_where</td>
<td>574</td>
</tr>
<tr>
<td>torch_zeros</td>
<td>575</td>
</tr>
<tr>
<td>torch_zeros_like</td>
<td>576</td>
</tr>
<tr>
<td>with_detect_anomaly</td>
<td>578</td>
</tr>
<tr>
<td>with_enable_grad</td>
<td>579</td>
</tr>
<tr>
<td>with_no_grad</td>
<td>580</td>
</tr>
</tbody>
</table>
as_array

<table>
<thead>
<tr>
<th>Index</th>
<th>581</th>
</tr>
</thead>
<tbody>
<tr>
<td>as_array</td>
<td>Converts to array</td>
</tr>
</tbody>
</table>

Description

Converts to array

Usage

```
as_array(x)
```

Arguments

```
x object to be converted into an array
```

AutogradContext

Class representing the context.

Description

Class representing the context.

Class representing the context.

Public fields

```
ptr (Dev related) pointer to the context c++ object.
```

Active bindings

```
needs_input_grad boolean listing arguments of forward and whether they require_grad.
saved_variables list of objects that were saved for backward via save_for_backward.
```

Methods

Public methods:

- `AutogradContext$new()`
- `AutogradContext$save_for_backward()`
- `AutogradContext$mark_non_differentiable()`
- `AutogradContext$mark_dirty()`
- `AutogradContext$clone()`

Method `new()`:

(Dev related) Initializes the context. Not user related.

Usage:

```
AutogradContext$new(
  ptr,
  env,
  argument_names = NULL,
  argument_needs_grad = NULL
)

Arguments:
ptr  pointer to the c++ object
env  environment that encloses both forward and backward
argument_names  names of forward arguments
argument_needs_grad  whether each argument in forward needs grad.

Method save_for_backward(): Saves given objects for a future call to backward().
This should be called at most once, and only from inside the forward() method.
Later, saved objects can be accessed through the saved_variables attribute. Before returning
them to the user, a check is made to ensure they weren’t used in any in-place operation that
modified their content.
Arguments can also be any kind of R object.
Usage:
AutogradContext$save_for_backward(...) 

Arguments:
  ...  any kind of R object that will be saved for the backward pass. It’s common to pass named
  arguments.

Method mark_non_differentiable(): Marks outputs as non-differentiable.
This should be called at most once, only from inside the forward() method, and all arguments
should be outputs.
This will mark outputs as not requiring gradients, increasing the efficiency of backward compu-
tation. You still need to accept a gradient for each output in backward(), but it’s always going to
be a zero tensor with the same shape as the shape of a corresponding output.
This is used e.g. for indices returned from a max Function.
Usage:
AutogradContext$mark_non_differentiable(...) 

Arguments:
  ...  non-differentiable outputs.

Method mark_dirty(): Marks given tensors as modified in an in-place operation.
This should be called at most once, only from inside the forward() method, and all arguments
should be inputs.
Every tensor that’s been modified in-place in a call to forward() should be given to this function,
to ensure correctness of our checks. It doesn’t matter whether the function is called before or after
modification.
Usage:
AutogradContext$mark_dirty(...)
autograd_backward

Arguments:
... tensors that are modified in-place.

Method clone(): The objects of this class are cloneable with this method.

Usage:
AutogradContext$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

autograd_backward Computes the sum of gradients of given tensors w.r.t. graph leaves.

Description

The graph is differentiated using the chain rule. If any of tensors are non-scalar (i.e. their data has more than one element) and require gradient, then the Jacobian-vector product would be computed, in this case the function additionally requires specifying grad_tensors. It should be a sequence of matching length, that contains the “vector” in the Jacobian-vector product, usually the gradient of the differentiated function w.r.t. corresponding tensors (None is an acceptable value for all tensors that don’t need gradient tensors).

Usage

autograd_backward(
  tensors,
  grad_tensors = NULL,
  retain_graph = create_graph,
  create_graph = FALSE
)

Arguments

tensors (list of Tensor) – Tensors of which the derivative will be computed.
grad_tensors (list of (Tensor or NULL)) The “vector” in the Jacobian-vector product, usually gradients w.r.t. the corresponding tensors.
retain_graph (bool, optional) – If FALSE, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to TRUE is not needed and often can be worked around in a much more efficient way. Defaults to the value of create_graph.
create_graph (bool, optional) – If TRUE, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to FALSE.

Details

This function accumulates gradients in the leaves - you might need to zero them before calling it.
Examples

```r
if (torch_is_installed()) {
  x <- torch_tensor(1, requires_grad = TRUE)
  y <- 2 * x

  a <- torch_tensor(1, requires_grad = TRUE)
  b <- 3 * a

  autograd_backward(list(y, b))
}
```

autograd_function  
Records operation history and defines formulas for differentiating ops.

Description

Every operation performed on Tensor's creates a new function object, that performs the computation, and records that it happened. The history is retained in the form of a DAG of functions, with edges denoting data dependencies (input <- output). Then, when backward is called, the graph is processed in the topological ordering, by calling backward() methods of each Function object, and passing returned gradients on to next Function's.

Usage

```r
autograd_function(forward, backward)
```

Arguments

- **forward**: Performs the operation. It must accept a context `ctx` as the first argument, followed by any number of arguments (tensors or other types). The context can be used to store tensors that can be then retrieved during the backward pass. See `AutogradContext` for more information about context methods.

- **backward**: Defines a formula for differentiating the operation. It must accept a context `ctx` as the first argument, followed by as many outputs as `forward()` returned (as a `list()`). The names of the arguments don't matter and they are passed in the order in which they were returned by `forward()`. The function should return a named list, where each argument is the gradient w.r.t the given output, and each element in the returned list should be the gradient w.r.t. the corresponding input. The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx$needs_input_grad` as a named list of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx$needs_input_grad$input = TRUE` if the input argument to `forward()` needs gradient computed w.r.t. the output. See `AutogradContext` for more information about context methods.
Examples

```r
if (torch_is_installed()) {

  exp2 <- autograd_function(
    forward = function(ctx, i) {
      result <- i$exp()
      ctx$save_for_backward(result = result)
      result
    },
    backward = function(ctx, grad_output) {
      list(i = grad_output * ctx$saved_variable$result)
    }
  )
}
```

autograd_grad

Computes and returns the sum of gradients of outputs w.r.t. the inputs.

Description

`grad_outputs` should be a list of length matching output containing the “vector” in Jacobian-vector product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesn’t require grad, then the gradient can be `None`.

Usage

```r
autograd_grad(
  outputs,
  inputs,
  grad_outputs = NULL,
  retain_graph = create_graph,
  create_graph = FALSE,
  allow_unused = FALSE
)
```

Arguments

- **outputs** (sequence of Tensor) – outputs of the differentiated function.
- **inputs** (sequence of Tensor) – Inputs w.r.t. which the gradient will be returned (and not accumulated into .grad).
- **grad_outputs** (sequence of Tensor) – The “vector” in the Jacobian-vector product. Usually gradients w.r.t. each output. None values can be specified for scalar Tensors or ones that don’t require grad. If a None value would be acceptable for all grad_tensors, then this argument is optional. Default: None.
- **retain_graph** (bool, optional) – If `FALSE`, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to `TRUE` is not needed and often can be worked around in a much more efficient way. Defaults to the value of create_graph.
autograd_set_grad_mode

**Create graph**

(BOOL, OPTIONAL) - IF TRUE, graph of the derivative will be constructed, allowing to compute higher order derivatives. DEFAULT: FALSE.

**Allow unused**

(BOOL, OPTIONAL) - IF FALSE, specifying inputs that were not used when computing outputs (and therefore their grad is always zero) is an error. DEFAULT: FALSE.

**Details**

If only_inputs is TRUE, the function will only return a list of gradients w.r.t the specified inputs. If it’s FALSE, then gradient w.r.t. all remaining leaves will still be computed, and will be accumulated into their .grad attribute.

**Examples**

```r
if (torch_is_installed()) {
  w <- torch_tensor(0.5, requires_grad = TRUE)
  b <- torch_tensor(0.9, requires_grad = TRUE)
  x <- torch_tensor(runif(100))
  y <- 2 * x + 1
  loss <- (y - (w * x + b))^2
  loss <- loss$mean()

  o <- autograd_grad(loss, list(w, b))
  o
}
```

---

autograd_set_grad_mode

*Set grad mode*

**Description**

Sets or disables gradient history.

**Usage**

autograd_set_grad_mode(enabled)

**Arguments**

enabled (BOOL) - bool wether to enable or disable the gradient recording.
### backends_cudnn_is_available

**CuDNN is available**

#### Description

CuDNN is available

#### Usage

```python
backends_cudnn_is_available()
```

### backends_cudnn_version

**CuDNN version**

#### Description

CuDNN version

#### Usage

```python
backends_cudnn_version()
```

### backends_mkldnn_is_available

**MKLDNN is available**

#### Description

MKLDNN is available

#### Usage

```python
backends_mkldnn_is_available()
```

#### Value

Returns whether LibTorch is built with MKL-DNN support.
backends_mkl_is_available

MKL is available

Description
MKL is available

Usage
backends_mkl_is_available()

Value
Returns whether LibTorch is built with MKL support.

backends_mps_is_available

MPS is available

Description
MPS is available

Usage
backends_mps_is_available()

Value
Returns whether LibTorch is built with MPS support.

backends_openmp_is_available

OpenMP is available

Description
OpenMP is available

Usage
backends_openmp_is_available()

Value
Returns whether LibTorch is built with OpenMP support.
Given a list of values (possibly containing numbers), returns a list where each value is broadcasted based on the following rules:

**Description**

Raises `value_error` if any of the values is not a numeric instance, a `torch.*Tensor` instance, or an instance implementing `torch_function` TODO: add `has_torch_function((v,))` See: https://github.com/pytorch/pytorch/blob/master/torch/distributions/utils.py

**Usage**

`broadcast_all(values)`

**Arguments**

- `values`: List of:
  - `torch.*Tensor` instances are broadcasted as per `broadcasting-semantics`.
  - numeric instances (scalars) are upcast to tensors having the same size and type as the first tensor passed to `values`. If all the values are scalars, then they are upcasted to scalar Tensors. values (list of numeric, `torch.*Tensor` or objects implementing `torch_function`)

**Constraint**

Abstract base class for constraints.

**Description**

Abstract base class for constraints.

Abstract base class for constraints.

**Details**

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

**Methods**

**Public methods:**

- `Constraint$check()`
- `Constraint$print()`
- `Constraint$clone()`

**Method** `check()`: Returns a byte tensor of `sample_shape + batch_shape` indicating whether each event in `value` satisfies this constraint.
Usage:
Constraint$check(value)

Arguments:
value each event in value will be checked.

Method print(): Define the print method for constraints,
Usage:
Constraint$print()

Method clone(): The objects of this class are cloneable with this method.
Usage:
Constraint$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

---

contrib_sort_vertices Contrib sort vertices

Description
Based on the implementation from Rotated_IoU

Usage
contrib_sort_vertices(vertices, mask, num_valid)

Arguments
vertices A Tensor with the vertices.
mask A tensors containing the masks.
um_valid A integer tensors.

Details
All tensors should be on a CUDA device so this function can be used.

Note
This function does not make part of the official torch API.
**cuda_amp_grad_scaler**  

**Examples**

```r
define_examples() {
  if (torch_is_installed()) {
    if (cuda_is_available()) {
      v <- torch_randn(8, 1024, 24, 2)$cuda()
      mean <- torch_mean(v, dim = 2, keepdim = TRUE)
      v <- v - mean
      m <- (torch_rand(8, 1024, 24) > 0.8)$cuda()
      nv <- torch_sum(m$to(dtype = torch_int()), dim = -1)$to(dtype = torch_int())$cuda()
      result <- contrib_sort_vertices(v, m, nv)
    }
  }
}
```

**cuda_amp_grad_scaler**  

*Creates a gradient scaler*

**Description**

A gradient scaler instance is used to perform dynamic gradient scaling to avoid gradient underflow when training with mixed precision.

**Usage**

```r
cuda_amp_grad_scaler(
  init_scale = 2^16,
  growth_factor = 2,
  backoff_factor = 0.5,
  growth_interval = 2000,
  enabled = TRUE
)
```

**Arguments**

- `init_scale`  
  a numeric value indicating the initial scale factor.
- `growth_factor`  
  a numeric value indicating the growth factor.
- `backoff_factor`  
  a numeric value indicating the backoff factor.
- `growth_interval`  
  a numeric value indicating the growth interval.
- `enabled`  
  a logical value indicating whether the gradient scaler should be enabled.

**Value**

A gradient scaler object.
cuda_current_device | Returns the index of a currently selected device.

**Description**

Returns the index of a currently selected device.

**Usage**

cuda_current_device()

---

cuda_device_count | Returns the number of GPUs available.

**Description**

Returns the number of GPUs available.

**Usage**

cuda_device_count()

---

cuda_empty_cache | Empty cache

**Description**

Releases all unoccupied cached memory currently held by the caching allocator so that those can be used in other GPU application and visible in nvidia-smi.

**Usage**

cuda_empty_cache()

**Note**

cuda_empty_cache() doesn’t increase the amount of GPU memory available for torch. However, it may help reduce fragmentation of GPU memory in certain cases. See Memory management article for more details about GPU memory management.
cuda_get_device_capability

Returns the major and minor CUDA capability of device

Description

Returns the major and minor CUDA capability of device.

Usage

cuda_get_device_capability(device = cuda_current_device())

Arguments

device

Integer value of the CUDA device to return capabilities of.

cuda_is_available

Returns a bool indicating if CUDA is currently available.

Description

Returns a bool indicating if CUDA is currently available.

Usage

cuda_is_available()

cuda_memory_stats

Returns a dictionary of CUDA memory allocator statistics for a given device.

Description

The return value of this function is a dictionary of statistics, each of which is a non-negative integer.

Usage

cuda_memory_stats(device = cuda_current_device())

cuda_memory_summary(device = cuda_current_device())

Arguments

device

Integer value of the CUDA device to return capabilities of.
Core statistics

- "allocated.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": number of allocation requests received by the memory allocator.
- "allocated_bytes.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": amount of allocated memory.
- "segment.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": number of reserved segments from cudaMalloc().
- "reserved_bytes.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": amount of reserved memory.
- "active.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": number of active memory blocks.
- "active_bytes.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": amount of active memory.
- "inactive_split.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": number of inactive, non-releasable memory blocks.
- "inactive_split_bytes.\{all,large_pool,small_pool\}.\{current,peak,allocated,freed\}": amount of inactive, non-releasable memory.

For these core statistics, values are broken down as follows.

Pool type:

- all: combined statistics across all memory pools.
- large_pool: statistics for the large allocation pool (as of October 2019, for size >= 1MB allocations).
- small_pool: statistics for the small allocation pool (as of October 2019, for size < 1MB allocations).

Metric type:

- current: current value of this metric.
- peak: maximum value of this metric.
- allocated: historical total increase in this metric.
- freed: historical total decrease in this metric.

Additional metrics

- "num_alloc_retries": number of failed cudaMalloc calls that result in a cache flush and retry.
- "num_ooms": number of out-of-memory errors thrown.
cuda_runtime_version

Returns the CUDA runtime version

**Description**

Returns the CUDA runtime version

**Usage**

```
cuda_runtime_version()
```

---

cuda_synchronize

Waits for all kernels in all streams on a CUDA device to complete.

**Description**

Waits for all kernels in all streams on a CUDA device to complete.

**Usage**

```
cuda_synchronize(device = NULL)
```

**Arguments**

```
device device for which to synchronize. It uses the current device given by cuda_current_device() if no device is specified.
```

dataloader

Data loader. Combines a dataset and a sampler, and provides single- or multi-process iterators over the dataset.

**Description**

Data loader. Combines a dataset and a sampler, and provides single- or multi-process iterators over the dataset.
Usage

dataloader(
    dataset,
    batch_size = 1,
    shuffle = FALSE,
    sampler = NULL,
    batch_sampler = NULL,
    num_workers = 0,
    collate_fn = NULL,
    pin_memory = FALSE,
    drop_last = FALSE,
    timeout = -1,
    worker_init_fn = NULL,
    worker_globals = NULL,
    worker_packages = NULL
)

Arguments

dataset (Dataset): dataset from which to load the data.
batch_size (int, optional): how many samples per batch to load (default: 1).
shuffle (bool, optional): set to TRUE to have the data reshuffled at every epoch (default: FALSE).
sampler (Sampler, optional): defines the strategy to draw samples from the dataset. If specified, shuffle must be False. Custom samplers can be created with sampler().
batch_sampler (Sampler, optional): like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last. Custom samplers can be created with sampler().
num_workers (int, optional): how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
collate_fn (callable, optional): merges a list of samples to form a mini-batch.
pin_memory (bool, optional): If TRUE, the data loader will copy tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type see the example below.
drop_last (bool, optional): set to TRUE to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If FALSE and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: FALSE)
timeout (numeric, optional): if positive, the timeout value for collecting a batch from workers. -1 means no timeout. (default: -1)
worker_init_fn (callable, optional): If not NULL, this will be called on each worker subprocess with the worker id (an int in [1, num_workers]) as input, after seeding and before data loading. (default: NULL)
worker_globals (list or character vector, optional) only used when num_workers > 0. If a character vector, then objects with those names are copied from the global environment
to the workers. If a named list, then this list is copied and attached to the worker
global environment. Notice that the objects are copied only once at the worker
initialization.

worker_packages
(character vector, optional) Only used if num_workers > 0 optional character
vector naming packages that should be loaded in each worker.

Parallel data loading

When using num_workers > 0 data loading will happen in parallel for each worker. Note that
batches are taken in parallel and not observations.

The worker initialization process happens in the following order:

• num_workers R sessions are initialized.

Then in each worker we perform the following actions:

• the torch library is loaded.
• a random seed is set both using set.seed() and using torch_manual_seed.
• packages passed to the worker_packages argument are loaded.
• objects passed trough the worker_globals parameters are copied into the global environment.
• the worker_init function is ran with an id argument.
• the dataset fetcher is copied to the worker.

See Also

dataset(), sampler()
### dataloader_next

*Get the next element of a dataloader iterator*

**Description**

Get the next element of a dataloader iterator

**Usage**

```
dataloader_next(iter, completed = NULL)
```

**Arguments**

- `iter` a DataLoader iter created with `dataloader_make_iter`.
- `completed` the returned value when the iterator is exhausted.

---

### dataset

*Helper function to create an R6 class*

**Description**

All datasets that represent a map from keys to data samples should subclass this class. All subclasses should overwrite the `.getitem()` method, which supports fetching a data sample for a given key. Subclasses could also optionally overwrite `.length()`, which is expected to return the size of the dataset (e.g. number of samples) used by many sampler implementations and the default options of `dataloader()`.

**Usage**

```
dataset(
  name = NULL,
  inherit = Dataset,
  ..., 
  private = NULL,
  active = NULL,
  parent_env = parent.frame()
)
```

**Arguments**

- `name` a name for the dataset. It's also used as the class for it.
- `inherit` you can optionally inherit from a dataset when creating a new dataset.
- `...` public methods for the dataset class
- `private` passed to `R6::R6Class()`.
- `active` passed to `R6::R6Class()`.
- `parent_env` An environment to use as the parent of newly-created objects.
The output is a function $f$ with class `dataset_generator`. Calling $f()$ creates a new instance of the R6 class `dataset`. The R6 class is stored in the enclosing environment of $f$ and can also be accessed through $f$'s attribute `Dataset`.

### Get a batch of observations

By default datasets are iterated by returning each observation/item individually. Often it’s possible to have an optimized implementation to take a batch of observations (eg, subsetting a tensor by multiple indexes at once is faster than subsetting once for each index), in this case you can implement a `.getbatch` method that will be used instead of `.getitem` when getting a batch of observations within the dataloader. `.getbatch` must work for batches of size larger or equal to 1 and care must be taken so it doesn’t drop the batch dimension when it’s queried with a length 1 batch index - for instance by using `drop=FALSE`. `.getitem()` is expected to not include the batch dimension as it’s added by the dataloader. For more on this see the vignette(`loading-data`).

### Note

`dataloader()` by default constructs a index sampler that yields integral indices. To make it work with a map-style dataset with non-integral indices/keys, a custom sampler must be provided.

---

### dataset_subset

**Dataset Subset**

**Description**

Subset of a dataset at specified indices.

**Usage**

```r
dataset_subset(dataset, indices)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dataset</code></td>
<td>(Dataset): The whole Dataset</td>
</tr>
<tr>
<td><code>indices</code></td>
<td>(sequence): Indices in the whole set selected for subset</td>
</tr>
</tbody>
</table>
Distribution is the abstract base class for probability distributions. Note: in Python, adding torch.Size objects works as concatenation. Try for example: torch.Size((2, 1)) + torch.Size((1,))

Public fields

- `.validate_args` whether to validate arguments
- `has_rsample` whether has an rsample
- `has_enumerate_support` whether has enumerate support

Active bindings

- `batch_shape` Returns the shape over which parameters are batched.
- `event_shape` Returns the shape of a single sample (without batching). Returns a dictionary from argument names to torch.Constraint objects that should be satisfied by each argument of this distribution. Args that are not tensors need not appear in this dict.
- `support` Returns a torch.Constraint object representing this distribution’s support.
- `mean` Returns the mean on of the distribution
- `variance` Returns the variance of the distribution
- `stddev` Returns the standard deviation of the distribution TODO: consider different message

Methods

Public methods:

- `Distribution$new()`
- `Distributionexpand()`
- `Distribution$sample()`
- `Distribution$rsample()`
- `Distribution$log_prob()`
- `Distribution$cdf()`
- `Distribution$icdf()`
- `Distribution$enumerate_support()`
- `Distribution$entropy()`
- `Distribution$perplexity()`
- `Distribution$.extended_shape()`
- `Distribution$.validate_sample()`
- `Distribution$print()`
- `Distribution$clone()`
**Method** `new()`: Initializes a distribution class.

*Usage:*
```
Distribution$new(batch_shape = NULL, event_shape = NULL, validate_args = NULL)
```

*Arguments:*
- `batch_shape` the shape over which parameters are batched.
- `event_shape` the shape of a single sample (without batching).
- `validate_args` whether to validate the arguments or not. Validation can be time consuming so you might want to disable it.

**Method** `expand()`: Returns a new distribution instance (or populates an existing instance provided by a derived class) with batch dimensions expanded to `batch_shape`. This method calls `expand` on the distribution’s parameters. As such, this does not allocate new memory for the expanded distribution instance. Additionally, this does not repeat any `args` checking or parameter broadcasting in `initialize`, when an instance is first created.

*Usage:*
```
Distribution$expand(batch_shape, .instance = NULL)
```

*Arguments:*
- `batch_shape` the desired expanded size.
- `instance` new instance provided by subclasses that need to override `expand`.

**Method** `sample()`: Generates a `sample_shape` shaped sample or `sample_shape` shaped batch of samples if the distribution parameters are batched.

*Usage:*
```
Distribution$sample(sample_shape = NULL)
```

*Arguments:*
- `sample_shape` the shape you want to sample.

**Method** `rsample()`: Generates a `sample_shape` shaped reparameterized sample or `sample_shape` shaped batch of reparameterized samples if the distribution parameters are batched.

*Usage:*
```
Distribution$rsample(sample_shape = NULL)
```

*Arguments:*
- `sample_shape` the shape you want to sample.

**Method** `log_prob()`: Returns the log of the probability density/mass function evaluated at `value`.

*Usage:*
```
Distribution$log_prob(value)
```

*Arguments:*
- `value` values to evaluate the density on.

**Method** `cdf()`: Returns the cumulative density/mass function evaluated at `value`.

*Usage:*
```
```
Distribution$\text{cdf}(\text{value})

**Arguments:**
value values to evaluate the density on.

**Method icdf():** Returns the inverse cumulative density/mass function evaluated at value.

@description Returns tensor containing all values supported by a discrete distribution. The result will enumerate over dimension 0, so the shape of the result will be (cardinality,) + batch_shape + event_shape (where event_shape = () for univariate distributions). Note that this enumerates over all batched tensors in lock-step list(c(0, 0), c(1, 1), ...). With expand=FALSE, enumeration happens along dim 0, but with the remaining batch dimensions being singleton dimensions, list(c(0), c(1), ...).

**Usage:**
Distribution$\text{icdf}(\text{value})

**Arguments:**
value values to evaluate the density on.

**Method enumerate_support():**

**Usage:**
Distribution$\text{enumerate_support}(\text{expand} = \text{TRUE})

**Arguments:**
expand (bool): whether to expand the support over the batch dims to match the distribution’s batch_shape.

**Returns:** Tensor iterating over dimension 0.

**Method entropy():** Returns entropy of distribution, batched over batch_shape.

**Usage:**
Distribution$\text{entropy()}

**Returns:** Tensor of shape batch_shape.

**Method perplexity():** Returns perplexity of distribution, batched over batch_shape.

**Usage:**
Distribution$\text{perplexity()}

**Returns:** Tensor of shape batch_shape.

**Method .extended_shape():** Returns the size of the sample returned by the distribution, given a sample_shape. Note, that the batch and event shapes of a distribution instance are fixed at the time of construction. If this is empty, the returned shape is upcast to (1,).

**Usage:**
Distribution$.\text{extended_shape}(\text{sample_shape} = \text{NULL})

**Arguments:**
sample_shape (torch_Size): the size of the sample to be drawn.

**Method .validate_sample():** Argument validation for distribution methods such as log_prob, cdf and icdf. The rightmost dimensions of a value to be scored via these methods must agree with the distribution’s batch and event shapes.
Usage:
Distribution$.validate_sample(value)

Arguments:
value (Tensor): the tensor whose log probability is to be computed by the log_prob method.

Method print(): Prints the distribution instance.

Usage:
Distribution$print()

Method clone(): The objects of this class are cloneable with this method.

Usage:
Distribution$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

```r


distr_bernoulli

Creates a Bernoulli distribution parameterized by probs or logits (but not both). Samples are binary (0 or 1). They take the value 1 with probability $p$ and 0 with probability $1 - p$.

Description

Creates a Bernoulli distribution parameterized by probs or logits (but not both). Samples are binary (0 or 1). They take the value 1 with probability $p$ and 0 with probability $1 - p$.

Usage

distr_bernoulli(probs = NULL, logits = NULL, validate_args = NULL)

Arguments

probs (numeric or torch_tensor): the probability of sampling 1
logits (numeric or torch_tensor): the log-odds of sampling 1
validate_args whether to validate arguments or not.

See Also

Distribution for details on the available methods.

Other distributions: distr_chisq(), distr_gamma(), distr_multivariate_normal(), distr_normal(), distr_poisson()

Examples

```r
if (torch_is_installed()) {
  m <- distr_bernoulli(0.3)
  m$sample() # 30% chance 1; 70% chance 0
}
```
distr_categorical

**Description**

Creates a categorical distribution parameterized by either `probs` or `logits` (but not both).

**Usage**

```r
distr_categorical(probs = NULL, logits = NULL, validate_args = NULL)
```

**Arguments**

- `probs` (Tensor): event probabilities
- `logits` (Tensor): event log probabilities (unnormalized)
- `validate_args` Additional arguments

**Note**

It is equivalent to the distribution that `torch_multinomial()` samples from.

Samples are integers from \{0, ..., K - 1\} where K is `probs$size(-1)`.

If `probs` is 1-dimensional with length-K, each element is the relative probability of sampling the class at that index.

If `probs` is N-dimensional, the first N-1 dimensions are treated as a batch of relative probability vectors.

The `probs` argument must be non-negative, finite and have a non-zero sum, and it will be normalized to sum to 1 along the last dimension. `attr:probs` will return this normalized value. The `logits` argument will be interpreted as unnormalized log probabilities and can therefore be any real number. It will likewise be normalized so that the resulting probabilities sum to 1 along the last dimension. `attr:logits` will return this normalized value.

See also: `torch_multinomial()`

**Examples**

```r
if (torch_is_installed()) {
  m <- distr_categorical(torch_tensor(c(0.25, 0.25, 0.25, 0.25)))
  mSAMPLE() # equal probability of 1,2,3,4
}
```
distr_chi2

*Create a Chi2 distribution parameterized by shape parameter df. This is exactly equivalent to distr_gamma(alpha=0.5*df, beta=0.5)*

---

**Description**

Creates a Chi2 distribution parameterized by shape parameter df. This is exactly equivalent to distr_gamma(alpha=0.5*df, beta=0.5)

**Usage**

```r
distr_chi2(df, validate_args = NULL)
```

**Arguments**

- `df`: (float or torch_tensor): shape parameter of the distribution
- `validate_args`: whether to validate arguments or not.

**See Also**

Distribution for details on the available methods.

Other distributions: distr_bernoulli(), distr_gamma(), distr_multivariate_normal(), distr_normal(), distr_poisson()

**Examples**

```r
if (torch_is_installed()) {
  m <- distr_chi2(torch_tensor(1.0))
  m$sample() # Chi2 distributed with shape df=1
  torch_tensor(0.1046)
}
```

---

distr_gamma

*Create a Gamma distribution parameterized by shape concentration and rate.*

---

**Description**

Creates a Gamma distribution parameterized by shape concentration and rate.

**Usage**

```r
distr_gamma(concentration, rate, validate_args = NULL)
```
distr_mixture_same_family

**Arguments**

- **concentration** (float or Tensor): shape parameter of the distribution (often referred to as alpha)
- **rate** (float or Tensor): rate = 1 / scale of the distribution (often referred to as beta)
- **validate_args** whether to validate arguments or not.

**See Also**

`Distribution` for details on the available methods.

Other distributions: `distr_bernoulli()`, `distr_chi2()`, `distr_multivariate_normal()`, `distr_normal()`, `distr_poisson()`

**Examples**

```r
if (torch_is_installed()) {
  m <- distr_gamma(torch_tensor(1.0), torch_tensor(1.0))
  m$sample() # Gamma distributed with concentration=1 and rate=1
}
```

---

**distr_mixture_same_family**

*Mixture of components in the same family*

**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 component) and a component distribution, i.e., a `Distribution` with a rightmost batch shape (equal to [k]) which indexes each (batch of) component.

**Usage**

```r
distr_mixture_same_family(
  mixture_distribution,
  component_distribution,
  validate_args = NULL
)
```

**Arguments**

- **mixture_distribution** `torch_distributions.Categorical-like` instance. Manages the probability of selecting component. The number of categories must match the rightmost batch dimension of the component_distribution. Must have either scalar batch_shape or batch_shape matching component_distribution.batch_shape[:-1]
distr_multivariate_normal

component_distribution

    torch_distributions.Distribution-like instance. Right-most batch dimension indexes component.

validate_args  Additional arguments

Examples

    if (torch_is_installed()) {
        # Construct Gaussian Mixture Model in 1D consisting of 5 equally
        # weighted normal distributions
        mix <- distr_categorical(torch_ones(5))
        comp <- distr_normal(torch_randn(5), torch_rand(5))
        gmm <- distr_mixture_same_family(mix, comp)
    }

---

distr_multivariate_normal

_Gaussian distribution_

Description

Creates a multivariate normal (also called Gaussian) distribution parameterized by a mean vector and a covariance matrix.

Usage

    distr_multivariate_normal(
        loc,
        covariance_matrix = NULL,
        precision_matrix = NULL,
        scale_tril = NULL,
        validate_args = NULL
    )

Arguments

    loc  (Tensor): mean of the distribution
    covariance_matrix  (Tensor): positive-definite covariance matrix
    precision_matrix  (Tensor): positive-definite precision matrix
    scale_tril  (Tensor): lower-triangular factor of covariance, with positive-valued diagonal
    validate_args  Bool wether to validate the arguments or not.
Details

The multivariate normal distribution can be parameterized either in terms of a positive definite covariance matrix $\Sigma$ or a positive definite precision matrix $\Sigma^{-1}$ or a lower-triangular matrix $L$ with positive-valued diagonal entries, such that $\Sigma = LL^\top$. This triangular matrix can be obtained via e.g. Cholesky decomposition of the covariance.

Note

Only one of covariance_matrix or precision_matrix or scale_tril can be specified. Using scale_tril will be more efficient: all computations internally are based on scale_tril. If covariance_matrix or precision_matrix is passed instead, it is only used to compute the corresponding lower triangular matrices using a Cholesky decomposition.

See Also

Distribution for details on the available methods.

Other distributions: distr_beroulli(), distr_chisq(), distr_gaussian(), distr_normal(), distr_poisson()

Examples

if (torch_is_installed()) {
  m <- distr_multivariate_normal(torch_zeros(2), torch_eye(2))
  m$sample() # normally distributed with mean=[0,0] and covariance_matrix=I
}

---

distr_normal

Creates a normal (also called Gaussian) distribution parameterized by loc and scale.

Description

Creates a normal (also called Gaussian) distribution parameterized by loc and scale.

Usage

distr_normal(loc, scale, validate_args = NULL)

Arguments

loc (float or Tensor): mean of the distribution (often referred to as mu)

scale (float or Tensor): standard deviation of the distribution (often referred to as sigma)

validate_args Additional arguments

Value

Object of torch_Normal class
distr_poisson

See Also

Distribution for details on the available methods.

Other distributions: distr_bernoulli(), distr_chisq(), distr_normal(), distr_multivariate_normal(), distr_poisson()

Examples

if (torch_is_installed()) {
  m <- distr_normal(loc = 0, scale = 1)
  m$sample() # normally distributed with loc=0 and scale=1
}


distr_poisson

Creates a Poisson distribution parameterized by rate, the rate parameter.

Description

Samples are nonnegative integers, with a pmf given by

\[ \text{rate}^k \frac{e^{-\text{rate}}}{k!} \]

Usage

distr_poisson(rate, validate_args = NULL)

Arguments

rate (numeric, torch_tensor): the rate parameter
validate_args whether to validate arguments or not.

See Also

Distribution for details on the available methods.

Other distributions: distr_bernoulli(), distr_chisq(), distr_normal(), distr_multivariate_normal(), distr_normal()

Examples

if (torch_is_installed()) {
  m <- distr_poisson(torch_tensor(4))
  m$sample()
}
### enumerate

**Description**

Enumerate an iterator

**Usage**

```r
enumerate(x, ...)  
```

**Arguments**

- `x`: the generator to enumerate.
- `...`: passed to specific methods.

---

### enumerate.dataloader

**Description**

Enumerate an iterator

**Usage**

```r
# S3 method for class 'dataloader'
enumerate(x, max_len = 1e+06, ...)  
```

**Arguments**

- `x`: the generator to enumerate.
- `max_len`: maximum number of iterations.
- `...`: passed to specific methods.
Description

List the Torch and Lantern libraries URLs to download as local files in order to proceed with `install_torch_from_file()`.

Installs Torch and its dependencies from files.

Usage

```r
get_install_libs_url(version = NA, type = NA)
```

```r
install_torch_from_file(version = NA, type = NA, libtorch, liblantern, ...)
```

Arguments

- **version**: Not used
- **type**: Not used. This function is deprecated.
- **libtorch**: The installation archive file to use for Torch. Shall be a "file://" URL scheme.
- **liblantern**: The installation archive file to use for Lantern. Shall be a "file://" URL scheme.
- **...**: other parameters to be passed to "install_torch()"

Details

When "install_torch()" initiated download is not possible, but installation archive files are present on local filesystem, "install_torch_from_file()" can be used as a workaround to installation issue. "libtorch" is the archive containing all torch modules, and "liblantern" is the C interface to libtorch that is used for the R package. Both are highly dependent, and should be checked through "get_install_libs_url()"

Examples

```r
if (torch_is_installed()) {
## Not run:
# on a linux CPU platform
get_install_libs_url()
# then after making both files available into /tmp/
Sys.setenv(TORCH_URL="/tmp/libtorch-v1.13.1.zip")
Sys.setenv(LANTERN_URL="/tmp/lantern-0.9.1.9001+cpu+arm64-Darwin.zip")
torch::install_torch()

## End(Not run)
}
```
install_torch  

Install Torch

Description
Installs Torch and its dependencies.

Usage
install_torch(reinstall = FALSE, ..., .inform_restart = TRUE)

Arguments
reinstall  Re-install Torch even if its already installed?
...  Currently unused.
.inform_restart  if TRUE and running in an interactive() session, after installation it will print a message to inform the user that the session must be restarted for torch to work correctly.

Details
This function is mainly controlled by environment variables that can be used to override the defaults:

- TORCH_HOME: the installation path. By default dependencies are installed within the package directory. Eg what’s given by system.file(package="torch").
- TORCH_URL: A URL, path to a ZIP file or a directory containing a LibTorch version. Files will be installed/copied to the TORCH_HOME directory.
- LANTERN_URL: Same as TORCH_URL but for the Lantern library.
- TORCH_INSTALL_DEBUG: Setting it to 1, shows debug log messages during installation.
- PRECXX11ABI: Setting it to 1 will will trigger the installation of a Pre-cxx11 ABI installation of LibTorch. This can be useful in environments with older versions of GLIBC like CentOS7 and older Debian/Ubuntu versions.
- LANTERN_BASE_URL: The base URL for lantern files. This allows passing a directory where lantern binaries are located. The filename is then constructed as usual.
- TORCH_COMMIT_SHA: torch repository commit sha to be used when querying lantern uploads. Set it to 'none' to avoid looking for build for that commit and use the latest build for the branch.
- CUDA: We try to automatically detect the CUDA version installed in your system, but you might want to manually set it here. You can also disable CUDA installation by setting it to 'cpu'.
- TORCH_R_VERSION: The R torch version. It’s unlikely that you need to change it, but it can be useful if you don’t have the R package installed, but want to install the dependencies.
The TORCH_INSTALL environment variable can be set to 0 to prevent auto-installing torch and TORCH_LOAD set to 0 to avoid loading dependencies automatically. These environment variables are meant for advanced use cases and troubleshooting only. When timeout error occurs during library archive download, or length of downloaded files differ from reported length, an increase of the timeout value should help.

---

**is_dataloader**

*Checks if the object is a dataloader*

**Description**

Checks if the object is a dataloader

**Usage**

```python
is_dataloader(x)
```

**Arguments**

- `x` object to check

---

**is_nn_buffer**

*Checks if the object is a nn_buffer*

**Description**

Checks if the object is a nn_buffer

**Usage**

```python
is_nn_buffer(x)
```

**Arguments**

- `x` object to check
is_nn_module

Checks if the object is an nn_module

Description
Checks if the object is an nn_module

Usage
is_nn_module(x)

Arguments
x object to check

is_nn_parameter

Checks if an object is a nn_parameter

Description
Checks if an object is a nn_parameter

Usage
is_nn_parameter(x)

Arguments
x the object to check

is_optimizer

Checks if the object is a torch optimizer

Description
Checks if the object is a torch optimizer

Usage
is_optimizer(x)

Arguments
x object to check
is_torch_device

Checks if object is a device

Description
Checks if object is a device

Usage
is_torch_device(x)

Arguments
x object to check

is_torch_dtype
Check if object is a torch data type

Description
Check if object is a torch data type

Usage
is_torch_dtype(x)

Arguments
x object to check

is_torch_layout
Check if an object is a torch layout.

Description
Check if an object is a torch layout.

Usage
is_torch_layout(x)

Arguments
x object to check
**is_torch_memory_format**

*Check if an object is a memory format*

**Description**

Check if an object is a memory format

**Usage**

```python
is_torch_memory_format(x)
```

**Arguments**

- `x` : object to check

**is_torch_qscheme**

*Checks if an object is a QScheme*

**Description**

Checks if an object is a QScheme

**Usage**

```python
is_torch_qscheme(x)
```

**Arguments**

- `x` : object to check

**is_undefined_tensor**

*Checks if a tensor is undefined*

**Description**

Checks if a tensor is undefined

**Usage**

```python
is_undefined_tensor(x)
```

**Arguments**

- `x` : tensor to check
iterable_dataset  Creates an iterable dataset

Description

Creates an iterable dataset

Usage

iterable_dataset(
  name,
  inherit = IterableDataset,
  ...,  
  private = NULL,
  active = NULL,
  parent_env = parent.frame()
)

Arguments

name  a name for the dataset. It’s also used as the class for it.
inherit  you can optionally inherit from a dataset when creating a new dataset.
...  public methods for the dataset class
private  passed to R6::R6Class().
active  passed to R6::R6Class().
parent_env  An environment to use as the parent of newly-created objects.

Examples

if (torch_is_installed()) {
  ids <- iterable_dataset(
    name = "hello",
    initialize = function(n = 5) {
      self$n <- n
      self$i <- 0
    },
    .iter = function() {
      i <- 0
      function() {
        i <<- i + 1
        if (i > self$n) {
          coro::exhausted()
        } else {
          i
        }
      }
    }
  )
}
jit_load

Description

Loads a script_function or script_module previously saved with jit_save.

Usage

jit_load(path, ...)

jit_compile

Compile TorchScript code into a graph

Description

See the TorchScript language reference for documentation on how to write TorchScript code.

Usage

jit_compile(source)

Arguments

source valid TorchScript source code.

Examples

if (torch_is_installed()) {
    comp <- jit_compile(""
    def fn (x):
        return torch.abs(x)

    def foo (x):
        return torch.sum(x)

    ")
    comp$fn(torch_tensor(-1))
    comp$foo(torch_randn(10))
}

jit_load

Loads a script_function or script_module previously saved with jit_save.
Arguments

path: a path to a script_function or script_module serialized with `jit_save()`. ... currently unused.

jit_ops

Enable idiomatic access to JIT operators from R.

Description

Call JIT operators directly from R, keeping the familiar argument types and argument order. Note, however, that:

- all arguments are required (no defaults)
- axis numbering (as well as position numbers overall) starts from 0
- scalars have to be wrapped in `jit_scalar()`

Usage

`jit_ops`

Format

An object of class `torch_ops` of length 0.

Examples

```r
if (torch_is_installed()) {
  t1 <- torch::torch_rand(4, 5)
  t2 <- torch::torch_ones(5, 4)
  # same as torch::torch_matmul(t1, t2)
  jit_ops$aten$matmul(t1, t2)

  # same as torch_split(torch::torch_arange(0, 3), 2, 1)
  jit_ops$aten$split(torch::torch_arange(0, 3), torch::jit_scalar(2L), torch::jit_scalar(0L))
}
```
jit_save

Saves a script_function to a path

**Description**

Saves a script_function to a path

**Usage**

```r
jit_save(obj, path, ...)
```

**Arguments**

- `obj`: An script_function to save
- `path`: The path to save the serialized function.
- `...`: currently unused

**Examples**

```r
if (torch_is_installed()) {
  fn <- function(x) {
    torch_relu(x)
  }

  input <- torch_tensor(c(-1, 0, 1))
  tr_fn <- jit_trace(fn, input)

  tmp <- tempfile("tst", fileext = "pt")
  jit_save(tr_fn, tmp)
}
```

jit_save_for_mobile

Saves a script_function or script_module in bytecode form, to be loaded on a mobile device

**Description**

Saves a script_function or script_module in bytecode form, to be loaded on a mobile device

**Usage**

```r
jit_save_for_mobile(obj, path, ...)
```
**jit_scalar**

**Arguments**

- **obj**: An `script_function` or `script_module` to save.
- **path**: The path to save the serialized function.
- **...**: currently unused

**Examples**

```r
if (torch_is_installed()) {
  fn <- function(x) {
    torch_relu(x)
  }

  input <- torch_tensor(c(-1, 0, 1))
  tr_fn <- jit_trace(fn, input)

  tmp <- tempfile("tst", fileext = "pt")
  jit_save_for_mobile(tr_fn, tmp)
}
```

**jit_scalar**

`jit_scalar` adds the 'jit_scalar' class to the input.

**Description**

Allows disambiguating length 1 vectors from scalars when passing them to the jit.

**Usage**

`jit_scalar(x)`

**Arguments**

- **x**: A length 1 R vector.

**jit_trace**

Trace a function and return an executable `script_function`.

**Description**

Using `jit_trace`, you can turn an existing R function into a TorchScript `script_function`. You must provide example inputs, and we run the function, recording the operations performed on all the tensors.

**Usage**

`jit_trace(func, ..., strict = TRUE)`
Arguments

func  An R function that will be run with example_inputs. func arguments and return values must be tensors or (possibly nested) lists that contain tensors. Can also be a nn_module(), in such case jit_trace_module() is used to trace that module.

elements inputs that will be passed to the function while tracing. The resulting trace can be run with inputs of different types and shapes assuming the traced operations support those types and shapes. example_inputs may also be a single Tensor in which case it is automatically wrapped in a list. Note that ... can not be named, and the order is respected.

strict run the tracer in a strict mode or not (default: TRUE). Only turn this off when you want the tracer to record your mutable container types (currently list/dict) and you are sure that the container you are using in your problem is a constant structure and does not get used as control flow (if, for) conditions.

Details

The resulting recording of a standalone function produces a script_function. In the future we will also support tracing nn_modules.

Value

An script_function if func is a function and script_module if func is a nn_module().

Warning

Tracing only correctly records functions and modules which are not data dependent (e.g., do not have conditionals on data in tensors) and do not have any untracked external dependencies (e.g., perform input/output or access global variables). Tracing only records operations done when the given function is run on the given tensors. Therefore, the returned script_function will always run the same traced graph on any input. This has some important implications when your module is expected to run different sets of operations, depending on the input and/or the module state. For example,

- Tracing will not record any control-flow like if-statements or loops. When this control-flow is constant across your module, this is fine and it often inlines the control-flow decisions. But sometimes the control-flow is actually part of the model itself. For instance, a recurrent network is a loop over the (possibly dynamic) length of an input sequence.
- In the returned script_function, operations that have different behaviors in training and eval modes will always behave as if it is in the mode it was in during tracing, no matter which mode the script_function is in.

In cases like these, tracing would not be appropriate and scripting is a better choice. If you trace such models, you may silently get incorrect results on subsequent invocations of the model. The tracer will try to emit warnings when doing something that may cause an incorrect trace to be produced.

Note

Scripting is not yet supported in R.
Examples

```r
if (torch_is_installed()) {
  fn <- function(x) {
    torch_relu(x)
  }
  input <- torch_tensor(c(-1, 0, 1))
  tr_fn <- jit_trace(fn, input)
  tr_fn(input)
}
```

---

**jit_trace_module**  
*Trace a module*

**Description**

Trace a module and return an executable ScriptModule that will be optimized using just-in-time compilation. When a module is passed to `jit_trace()`, only the forward method is run and traced. With `jit_trace_module()`, you can specify a named list of method names to example inputs to trace (see the inputs) argument below.

**Usage**

```r
jit_trace_module(mod, ..., strict = TRUE)
```

**Arguments**

- **mod**
  A torch `nn_module()` containing methods whose names are specified in inputs. The given methods will be compiled as a part of a single ScriptModule.

- **...**
  A named list containing sample inputs indexed by method names in mod. The inputs will be passed to methods whose names correspond to inputs keys while tracing. list('forward'=example_forward_input, 'method2'=example_method2_input).

- **strict**
  run the tracer in a strict mode or not (default: `TRUE`). Only turn this off when you want the tracer to record your mutable container types (currently list/dict) and you are sure that the container you are using in your problem is a constant structure and does not get used as control flow (if, for) conditions.

**Details**

See `jit_trace` for more information on tracing.

**Examples**

```r
if (torch_is_installed()) {
  linear <- nn_linear(10, 1)
  tr_linear <- jit_trace_module(linear, forward = list(torch_randn(10, 10)))
  x <- torch_randn(10, 10)
  torch_allclose(linear(x), tr_linear(x))
}
```
jit_tuple

*Adds the 'jit_tuple' class to the input*

Description

Allows specifying that an output or input must be considered a jit tuple and instead of a list or dictionary when tracing.

Usage

```
jit_tuple(x)
```

Arguments

- **x**: the list object that will be converted to a tuple.

linalg_cholesky

*Computes the Cholesky decomposition of a complex Hermitian or real symmetric positive-definite matrix.*

Description

Letting \( \cdot \) be \( \cdot \) or \( \cdot \), the **Cholesky decomposition** of a complex Hermitian or real symmetric positive-definite matrix \( \cdot \) is defined as

Usage

```
linalg_cholesky(A)
```

Arguments

- **A** (Tensor): tensor of shape \((*, n, n)\) where \( * \) is zero or more batch dimensions consisting of symmetric or Hermitian positive-definite matrices.

Details

Equation not displayed. Install `katex` then re-install `torch`.

where \( \cdot \) is a lower triangular matrix and \( \cdot \) is the conjugate transpose when \( \cdot \) is complex, and the transpose when \( \cdot \) is real-valued.

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions.
See Also

- `linalg_cholesky_ex()` for a version of this operation that skips the (slow) error checking by default and instead returns the debug information. This makes it a faster way to check if a matrix is positive-definite. `linalg_eigh()` for a different decomposition of a Hermitian matrix. The eigenvalue decomposition gives more information about the matrix but it slower to compute than the Cholesky decomposition.

Other `linalg`: `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_eig()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalgqr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()`, `linalg_svd()` , `linalg_tensorsolve()`, `linalg_vector_norm()`

Examples

```r
if (torch_is_installed()) {
  a <- torch_eye(10)
  linalg_cholesky(a)
}
```

---

**linalg_cholesky_ex**  Computes the Cholesky decomposition of a complex Hermitian or real symmetric positive-definite matrix.

---

Description

This function skips the (slow) error checking and error message construction of `linalg_cholesky()`, instead directly returning the LAPACK error codes as part of a named tuple (`L`, `info`). This makes this function a faster way to check if a matrix is positive-definite, and it provides an opportunity to handle decomposition errors more gracefully or performantly than `linalg_cholesky()` does. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if A is a batch of matrices then the output has the same batch dimensions. If A is not a Hermitian positive-definite matrix, or if it’s a batch of matrices and one or more of them is not a Hermitian positive-definite matrix, then info stores a positive integer for the corresponding matrix. The positive integer indicates the order of the leading minor that is not positive-definite, and the decomposition could not be completed. info filled with zeros indicates that the decomposition was successful. If check_errors=TRUE and info contains positive integers, then a RuntimeError is thrown.

Usage

```r
linalg_cholesky_ex(A, check_errors = FALSE)
```

Arguments

- `A` (Tensor): the Hermitian \( n \times n \) matrix or the batch of such matrices of size (*, n, n) where * is one or more batch dimensions.
- `check_errors` (bool, optional): controls whether to check the content of `infos`. Default: FALSE.
Note

If A is on a CUDA device, this function may synchronize that device with the CPU.
This function is "experimental" and it may change in a future PyTorch release.

See Also

linalg_cholesky() is a NumPy compatible variant that always checks for errors.

Other linalg: linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_or(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

```r
if (torch_is_installed()) {
  A <- torch_randn(2, 2)
  out <- linalg_cholesky_ex(A)
  out
}
```

### Usage

```r
linalg_cond(A, p = NULL)
```

#### Arguments

- **A** (Tensor): tensor of shape \((*, m, n)\) where \(*\) is zero or more batch dimensions for \(p\) in \((2, -2)\), and of shape \((*, n, n)\) where every matrix is invertible for \(p\) in \(\{\text{`fro'}, \text{`nuc'}, \text{inf}, -\text{inf}, 1, -1\}\).
- **p** (int, inf, -inf, 'fro', 'nuc', optional): the type of the matrix norm to use in the computations (see above). Default: NULL

### Description

Letting \(.\) be \(.\) or \(.\), the **condition number** \(.\) of a matrix \(.\) is defined as

- **linalg_cond**

  **Computes the condition number of a matrix with respect to a matrix norm.**

- **Arguments**
  - **A** (Tensor): tensor of shape \((*, m, n)\) where \(*\) is zero or more batch dimensions for \(p\) in \((2, -2)\), and of shape \((*, n, n)\) where every matrix is invertible for \(p\) in \(\{\text{`fro'}, \text{`nuc'}, \text{inf}, -\text{inf}, 1, -1\}\).
  - **p** (int, inf, -inf, 'fro', 'nuc', optional): the type of the matrix norm to use in the computations (see above). Default: NULL
Details

Equation not displayed. Install 'katex' then re-install 'torch'.

The condition number of $A$ measures the numerical stability of the linear system $AX = B$ with respect to a matrix norm.

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if $A$ is a batch of matrices then the output has the same batch dimensions.

$p$ defines the matrix norm that is computed. See the table in 'Details' to find the supported norms.

For $p$ is one of ('fro', 'nuc', inf, -inf, 1, -1), this function uses $\text{linalg\_norm()}$ and $\text{linalg\_inv()}$.

As such, in this case, the matrix (or every matrix in the batch) $A$ has to be square and invertible.

For $p$ in $(2, -2)$, this function can be computed in terms of the singular values.

Equation not displayed. Install 'katex' then re-install 'torch'.

In these cases, it is computed using $\text{linalg\_svd()}$. For these norms, the matrix (or every matrix in the batch) $A$ may have any shape.

$p$ matrix norm
NULL 2-norm (largest singular value)
'fro' Frobenius norm
'nuc' nuclear norm
Inf max(sum(abs(x), dim=2))
-Inf min(sum(abs(x), dim=2))
1 max(sum(abs(x), dim=1))
-1 min(sum(abs(x), dim=1))
2 largest singular value
-2 smallest singular value

Value

A real-valued tensor, even when $A$ is complex.

Note

When inputs are on a CUDA device, this function synchronizes that device with the CPU if $p$ is one of ('fro', 'nuc', inf, -inf, 1, -1).

Examples

```r
if (torch_is_installed()) {
  a <- torch_tensor(rbind(c(1., 0, -1), c(0, 1, 0), c(1, 0, 1)))
  linalg_cond(a)
  linalg_cond(a, "fro")
}
```
linalg_det

Computes the determinant of a square matrix.

Description

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions.

Usage

linalg_det(A)

Arguments

\( A \) (Tensor): tensor of shape \((*, n, n)\) where \(*\) is zero or more batch dimensions.

See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg qr(), linalg slogdet(), linalg solve(), linalg svdvals(), linalg svd(), linalg tensorinv(), linalg tensorsolve(), linalg vector norm()

Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(3, 3)
  linalg_det(a)

  a <- torch_randn(3, 3, 3)
  linalg_det(a)
}
```

linalg_eig

Computes the eigenvalue decomposition of a square matrix if it exists.

Description

Letting \( \cdot \) be \( . \) or \( . \), the eigenvalue decomposition of a square matrix \( \cdot \) (if it exists) is defined as

Usage

linalg_eig(A)
Arguments

\(A\) (Tensor): tensor of shape \((*, n, n)\) where \(*\) is zero or more batch dimensions consisting of diagonalizable matrices.

Details

Equation not displayed. Install 'katex' then re-install 'torch'.

This decomposition exists if and only if \(A\) is diagonalizable. This is the case when all its eigenvalues are different. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \(A\) is a batch of matrices then the output has the same batch dimensions.

Value

A list \((\text{eigenvalues}, \text{eigenvectors})\) which corresponds to \(A\) and \(A\) above. Eigenvalues and eigenvectors will always be complex-valued, even when \(A\) is real. The eigenvectors will be given by the columns of eigenvectors.

Warning

- This function assumes that \(A\) is diagonalizable (for example, when all the eigenvalues are different). If it is not diagonalizable, the returned eigenvalues will be correct but .
- The eigenvectors of a matrix are not unique, nor are they continuous with respect to \(A\). Due to this lack of uniqueness, different hardware and software may compute different eigenvectors. This non-uniqueness is caused by the fact that multiplying an eigenvector by a non-zero number produces another set of valid eigenvectors of the matrix. In this implementation, the returned eigenvectors are normalized to have norm 1 and largest real component.
- Gradients computed using \(V\) will only be finite when \(A\) does not have repeated eigenvalues. Furthermore, if the distance between any two eigenvalues is close to zero, the gradient will be numerically unstable, as it depends on the eigenvalues through the computation of .

Note

The eigenvalues and eigenvectors of a real matrix may be complex.

See Also

- `linalg_eigvals()` computes only the eigenvalues. Unlike `linalg_eig()`, the gradients of `linalg_eigvals()` are always numerically stable.
- `linalg_eigh()` for a (faster) function that computes the eigenvalue decomposition for Hermitian and symmetric matrices.
- `linalg_svd()` for a function that computes another type of spectral decomposition that works on matrices of any shape.
- `linalg_qr()` for another (much faster) decomposition that works on matrices of any shape.

Other linalg: `linalg_cholesky_ex()`, `linalg_cholesky()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg_qr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()` `linalg_svd()` , `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`
Examples

```python
if (torch_is_installed()) {
    a <- torch_randn(2, 2)
    wv <- linalg_eig(a)
}
```

---

**linalg_eigh**

*Computes the eigenvalue decomposition of a complex Hermitian or real symmetric matrix.*

---

**Description**

Letting \( \cdot \) be \( \cdot \) or \( \cdot \), the eigenvalue decomposition of a complex Hermitian or real symmetric matrix \( A \) is defined as

**Usage**

```python
linalg_eigh(A, UPLO = "L")
```

**Arguments**

- **A** (Tensor): tensor of shape \( (*) \), \( n, n \) where \( * \) is zero or more batch dimensions consisting of symmetric or Hermitian matrices.
- **UPLO** ('L', 'U', optional): controls whether to use the upper or lower triangular part of \( A \) in the computations. Default: 'L'.

**Details**

Equation not displayed. Install 'katex' then re-install 'torch'.

where \( \cdot \) is the conjugate transpose when \( \cdot \) is complex, and the transpose when \( \cdot \) is real-valued. \( \cdot \) is orthogonal in the real case and unitary in the complex case.

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions.

\( A \) is assumed to be Hermitian (resp. symmetric), but this is not checked internally, instead:

- If \( \text{UPLO} = \text{'L'} \) (default), only the lower triangular part of the matrix is used in the computation.
- If \( \text{UPLO} = \text{'U'} \), only the upper triangular part of the matrix is used. The eigenvalues are returned in ascending order.

**Value**

A list (eigenvalues, eigenvectors) which corresponds to \( \cdot \) and \( \cdot \) above. eigenvalues will always be real-valued, even when \( A \) is complex.

It will also be ordered in ascending order. eigenvectors will have the same dtype as \( A \) and will contain the eigenvectors as its columns.
Warning

- The eigenvectors of a symmetric matrix are not unique, nor are they continuous with respect to \( A \). Due to this lack of uniqueness, different hardware and software may compute different eigenvectors. This non-uniqueness is caused by the fact that multiplying an eigenvector by \(-1\) in the real case or by \(i\) in the complex case produces another set of valid eigenvectors of the matrix. This non-uniqueness problem is even worse when the matrix has repeated eigenvalues. In this case, one may multiply the associated eigenvectors spanning the subspace by a rotation matrix and the resulting eigenvectors will be valid eigenvectors.

- Gradients computed using the eigenvectors tensor will only be finite when \( A \) has unique eigenvalues. Furthermore, if the distance between any two eigenvalues is close to zero, the gradient will be numerically unstable, as it depends on the eigenvalues through the computation of \( A \).

Note

The eigenvalues of real symmetric or complex Hermitian matrices are always real.

See Also

- `linalg_eigvalsh()` computes only the eigenvalues values of a Hermitian matrix. Unlike `linalg_eigh()`, the gradients of `linalg_eigvalsh()` are always numerically stable.

- `linalg_cholesky()` for a different decomposition of a Hermitian matrix. The Cholesky decomposition gives less information about the matrix but is much faster to compute than the eigenvalue decomposition.

- `linalg_eig()` for a (slower) function that computes the eigenvalue decomposition of a not necessarily Hermitian square matrix.

- `linalg_svd()` for a (slower) function that computes the more general SVD decomposition of matrices of any shape.

- `linalg_qr()` for another (much faster) decomposition that works on general matrices.

Other linalg: `linalg_cholesky_ex()`, `linalg_cholesky()`, `linalg_det()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_eig()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg qr()`, `linalg slogdet()`, `linalg solve()`, `linalg svdvals()`, `linalg svd()`, `linalg tensorinv()`, `linalg tensorsolve()`, `linalg vector_norm()`

Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(2, 2)
  linalg_eigh(a)
}
```
linalg_eigvals  Computes the eigenvalues of a square matrix.

Description

Letting \( A \) be \( \mathbb{R} \), or \( \mathbb{C} \), the **eigenvalues** of a square matrix \( A \) are defined as the roots (counted with multiplicity) of the polynomial \( p \) of degree \( n \) given by

\[
p(x) = \det(A - xI_n)
\]

Usage

linalg_eigvals(A)

Arguments

\( A \)  (Tensor): tensor of shape \((\ast, n, n)\) where \( \ast \) is zero or more batch dimensions.

Details

Equation not displayed. Install 'katex' then re-install 'torch'.

where \( I_n \) is the \( n \)-dimensional identity matrix. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions.

Note

The eigenvalues of a real matrix may be complex, as the roots of a real polynomial may be complex. The eigenvalues of a matrix are always well-defined, even when the matrix is not diagonalizable.

See Also

linalg_eig() computes the full eigenvalue decomposition.

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalgqr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(2, 2)
  w <- linalg_eigvals(a)
}
```
Description

Letting \( A \) be a complex Hermitian or real symmetric matrix, the \textbf{eigenvalues} of \( A \) are defined as the roots (counted with multiplicity) of the polynomial \( p \) of degree \( n \) given by:

\[
p(x) = |xI - A|
\]

where \( I \) is the \( n \)-dimensional identity matrix.

The eigenvalues of a real symmetric or complex Hermitian matrix are always real. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions. The eigenvalues are returned in ascending order.

A is assumed to be Hermitian (resp. symmetric), but this is not checked internally, instead:

- If \( \text{UPLO} = 'L' \) (default), only the lower triangular part of the matrix is used in the computation.
- If \( \text{UPLO} = 'U' \), only the upper triangular part of the matrix is used.

Value

A real-valued tensor containing the eigenvalues even when \( A \) is complex. The eigenvalues are returned in ascending order.

See Also

- \texttt{linalg.eigh()} computes the full eigenvalue decomposition.

Other \texttt{linalg}: \texttt{linalg.cholesky()} , \texttt{linalg.cholesky_ex()}, \texttt{linalg.cholesky()}, \texttt{linalg.det()}, \texttt{linalg.eigh()}, \texttt{linalg.eigvals()}, \texttt{linalg.eig()}, \texttt{linalg.householder_product()}, \texttt{linalg.inv()} , \texttt{linalg.inv_ex()} , \texttt{linalg.lstsq()} , \texttt{linalg.matrix_norm()}, \texttt{linalg.matrix_rank()}, \texttt{linalg_multi_dot()}, \texttt{linalg.norm()}, \texttt{linalg.pinv()}, \texttt{linalgqr()}, \texttt{linalg_slogdet()}, \texttt{linalg_solve()}, \texttt{linalg_svdvals()}, \texttt{linalg_svd()}, \texttt{linalg_tensorinv()}, \texttt{linalg_tensorsolve()}, \texttt{linalg_vector_norm()}. 
Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(2, 2)
  linalg_eigvalsh(a)
}
```

**linalg_householder_product**

*Computes the first n columns of a product of Householder matrices.*

Description

Letting \( \cdot \) be . or ., for a matrix . with columns . with . and a vector . with ., this function computes the first . columns of the matrix.

Usage

```r
linalg_householder_product(A, tau)
```

Arguments

- **A** (Tensor): tensor of shape \((*, m, n)\) where \(*\) is zero or more batch dimensions.
- **tau** (Tensor): tensor of shape \((*, k)\) where \(*\) is zero or more batch dimensions.

Details

Equation not displayed. Install `katex` then re-install `torch`.

where . is the \(m\)-dimensional identity matrix and . is the conjugate transpose when . is complex, and the transpose when . is real-valued. See [Representation of Orthogonal or Unitary Matrices](#) for further details.

Supports inputs of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if the inputs are batches of matrices then the output has the same batch dimensions.

Note

This function only uses the values strictly below the main diagonal of \(A\). The other values are ignored.

See Also

- `torch_geqrf()` can be used together with this function to form the \(Q\) from the linalg qr() decomposition.
torch.ormqr() is a related function that computes the matrix multiplication of a product of Householder matrices with another matrix. However, that function is not supported by autograd.

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalgqr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

**Examples**

```r
if (torch_is_installed()) {
  A <- torch_randn(2, 2)
  h_tau <- torch_geqrf(A)
  Q <- linalg_householder_product(h_tau[[1]], h_tau[[2]])
  torch_allclose(Q, linalg_qr(A)[[1]])
}
```

---

**linalg.inv**

Computes the inverse of a square matrix if it exists.

**Description**

Throws a runtime_error if the matrix is not invertible.

**Usage**

`linalg.inv(A)`

**Arguments**

`A` (Tensor): tensor of shape (*, n, n) where * is zero or more batch dimensions consisting of invertible matrices.

**Details**

Letting . be . or . for a matrix ., its inverse matrix . (if it exists) is defined as

Equation not displayed. Install 'katex' then re-install 'torch'. where . is the n-dimensional identity matrix.

The inverse matrix exists if and only if . is invertible. In this case, the inverse is unique. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if `A` is a batch of matrices then the output has the same batch dimensions.

Consider using `linalg.solve()` if possible for multiplying a matrix on the left by the inverse, as `linalg.solve(A, B) == A$inv() %*% B` It is always preferred to use `linalg.solve()` when possible, as it is faster and more numerically stable than computing the inverse explicitly.
See Also

- \texttt{linalg_pinv()} computes the pseudoinverse (Moore-Penrose inverse) of matrices of any shape.
- \texttt{linalg_solve()} computes $A^{-1} \times B$ with a numerically stable algorithm.

Other \texttt{linalg}: \texttt{linalg_cholesky_ex()}, \texttt{linalg_cholesky()}, \texttt{linalg_det()}, \texttt{linalg_eigh()}, \texttt{linalg_eigvalsh()}, \texttt{linalg_eigvals()}, \texttt{linalg_eig()}, \texttt{linalg_householder_product()}, \texttt{linalg_inv_ex()}, \texttt{linalg_lstsq()}, \texttt{linalg_matrix_norm()}, \texttt{linalg_matrix_power()}, \texttt{linalg_matrix_rank()}, \texttt{linalg_multi_dot()}, \texttt{linalg_norm()}, \texttt{linalg_pinv()}, \texttt{linalg_or()}, \texttt{linalg_slogdet()}, \texttt{linalg_solve()}, \texttt{linalg_svdvals()}, \texttt{linalg_svd()}, \texttt{linalg_tensorinv()}, \texttt{linalg_tensorsolve()}, \texttt{linalg_vector_norm()}

Examples

```r
if (torch_is_installed()) {
  A <- torch_randn(4, 4)
  linalg_inv(A)
}
```

\texttt{linalg_inv_ex} \hspace{1cm} \textit{Computes the inverse of a square matrix if it is invertible.}

Description

Returns a namedtuple \texttt{(inverse, info)}. \texttt{inverse} contains the result of inverting $A$ and \texttt{info} stores the LAPACK error codes. If $A$ is not an invertible matrix, or if it's a batch of matrices and one or more of them is not an invertible matrix, then \texttt{info} stores a positive integer for the corresponding matrix. The positive integer indicates the diagonal element of the LU decomposition of the input matrix that is exactly zero. \texttt{info} filled with zeros indicates that the inversion was successful. If \texttt{check_errors=TRUE} and \texttt{info} contains positive integers, then a RuntimeError is thrown. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if $A$ is a batch of matrices then the output has the same batch dimensions.

Usage

```r
linalg_inv_ex(A, check_errors = FALSE)
```

Arguments

- $A$ (Tensor): tensor of shape (*, n, n) where * is zero or more batch dimensions consisting of square matrices.
- check\_errors (bool, optional): controls whether to check the content of \texttt{info}. Default: \texttt{FALSE}.

Note

If $A$ is on a CUDA device then this function may synchronize that device with the CPU.

This function is "experimental" and it may change in a future PyTorch release.
See Also

linalg_inv() is a NumPy compatible variant that always checks for errors.

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

if (torch_is_installed()) {
  A <- torch_randn(3, 3)
  out <- linalg_inv_ex(A)
}

linalg_lstsq

Computes a solution to the least squares problem of a system of linear equations.

Description

Letting \( A \) be \( m \times n \), the least squares problem for a linear system \( Ax = b \) with \( \cdot \) is defined as

Usage

linalg_lstsq(A, B, rcond = NULL, ..., driver = NULL)

Arguments

- **A** (Tensor): lhs tensor of shape \( (*, m, n) \) where \( * \) is zero or more batch dimensions.
- **B** (Tensor): rhs tensor of shape \( (*, m, k) \) where \( * \) is zero or more batch dimensions.
- **rcond** (float, optional): used to determine the effective rank of \( A \). If \( rcond = NULL \), \( rcond \) is set to the machine precision of the dtype of \( A \) times \( \max(m, n) \). Default: NULL.
- **...** (unused).
- **driver** (str, optional): name of the LAPACK/MAGMA method to be used. If NULL, 'gelsy' is used for CPU inputs and 'gels' for CUDA inputs. Default: NULL.
Details

Equation not displayed. Install 'katex' then re-install 'torch'.

where \( \| \cdot \| \) denotes the Frobenius norm. Supports inputs of float, double, cfloat and cdouble dtypes.

Also supports batches of matrices, and if the inputs are batches of matrices then the output has the same batch dimensions. \texttt{driver} chooses the LAPACK/MAGMA function that will be used.

For CPU inputs the valid values are \texttt{'gels'}, \texttt{'gelsy'}, \texttt{'gelsd'}, \texttt{'gelss'}. For CUDA input, the only valid driver is \texttt{'gels'}, which assumes that \( A \) is full-rank.

To choose the best driver on CPU consider:

- If \( A \) is well-conditioned (its condition number is not too large), or you do not mind some precision loss.
- For a general matrix: \texttt{'gelsy'} (QR with pivoting) (default)
- If \( A \) is full-rank: \texttt{'gels'} (QR)
- If \( A \) is not well-conditioned.
- \texttt{'gelsd'} (tridiagonal reduction and SVD)
- But if you run into memory issues: \texttt{'gelss'} (full SVD).

See also the full description of these drivers

\texttt{rcond} is used to determine the effective rank of the matrices in \( A \) when \texttt{driver} is one of (\texttt{'gelsy'}, \texttt{'gelsd'}, \texttt{'gelss'}). In this case, if \( \sigma_k \) are the singular values of \( A \) in decreasing order, \( \sigma_k \) will be rounded down to zero if \( \sigma_k \leq \texttt{rcond} \). If \texttt{rcond = NULL} (default), \texttt{rcond} is set to the machine precision of the dtype of \( A \).

This function returns the solution to the problem and some extra information in a list of four tensors \((\texttt{solution}, \texttt{residuals}, \texttt{rank}, \texttt{singular\_values})\). For inputs \( A, B \) of shape \((*, m, n), (*, m, k)\) respectively, it contains

- \texttt{solution}: the least squares solution. It has shape \((*, n, k)\).
- \texttt{residuals}: the squared residuals of the solutions, that is, \( \| \cdot \| \). It has shape equal to the batch dimensions of \( A \). It is computed when \( m > n \) and every matrix in \( A \) is full-rank, otherwise, it is an empty tensor. If \( A \) is a batch of matrices and any matrix in the batch is not full rank, then an empty tensor is returned. This behavior may change in a future PyTorch release.
- \texttt{rank}: tensor of ranks of the matrices in \( A \). It has shape equal to the batch dimensions of \( A \). It is computed when \texttt{driver} is one of (\texttt{'gelsy'}, \texttt{'gelsd'}, \texttt{'gelss'}), otherwise it is an empty tensor.
- \texttt{singular\_values}: tensor of singular values of the matrices in \( A \). It has shape \((*, \min(m, n))\). It is computed when \texttt{driver} is one of (\texttt{'gelsd'}, \texttt{'gelss'}), otherwise it is an empty tensor.

Value

A list \((\texttt{solution}, \texttt{residuals}, \texttt{rank}, \texttt{singular\_values})\).

Warning

The default value of \texttt{rcond} may change in a future PyTorch release. It is therefore recommended to use a fixed value to avoid potential breaking changes.
Note

This function computes \( X = A^{\text{pinverse}} \times B \) in a faster and more numerically stable way than performing the computations separately.

See Also

Other linalg: `linalg_chol()`, `linalg_cholesky()`, `linalg_cholesky_ex()`, `linalg_eigh()`, `linalg_eig()`, `linalg_eigvals()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg_qr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()`, `linalg_svd()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

Examples

```r
if (torch_is_installed()) {
  A <- torch_tensor(rbind(c(10, 2, 3), c(3, 10, 5), c(5, 6, 12)))$unsqueeze(1)  # shape (1, 3, 3)
  B <- torch_stack(list(
    rbind(c(2, 5, 1), c(3, 2, 1), c(5, 1, 9)),
    rbind(c(4, 2, 9), c(2, 0, 3), c(2, 5, 3))
  ), dim = 1)  # shape (2, 3, 3)
  X <- linalg_lstsq(A, B)$solution  # A is broadcasted to shape (2, 3, 3)
}
```

---

**linalg_matrix_norm**

*Computes a matrix norm.*

**Description**

If A is complex valued, it computes the norm of A$abs() Support input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices: the norm will be computed over the dimensions specified by the 2-tuple dim and the other dimensions will be treated as batch dimensions. The output will have the same batch dimensions.

**Usage**

```r
linalg_matrix_norm(
  A,
  ord = "fro",
  dim = c(-2, -1),
  keepdim = FALSE,
  dtype = NULL
)
```
Arguments

A  (Tensor): tensor with two or more dimensions. By default its shape is interpreted as (*, m, n) where * is zero or more batch dimensions, but this behavior can be controlled using dim.


dim  (int, Tupleint, optional): dimensions over which to compute the vector or matrix norm. See above for the behavior when dim=NULL. Default: NULL

keepdim  (bool, optional): If set to TRUE, the reduced dimensions are retained in the result as dimensions with size one. Default: FALSE

dtype  dtype (torch_dtype, optional): If specified, the input tensor is cast to dtype before performing the operation, and the returned tensor’s type will be dtype. Default: NULL

Details

ord defines the norm that is computed. The following norms are supported:

<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL  (default)</td>
<td>Frobenius norm</td>
<td>2-norm (see below)</td>
</tr>
<tr>
<td>&quot;fro&quot;</td>
<td>Frobenius norm</td>
<td>– not supported –</td>
</tr>
<tr>
<td>&quot;nuc&quot;</td>
<td>nuclear norm</td>
<td>– not supported –</td>
</tr>
<tr>
<td>Inf</td>
<td>max(sum(abs(x), dim=2))</td>
<td>max(abs(x))</td>
</tr>
<tr>
<td>-Inf</td>
<td>min(sum(abs(x), dim=2))</td>
<td>min(abs(x))</td>
</tr>
<tr>
<td>0</td>
<td>– not supported –</td>
<td>sum(x != 0)</td>
</tr>
<tr>
<td>1</td>
<td>max(sum(abs(x), dim=1))</td>
<td>as below</td>
</tr>
<tr>
<td>-1</td>
<td>min(sum(abs(x), dim=1))</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>largest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other int or float</td>
<td>– not supported –</td>
<td>sum(abs(x)^{ord})^{(1 / ord)}</td>
</tr>
</tbody>
</table>

See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalgqr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

if (torch_is_installed()) {
  a <- torch_arange(0, 8, dtype = torch_float())$reshape(c(3, 3))
linalg_matrix_norm(a)
linalg_matrix_norm(a, ord = -1)
b <- a$expand(c(2, -1, -1))
linalg_matrix_norm(b)
linalg_matrix_norm(b, dim = c(1, 3))
linalg_matrix_power

Computes the n-th power of a square matrix for an integer n.

Description

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if A is a batch of matrices then the output has the same batch dimensions.

Usage

linalg_matrix_power(A, n)

Arguments

A (Tensor): tensor of shape (*, m, m) where * is zero or more batch dimensions.

n (int): the exponent.

Details

If n=0, it returns the identity matrix (or batch) of the same shape as A. If n is negative, it returns the inverse of each matrix (if invertible) raised to the power of abs(n).

See Also

linalg_solve() computes A\$inverse() %*% B with a numerically stable algorithm.

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

if (torch_is_installed()) {
  A <- torch_randn(3, 3)
  linalg_matrix_power(A, 0)
}
linalg_matrix_rank  Computes the numerical rank of a matrix.

Description

The matrix rank is computed as the number of singular values (or eigenvalues in absolute value when hermitian = TRUE) that are greater than the specified tol threshold.

Usage

linalg_matrix_rank(
    A,
    ...,  
    atol = NULL,
    rtol = NULL,
    tol = NULL,
    hermitian = FALSE
)

Arguments

A        (Tensor): tensor of shape (*, m, n) where * is zero or more batch dimensions.
...      Not currently used.
atol     the absolute tolerance value. When NULL it’s considered to be zero.
rtol     the relative tolerance value. See above for the value it takes when NULL.
tol      (float, Tensor, optional): the tolerance value. See above for the value it takes when NULL. Default: NULL.
hermitian (bool, optional): indicates whether A is Hermitian if complex or symmetric if real. Default: FALSE.

Details

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if A is a batch of matrices then the output has the same batch dimensions.

If hermitian = TRUE, A is assumed to be Hermitian if complex or symmetric if real, but this is not checked internally. Instead, just the lower triangular part of the matrix is used in the computations.

If tol is not specified and A is a matrix of dimensions (m, n), the tolerance is set to be

\[
\text{Equation not displayed. Install 'katex' then re-install 'torch'.}
\]

where \( \cdot \) is the largest singular value (or eigenvalue in absolute value when hermitian = TRUE), and \( \cdot \) is the epsilon value for the dtype of A (see torch_finfo()).

If A is a batch of matrices, tol is computed this way for every element of the batch.
See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

```r
if (torch_is_installed()) {
  a <- torch_eye(10)
  linalg_matrix_rank(a)
}
```

linalg_multi_dot

Efficiently multiplies two or more matrices

Description

Efficiently multiplies two or more matrices by reordering the multiplications so that the fewest arithmetic operations are performed.

Usage

```r
linalg_multi_dot(tensors)
```

Arguments

- `tensors` (Sequence[Tensor]): two or more tensors to multiply. The first and last tensors may be 1D or 2D. Every other tensor must be 2D.

Details

Supports inputs of float, double, cfloat and cdouble dtypes. This function does not support batched inputs.

Every tensor in `tensors` must be 2D, except for the first and last which may be 1D. If the first tensor is a 1D vector of shape `(n,)` it is treated as a row vector of shape `(1, n)`, similarly if the last tensor is a 1D vector of shape `(n,)` it is treated as a column vector of shape `(n, 1)`.

If the first and last tensors are matrices, the output will be a matrix. However, if either is a 1D vector, then the output will be a 1D vector.
Note

This function is implemented by chaining `torch_mm()` calls after computing the optimal matrix multiplication order.

The cost of multiplying two matrices with shapes \((a, b)\) and \((b, c)\) is \(a \times b \times c\). Given matrices \(A\), \(B\), \(C\) with shapes \((10, 100)\), \((100, 5)\), \((5, 50)\) respectively, we can calculate the cost of different multiplication orders as follows:

Equation not displayed. Install 'katex' then re-install 'torch'.

In this case, multiplying \(A\) and \(B\) first followed by \(C\) is 10 times faster.

See Also

Other linalg: `linalg_cholesky_ex()`, `linalg_cholesky()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_eig()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_norm()`, `linalg_pinv()`, `linalgqr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()`, `linalg_svd()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

Examples

```r
if (torch_is_installed()) {
  linalg_multi_dot(list(torch_tensor(c(1, 2)), torch_tensor(c(2, 3))))
}
```

---

`linalg_norm` Computes a vector or matrix norm.

Description

If \(A\) is complex valued, it computes the norm of \(A\)\$abs() Supports input of float, double, cfloat and cdouble dtypes. Whether this function computes a vector or matrix norm is determined as follows:

Usage

`linalg_norm(A, ord = NULL, dim = NULL, keepdim = FALSE, dtype = NULL)`

Arguments

- **A** (Tensor): tensor of shape \((*, n)\) or \((*, m, n)\) where * is zero or more batch dimensions
- **dim** (int, Tupleint, optional): dimensions over which to compute the vector or matrix norm. See above for the behavior when dim=NULL. Default: NULL
- **keepdim** (bool, optional): If set to TRUE, the reduced dimensions are retained in the result as dimensions with size one. Default: FALSE
dtype

dtype (torch_dtype, optional): If specified, the input tensor is cast to dtype before performing the operation, and the returned tensor’s type will be dtype. Default: NULL

Details

- If dim is an int, the vector norm will be computed.
- If dim is a 2-tuple, the matrix norm will be computed.
- If dim=NULL and ord=NULL, A will be flattened to 1D and the 2-norm of the resulting vector will be computed.
- If dim=NULL and ord!=NULL, A must be 1D or 2D.

ord defines the norm that is computed. The following norms are supported:

<table>
<thead>
<tr>
<th>ord</th>
<th>norm for matrices</th>
<th>norm for vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL (default)</td>
<td>Frobenius norm</td>
<td>2-norm (see below)</td>
</tr>
<tr>
<td>&quot;fro&quot;</td>
<td>Frobenius norm</td>
<td>– not supported –</td>
</tr>
<tr>
<td>&quot;nuc&quot;</td>
<td>nuclear norm</td>
<td>– not supported –</td>
</tr>
<tr>
<td>Inf</td>
<td>max(sum(abs(x), dim=2))</td>
<td>max(abs(x))</td>
</tr>
<tr>
<td>-Inf</td>
<td>min(sum(abs(x), dim=2))</td>
<td>min(abs(x))</td>
</tr>
<tr>
<td>0</td>
<td>– not supported –</td>
<td>sum(x != 0)</td>
</tr>
<tr>
<td>1</td>
<td>max(sum(abs(x), dim=1)) as below</td>
<td>sum(x != 0)</td>
</tr>
<tr>
<td>-1</td>
<td>min(sum(abs(x), dim=1)) as below</td>
<td>as below</td>
</tr>
<tr>
<td>2</td>
<td>largest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>-2</td>
<td>smallest singular value</td>
<td>as below</td>
</tr>
<tr>
<td>other int or float</td>
<td>– not supported –</td>
<td>sum(abs(x)^{ord})^{(1 / ord)}</td>
</tr>
</tbody>
</table>

See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

```r
if (torch_is_installed()) {
  a <- torch_arange(0, 8, dtype = torch_float()) - 4
  a
  b <- a$reshape(c(3, 3))
  b

  linalg_norm(a)
  linalg_norm(b)
}
```
linalg_pinv

*Computes the pseudoinverse (Moore-Penrose inverse) of a matrix.*

**Description**

The pseudoinverse may be defined algebraically but it is more computationally convenient to understand it through the SVD. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if $A$ is a batch of matrices then the output has the same batch dimensions.

**Usage**

$$\text{linalg

linalg_pinv}(A, \text{rcond} = \text{NULL}, \text{hermitian} = \text{FALSE}, \text{atol} = \text{NULL}, \text{rtol} = \text{NULL})$$

**Arguments**

- $A$: (Tensor): tensor of shape $(*, m, n)$ where * is zero or more batch dimensions.
- $\text{rcond}$: (float or Tensor, optional): the tolerance value to determine when is a singular value zero. If it is a torch.Tensor, its shape must be broadcastable to that of the singular values of $A$ as returned by $\text{linalg_svd()}$. Alias for $\text{rtol}$. Default: 0.
- $\text{hermitian}$: (bool, optional): indicates whether $A$ is Hermitian if complex or symmetric if real. Default: FALSE.
- $\text{atol}$: the absolute tolerance value. When NULL it’s considered to be zero.
- $\text{rtol}$: the relative tolerance value. See above for the value it takes when NULL.

**Details**

If $\text{hermitian} = \text{TRUE}$, $A$ is assumed to be Hermitian if complex or symmetric if real, but this is not checked internally. Instead, just the lower triangular part of the matrix is used in the computations. The singular values (or the norm of the eigenvalues when $\text{hermitian} = \text{TRUE}$) that are below the specified $\text{rcond}$ threshold are treated as zero and discarded in the computation.

**Note**

This function uses $\text{linalg_svd()}$ if $\text{hermitian} = \text{FALSE}$ and $\text{linalg_eigh()}$ if $\text{hermitian} = \text{TRUE}$. For CUDA inputs, this function synchronizes that device with the CPU.

Consider using $\text{linalg_lstsq()}$ if possible for multiplying a matrix on the left by the pseudoinverse, as $\text{linalg_lstsq}(A, B)\text{solution} = A\text{pinv()} \times B$

It is always preferred to use $\text{linalg_lstsq()}$ when possible, as it is faster and more numerically stable than computing the pseudoinverse explicitly.
See Also

- \texttt{linalg_inv()} computes the inverse of a square matrix.
- \texttt{linalg_lstsq()} computes \( A^\dagger \times B \) with a numerically stable algorithm.

Other \texttt{linalg}: \texttt{linalg_cholesky\_ex()}, \texttt{linalg_cholesky()}, \texttt{linalg_det()}, \texttt{linalg_eigh()}, \texttt{linalg_eigvalsh()}, \texttt{linalg_eigvals()}, \texttt{linalg_eig()}, \texttt{linalg_householder_product()}, \texttt{linalg_inv\_ex()}, \texttt{linalg_inv()}, \texttt{linalg_lstsq()}, \texttt{linalg_matrix\_norm()}, \texttt{linalg_matrix\_power()}, \texttt{linalg_matrix\_rank()}, \texttt{linalg_multi\_dot()}, \texttt{linalg_norm()}, \texttt{linalg\_qr()}, \texttt{linalg\_slogdet()}, \texttt{linalg\_solve()}, \texttt{linalg\_svdvals()}, \texttt{linalg\_svd()}, \texttt{linalg\_tensordot()}, \texttt{linalg\_tensorsolve()}, \texttt{linalg\_vector\_norm()}

Examples

```r
if (torch_is_installed()) {
  A <- torch_randn(3, 5)
  linalg_pinv(A)
}
```

---

\textbf{\texttt{linalg\_qr}} \hspace{1cm} Computes the QR decomposition of a matrix.

---

\textbf{Description}

Letting \( Q \) be \( \mathbb{R}^m \times n \) or \( \mathbb{C}^m \times n \), the full QR decomposition of a matrix \( A \) is defined as

\textbf{Usage}

```
linalg_qr(A, mode = "reduced")
```

\textbf{Arguments}

- \texttt{A} (Tensor): tensor of shape \((*, m, n)\) where * is zero or more batch dimensions.
- \texttt{mode} (str, optional): one of 'reduced', 'complete', 'r'. Controls the shape of the returned tensors. Default: 'reduced'.

\textbf{Details}

Equation not displayed. Install 'katex' then re-install 'torch'.

where \( Q \) is orthogonal in the real case and unitary in the complex case, and \( R \) is upper triangular. When \( m > n \) (tall matrix), as \( R \) is upper triangular, its last \( m - n \) rows are zero. In this case, we can drop the last \( m - n \) columns of \( Q \) to form the reduced QR decomposition:

Equation not displayed. Install 'katex' then re-install 'torch'.

The reduced QR decomposition agrees with the full QR decomposition when \( n \geq m \) (wide matrix). Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if \( A \) is a batch of matrices then the output has the same batch dimensions. The parameter \texttt{mode} chooses between the full and reduced QR decomposition.

If \( A \) has shape \((*, m, n)\), denoting \( k = \min(m, n) \)
• mode = 'reduced' (default): Returns (Q, R) of shapes (*, m, k), (*, k, n) respectively.
• mode = 'complete': Returns (Q, R) of shapes (*, m, m), (*, m, n) respectively.
• mode = 'r': Computes only the reduced R. Returns (Q, R) with Q empty and R of shape (*, k, n).

Value
  A list (Q, R).

See Also
  Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples
  if (torch_is_installed()) {
    a <- torch_tensor(rbind(c(12., -51, 4), c(6, 167, -68), c(-4, 24, -41)))
    qr <- linalg_qr(a)
    torch_mm(qr[[1]], qr[[2]])$round()
    torch_mm(qr[[1]]$t(), qr[[1]])$round()
  }

linalg_slogdet

Computes the sign and natural logarithm of the absolute value of the
determinant of a square matrix.

Description
  For complex A, it returns the angle and the natural logarithm of the modulus of the determinant, that is, a logarithmic polar decomposition of the determinant. Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if A is a batch of matrices then the output has the same batch dimensions.

Usage
  linalg_slogdet(A)

Arguments
  A (Tensor): tensor of shape (*, n, n) where * is zero or more batch dimensions.
Value

A list (sign, logabsdet). logabsdet will always be real-valued, even when A is complex. sign will have the same dtype as A.

Notes

• The determinant can be recovered as sign * exp(logabsdet).
• When a matrix has a determinant of zero, it returns (0, -Inf).

See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

if (torch_is_installed()) {
  a <- torch_randn(3, 3)
  linalg_slogdet(a)
}

linalg_solve

Computes the solution of a square system of linear equations with a unique solution.

Description

Letting . be . or ., this function computes the solution . of the linear system associated to ., which is defined as

Usage

linalg_solve(A, B)

Arguments

A  (Tensor): tensor of shape (*, n, n) where * is zero or more batch dimensions.
B  (Tensor): right-hand side tensor of shape (*, n) or (*, n, k) or (n,) or (n, k) according to the rules described above
Details

\[AX = B\]

This system of linear equations has one solution if and only if \(A\) is invertible. This function assumes that \(A\) is invertible. Supports inputs of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if the inputs are batches of matrices then the output has the same batch dimensions.

Letting \(*\) be zero or more batch dimensions,

- If \(A\) has shape \((*, n, n)\) and \(B\) has shape \((*, n)\) (a batch of vectors) or shape \((*, n, k)\) (a batch of matrices or "multiple right-hand sides"), this function returns \(X\) of shape \((*, n)\) or \((*, n, k)\) respectively.
- Otherwise, if \(A\) has shape \((*, n, n)\) and \(B\) has shape \((n,)\) or \((n, k)\), \(B\) is broadcasted to have shape \((*, n)\) or \((*, n, k)\) respectively.

This function then returns the solution of the resulting batch of systems of linear equations.

Note

This function computes \(X = A^{-1} @ B\) in a faster and more numerically stable way than performing the computations separately.

See Also

Other linalg: linalg_cholesky_ex(), linalg_cholesky(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve(), linalg_vector_norm()

Examples

```python
if (torch_is_installed()) {
    A <- torch_randn(3, 3)
    b <- torch_randn(3)
    x <- linalg_solve(A, b)
    torch_allclose(torch_matmul(A, x), b)
}
```

---

**linalg_svd**

Computes the singular value decomposition (SVD) of a matrix.

Description

Letting \(A\) be \(m \times n\), the **full SVD** of a matrix \(A\), if \(k = \min(m, n)\), is defined as
Usage

linalg_svd(A, full_matrices = TRUE)

Arguments

A (Tensor): tensor of shape (*, m, n) where * is zero or more batch dimensions.
full_matrices (bool, optional): controls whether to compute the full or reduced SVD, and consequently, the shape of the returned tensors U and V. Default: TRUE.

Details

Equation not displayed. Install 'katex' then re-install 'torch'.
where . , . is the conjugate transpose when . is complex, and the transpose when . is real-valued.
The matrices . , . (and thus . ) are orthogonal in the real case, and unitary in the complex case.
When m > n (resp. m < n) we can drop the last m - n (resp. n - m) columns of U (resp. V) to form the reduced SVD:
Equation not displayed. Install 'katex' then re-install 'torch'.
where . .
In this case, . and . also have orthonormal columns. Supports input of float, double, cfloat and cdouble dtypes.
Also supports batches of matrices, and if A is a batch of matrices then the output has the same batch dimensions.
The returned decomposition is a named tuple (U, S, V) which corresponds to . . . above.
The singular values are returned in descending order. The parameter full_matrices chooses between the full (default) and reduced SVD.

Value

A list (U, S, V) which corresponds to . . . above. S will always be real-valued, even when A is complex. It will also be ordered in descending order. U and V will have the same dtype as A. The left / right singular vectors will be given by the columns of U and the rows of V respectively.

Warnings

The returned tensors U and V are not unique, nor are they continuous with respect to A. Due to this lack of uniqueness, different hardware and software may compute different singular vectors. This non-uniqueness is caused by the fact that multiplying any pair of singular vectors . by -1 in the real case or by . in the complex case produces another two valid singular vectors of the matrix. This non-uniqueness problem is even worse when the matrix has repeated singular values. In this case, one may multiply the associated singular vectors of U and V spanning the subspace by a rotation matrix and the resulting vectors will span the same subspace.
Gradients computed using U or V will only be finite when A does not have zero as a singular value or repeated singular values. Furthermore, if the distance between any two singular values is close to zero, the gradient will be numerically unstable, as it depends on the singular values . through the computation of . . The gradient will also be numerically unstable when A has small singular values, as it also depends on the computation of . .
When full_matrices=TRUE, the gradients with respect to $U[\ldots, :, \min(m, n):]$ and $Vh[\ldots, \min(m, n):, :]$ will be ignored, as those vectors can be arbitrary bases of the corresponding subspaces.

See Also

- `linalg_svdvals()` computes only the singular values. Unlike `linalg_svd()`, the gradients of `linalg_svdvals()` are always numerically stable.
- `linalg_eig()` for a function that computes another type of spectral decomposition of a matrix. The eigendecomposition works just on on square matrices.
- `linalg_eigh()` for a (faster) function that computes the eigenvalue decomposition for Hermitian and symmetric matrices.
- `linalg_qr()` for another (much faster) decomposition that works on general matrices.

Other linalg: `linalg_cholesky_ex()`, `linalg_cholesky()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_eig()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg_qr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(5, 3)
  linalg_svd(a, full_matrices = FALSE)
}
```

---

`linalg_svdvals`  
*Computes the singular values of a matrix.*

Description

Supports input of float, double, cfloat and cdouble dtypes. Also supports batches of matrices, and if $A$ is a batch of matrices then the output has the same batch dimensions. The singular values are returned in descending order.

Usage

`linalg_svdvals(A)`

Arguments

$A$  
(Tensor): tensor of shape $(\ast, m, n)$ where $\ast$ is zero or more batch dimensions.

Value

A real-valued tensor, even when $A$ is complex.
See Also

- \texttt{linalg_svd()} computes the full singular value decomposition.

Other linalg: \texttt{linalg_cholesky_ex()}, \texttt{linalg_cholesky()}, \texttt{linalg_det()}, \texttt{linalg_eigh()}, \texttt{linalg_eigvalsh()}, \texttt{linalg_eigvals()}, \texttt{linalg_eig()}, \texttt{linalg_householder_product()}, \texttt{linalg_inv_ex()}, \texttt{linalg_inv()}, \texttt{linalg_lstsq()}, \texttt{linalg_matrix_norm()}, \texttt{linalg_matrix_power()}, \texttt{linalg_matrix_rank()}, \texttt{linalg_multi_dot()}, \texttt{linalg_norm()}, \texttt{linalg_pinv()}, \texttt{linalg_qr()}, \texttt{linalg_slogdet()}, \texttt{linalg_solve()}, \texttt{linalg_svd()}, \texttt{linalg_tensorinv()}, \texttt{linalg_tensorsolve()}, \texttt{linalg_vector_norm()}

Examples

```r
if (torch_is_installed()) {
  A <- torch_randn(5, 3)
  S <- linalg_svdvals(A)
  S
}
```

\texttt{linalg_tensorinv} \hspace{1cm} \textit{Computes the multiplicative inverse of torch_tensordot()}

Description

If \( m \) is the product of the first \( \text{ind} \) dimensions of \( A \) and \( n \) is the product of the rest of the dimensions, this function expects \( m \) and \( n \) to be equal. If this is the case, it computes a tensor \( X \) such that \texttt{tensordot(A, X, ind)} is the identity matrix in dimension \( m \).

Usage

\texttt{linalg_tensorinv(A, ind = 3L)}

Arguments

- \( A \) (Tensor): tensor to invert.
- \( \text{ind} \) (int): index at which to compute the inverse of \texttt{torch_tensordot()}. Default: 3.

Details

Supports input of float, double, cfloat and cdouble dtypes.

Note

Consider using \texttt{linalg_tensorsolve()} if possible for multiplying a tensor on the left by the tensor inverse as \( \texttt{linalg_tensorsolve(A, B)} = \texttt{torch_tensordot(linalg_tensorinv(A), B)} \).

It is always prefered to use \texttt{linalg_tensorsolve()} when possible, as it is faster and more numerically stable than computing the pseudoinverse explicitly.
See Also

- `linalg_tensorsolve()` computes `torch_tensordot(linalg_tensorinv(A), B)`.

Other linalg: `linalg_cholesky_ex()`, `linalg_cholesky()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_eigvals()`, `linalg_householder_product()`, `linalg_inv_ex()`, `linalg_inv()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_matrix_rank()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg_qr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_svdvals()`, `linalg_svd()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

Examples

```r
if (torch_is_installed()) {
  A <- torch_eye(4 * 6)$reshape(c(4, 6, 8, 3))
  Ainv <- linalg_tensorinv(A, ind = 3)
  Ainv$shape
  B <- torch_randn(4, 6)
  torch_allclose(torch_tensordot(Ainv, B), linalg_tensorsolve(A, B))

  A <- torch_randn(4, 4)
  Atensorinv <- linalg_tensorinv(A, 2)
  Ainv <- linalg_inv(A)
  torch_allclose(Atensorinv, Ainv)
}
```

linalg_tensorsolve Computes the solution \( X \) to the system \( \text{torch_tensordot}(A, X) = B \).

Description

If \( m \) is the product of the first \( B\.ndim \) dimensions of \( A \) and \( n \) is the product of the rest of the dimensions, this function expects \( m \) and \( n \) to be equal. The returned tensor \( x \) satisfies \( \text{tensordot}(A, x, \text{dims}=x\.ndim) = B \).

Usage

```
linalg_tensorsolve(A, B, dims = NULL)
```

Arguments

- \( A \) (Tensor): tensor to solve for.
- \( B \) (Tensor): the solution
- \( \text{dims} \) (Tupleint, optional): dimensions of \( A \) to be moved. If NULL, no dimensions are moved. Default: NULL.

Details

If \( \text{dims} \) is specified, \( A \) will be reshaped as \( A = \text{movedim}(A, \text{dims}, \text{seq}(\text{len}(\text{dims}) - A\.ndim + 1, 0)) \)

Supports inputs of float, double, cfloat and cdouble dtypes.
linalg_vector_norm

Computes a vector norm.

Description

If A is complex valued, it computes the norm of A.abs(). Supports input of float, double, cfloat and cdouble dtypes. This function does not necessarily treat multidimensional A as a batch of vectors, instead:

Usage

linalg_vector_norm(A, ord = 2, dim = NULL, keepdim = FALSE, dtype = NULL)

Arguments

A (Tensor): tensor, flattened by default, but this behavior can be controlled using dim.
dim (int, TupleInt, optional): dimensions over which to compute the vector or matrix norm. See above for the behavior when dim=NULL. Default: NULL
keepdim (bool, optional): If set to TRUE, the reduced dimensions are retained in the result as dimensions with size one. Default: FALSE
dtype (torch_dtype, optional): If specified, the input tensor is cast to dtype before performing the operation, and the returned tensor’s type will be dtype. Default: NULL

Details

- If dim=NULL, A will be flattened before the norm is computed.
- If dim is an int or a tuple, the norm will be computed over these dimensions and the other dimensions will be treated as batch dimensions.

This behavior is for consistency with linalg_norm().

ord defines the norm that is computed. The following norms are supported:

\[
\begin{array}{ccc}
\text{ord} & \text{norm for matrices} & \text{norm for vectors} \\
\text{NULL (default)} & \text{Frobenius norm} & 2\text{-norm (see below)} \\
"fro" & \text{Frobenius norm} & \text{not supported} \\
"nuc" & \text{nuclear norm} & \text{not supported} \\
\text{Inf} & \max(\sum(\text{abs}(x), \text{dim}=2)) & \max(\text{abs}(x)) \\
\text{-Inf} & \min(\sum(\text{abs}(x), \text{dim}=2)) & \min(\text{abs}(x)) \\
0 & \text{not supported} & \sum(x \neq 0) \\
1 & \max(\sum(\text{abs}(x), \text{dim}=1)) & \text{as below} \\
-1 & \min(\sum(\text{abs}(x), \text{dim}=1)) & \text{as below} \\
2 & \text{largest singular value} & \text{as below} \\
-2 & \text{smallest singular value} & \text{as below} \\
other \text{int or float} & \text{not supported} & \sum(\text{abs}(x)^{\text{ord}})^{(1 / \text{ord})}
\end{array}
\]

See Also

Other linalg: linalg_chol_ex(), linalg_chol(), linalg_det(), linalg_eigh(), linalg_eigvalsh(), linalg_eigvals(), linalg_eig(), linalg_householder_product(), linalg_inv_ex(), linalg_inv(), linalg_lstsq(), linalg_matrix_norm(), linalg_matrix_power(), linalg_matrix_rank(), linalg_multi_dot(), linalg_norm(), linalg_pinv(), linalg_qr(), linalg_slogdet(), linalg_solve(), linalg_svdvals(), linalg_svd(), linalg_tensorinv(), linalg_tensorsolve()

Examples

```r
if (torch_is_installed()) {
  a <- torch_arange(0, 8, dtype = torch_float()) - 4
  a
  b <- a$reshape(c(3, 3))
  b

  linalg_vector_norm(a, ord = 3.5)
  linalg_vector_norm(b, ord = 3.5)
}
```
load_state_dict  

Load a state dict file

Description
This function should only be used to load models saved in python. For it to work correctly you need to use torch.save with the flag: _use_new_zipfile_serialization=True and also remove all nn.Parameter classes from the tensors in the dict.

Usage
load_state_dict(path, ..., legacy_stream = FALSE)

Arguments
- path: to the state dict file
- ...: additional arguments that are currently not used.
- legacy_stream: if TRUE then the state dict is loaded using a a legacy way of handling streams.

Details
The above might change with development of this in pytorch’s C++ api.

Value
a named list of tensors.

local_autocast  

Autocast context manager

Description
Allow regions of your code to run in mixed precision. In these regions, ops run in an op-specific dtype chosen by autocast to improve performance while maintaining accuracy.

Usage
local_autocast(
  device_type,
  dtype = NULL,
  enabled = TRUE,
  cache_enabled = NULL,
  ...,  
  .env = parent.frame()
)

with_autocast(
  code,
  ..., device_type, dtype = NULL, enabled = TRUE, cache_enabled = NULL
)

Arguments

- **device_type**: a character string indicating whether to use 'cuda' or 'cpu' device.
- **dtype**: a torch data type indicating whether to use torch_float16() or torch_bfloat16().
- **enabled**: a logical value indicating whether autocasting should be enabled in the region. Default: TRUE
- **cache_enabled**: a logical value indicating whether the weight cache inside autocast should be enabled.
- **...**: currently unused.
- **.env**: The environment to use for scoping.
- **code**: code to be executed with no gradient recording.

Details

When entering an autocast-enabled region, Tensors may be any type. You should not call half() or bfloat16() on your model(s) or inputs when using autocasting.

autocast should only be enabled during the forward pass(es) of your network, including the loss computation(s). Backward passes under autocast are not recommended. Backward ops run in the same type that autocast used for corresponding forward ops.

Functions

- **with_autocast()**: A with context for automatic mixed precision.

See Also

- cuda_amp_grad_scaler() to perform dynamic gradient scaling.

Examples

```r
if (torch_is_installed()) {
  x <- torch_randn(5, 5, dtype = torch_float32())
  y <- torch_randn(5, 5, dtype = torch_float32())

  foo <- function(x, y) {
    local_autocast(device = "cpu")
    z <- torch_mm(x, y)
    w <- torch_mm(z, x)
  }
}
```
\begin{code}
\begin{verbatim}

\}
\}
out <- foo(x, y)
\}
\end{verbatim}
\end{code}

\section*{local_device  \quad Device contexts}

\textbf{Description}

Device contexts

\textbf{Usage}

\begin{verbatim}
local_device(device, ..., .env = parent.frame())
\end{verbatim}

\begin{verbatim}
with_device(code, ..., device)
\end{verbatim}

\textbf{Arguments}

\begin{verbatim}
device A torch device to be used by default when creating new tensors.
dev ... currently unused.
env .env The environment to use for scoping.
code code The code to be evaluated in the modified environment.
\end{verbatim}

\textbf{Functions}

- \texttt{with_device()}: Modifies the default device for the selected context.

\section*{lr_cosine_annealing  \quad Set the learning rate of each parameter group using a cosine annealing schedule}

\textbf{Description}

Set the learning rate of each parameter group using a cosine annealing schedule

\textbf{Usage}

\begin{verbatim}
lr_cosine_annealing(
    optimizer,
    T_max,
    eta_min = 0,
    last_epoch = -1,
    verbose = FALSE
)
\end{verbatim}
Arguments

- **optimizer** (Optimizer): Wrapped optimizer.
- **T_max** Maximum number of iterations
- **eta_min** Minimum learning rate. Default: 0.
- **last_epoch** The index of the last epoch
- **verbose** (bool): If TRUE, prints a message to stdout for each update. Default: FALSE.

---

**lr_lambda**

Sets the learning rate of each parameter group to the initial lr times a given function. When last_epoch=-1, sets initial lr as lr.

---

Description

Sets the learning rate of each parameter group to the initial lr times a given function. When last_epoch=-1, sets initial lr as lr.

Usage

```
lr_lambda(optimizer, lr_lambda, last_epoch = -1, verbose = FALSE)
```

Arguments

- **optimizer** (Optimizer): Wrapped optimizer.
- **lr_lambda** (function or list): A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- **last_epoch** (int): The index of last epoch. Default: -1.
- **verbose** (bool): If TRUE, prints a message to stdout for each update. Default: FALSE.

Examples

```
if (torch_is_installed()) {
  # Assuming optimizer has two groups.
  lambda1 <- function(epoch) epoch %/% 30
  lambda2 <- function(epoch) 0.95^epoch
  ## Not run:
  scheduler <- lr_lambda(optimizer, lr_lambda = list(lambda1, lambda2))
  for (epoch in 1:100) {
    train(...)
    validate(...)
    scheduler$step()
  }
  ## End(Not run)
}
```
**lr_multiplicative**

Multiply the learning rate of each parameter group by the factor given in the specified function. When last_epoch=-1, sets initial lr as lr.

**Description**

Multiply the learning rate of each parameter group by the factor given in the specified function. When last_epoch=-1, sets initial lr as lr.

**Usage**

```r
lr_multiplicative(optimizer, lr_lambda, last_epoch = -1, verbose = FALSE)
```

**Arguments**

- `optimizer` (Optimizer): Wrapped optimizer.
- `lr_lambda` (function or list): A function which computes a multiplicative factor given an integer parameter epoch, or a list of such functions, one for each group in optimizer.param_groups.
- `last_epoch` (int): The index of last epoch. Default: -1.
- `verbose` (bool): If TRUE, prints a message to stdout for each update. Default: FALSE.

**Examples**

```r
if (torch_is_installed()) {
  ## Not run:
  lmbda <- function(epoch) 0.95
  scheduler <- lr_multiplicative(optimizer, lr_lambda = lmbda)
  for (epoch in 1:100) {
    train(...) 
    validate(...) 
    scheduler$step()
  }

  ## End(Not run)
}
```

**lr_one_cycle**

Once cycle learning rate

**Description**

Sets the learning rate of each parameter group according to the 1cycle learning rate policy. The 1cycle policy anneals the learning rate from an initial learning rate to some maximum learning rate and then from that maximum learning rate to some minimum learning rate much lower than the initial learning rate.
Usage

```
lr_one_cycle(
    optimizer,
    max_lr,
    total_steps = NULL,
    epochs = NULL,
    steps_per_epoch = NULL,
    pct_start = 0.3,
    anneal_strategy = "cos",
    cycle_momentum = TRUE,
    base_momentum = 0.85,
    max_momentum = 0.95,
    div_factor = 25,
    final_div_factor = 10000,
    last_epoch = -1,
    verbose = FALSE
)
```

Arguments

- **optimizer** (Optimizer): Wrapped optimizer.
- **max_lr** (float or list): Upper learning rate boundaries in the cycle for each parameter group.
- **total_steps** (int): The total number of steps in the cycle. Note that if a value is not provided here, then it must be inferred by providing a value for epochs and steps_per_epoch. Default: NULL
- **epochs** (int): The number of epochs to train for. This is used along with steps_per_epoch in order to infer the total number of steps in the cycle if a value for total_steps is not provided. Default: NULL
- **steps_per_epoch** (int): The number of steps per epoch to train for. This is used along with epochs in order to infer the total number of steps in the cycle if a value for total_steps is not provided. Default: NULL
- **pct_start** (float): The percentage of the cycle (in number of steps) spent increasing the learning rate. Default: 0.3
- **anneal_strategy** (str): {'cos', 'linear'} Specifies the annealing strategy: "cos" for cosine annealing, "linear" for linear annealing. Default: 'cos'
- **cycle_momentum** (bool): If TRUE, momentum is cycled inversely to learning rate between 'base_momentum' and 'max_momentum'. Default: TRUE
- **base_momentum** (float or list): Lower momentum boundaries in the cycle for each parameter group. Note that momentum is cycled inversely to learning rate; at the peak of a cycle, momentum is 'base_momentum' and learning rate is 'max_lr'. Default: 0.85
- **max_momentum** (float or list): Upper momentum boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_momentum - base_momentum).
Note that momentum is cycled inversely to learning rate; at the start of a cycle, momentum is 'max_momentum' and learning rate is 'base_lr' Default: 0.95

**div_factor** *(float):* Determines the initial learning rate via initial_lr = max_lr/div_factor
Default: 25

**final_div_factor** *(float):* Determines the minimum learning rate via min_lr = initial_lr/final_div_factor
Default: 1e4

**last_epoch** *(int):* The index of the last batch. This parameter is used when resuming a training job. Since step() should be invoked after each batch instead of after each epoch, this number represents the total number of batches computed, not the total number of epochs computed. When last_epoch=-1, the schedule is started from the beginning. Default: -1

**verbose** *(bool):* If TRUE, prints a message to stdout for each update. Default: FALSE.

### Details

This policy was initially described in the paper *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.*

The 1cycle learning rate policy changes the learning rate after every batch. step should be called after a batch has been used for training. This scheduler is not chainable.

Note also that the total number of steps in the cycle can be determined in one of two ways (listed in order of precedence):

- A value for total_steps is explicitly provided.
- A number of epochs (epochs) and a number of steps per epoch (steps_per_epoch) are provided.

In this case, the number of total steps is inferred by total_steps = epochs * steps_per_epoch

You must either provide a value for total_steps or provide a value for both epochs and steps_per_epoch.

### Examples

```r
if (torch_is_installed()) {
    ## Not run:
data_loader <- dataloader(...)
optimizer <- optim_sgd(model$parameters, lr = 0.1, momentum = 0.9)
scheduler <- lr_one_cycle(optimizer,
                          max_lr = 0.01, steps_per_epoch = length(data_loader),
                          epochs = 10
)

for (i in 1:epochs) {
    coro::loop(for (batch in data_loader) {
        train_batch(...)
        scheduler$step()
    })
}

## End(Not run)
}
```
Description

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Usage

```r
lr_reduce_on_plateau(
  optimizer,
  mode = "min",
  factor = 0.1,
  patience = 10,
  threshold = 1e-04,
  threshold_mode = "rel",
  cooldown = 0,
  min_lr = 0,
  eps = 1e-08,
  verbose = FALSE
)
```

Arguments

- `optimizer` (Optimizer): Wrapped optimizer.
- `mode` (str): One of min, max. In min mode, lr will be reduced when the quantity monitored has stopped decreasing; in max mode it will be reduced when the quantity monitored has stopped increasing. Default: 'min'.
- `factor` (float): Factor by which the learning rate will be reduced. new_lr <- lr * factor. Default: 0.1.
- `patience` (int): Number of epochs with no improvement after which learning rate will be reduced. For example, if patience = 2, then we will ignore the first 2 epochs with no improvement, and will only decrease the LR after the 3rd epoch if the loss still hasn’t improved then. Default: 10.
- `threshold` (float): Threshold for measuring the new optimum, to only focus on significant changes. Default: 1e-4.
- `threshold_mode` (str): One of rel, abs. In rel mode, dynamic_threshold <- best * (1 + threshold) in 'max' mode or best * (1 - threshold) in min mode. In abs mode, dynamic_threshold <- best + threshold in max mode or best - threshold in min mode. Default: 'rel'.
- `cooldown` (int): Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
lr_scheduler

min_lr (float or list): A scalar or a list of scalars. A lower bound on the learning rate of all param groups or each group respectively. Default: 0.

eps (float): Minimal decay applied to lr. If the difference between new and old lr is smaller than eps, the update is ignored. Default: 1e-8.

verbose (bool): If TRUE, prints a message to stdout for each update. Default: FALSE.

Examples

if (torch_is_installed()) {
    ## Not run:
    optimizer <- optim_sgd(model$parameters(), lr=0.1, momentum=0.9)
    scheduler <- lr_reduce_on_plateau(optimizer, 'min')
    for (epoch in 1:10) {
        train(...)
        val_loss <- validate(...)  
        # note that step should be called after validate
        scheduler$step(val_loss)
    }
    ## End(Not run)
}

lr_scheduler

Creates learning rate schedulers

Description

Creates learning rate schedulers

Usage

lr_scheduler(
    classname = NULL,
    inherit = LRScheduler,
    ...
)

Arguments

classname optional name for the learning rate scheduler

inherit an optional learning rate scheduler to inherit from

... named list of methods. You must implement the get_lr() method that doesn’t take any argument and returns learning rates for each param_group in the optimizer.

parent_env passed to R6::R6Class().
lr_step

Step learning rate decay

Description
Decays the learning rate of each parameter group by gamma every step_size epochs. Notice that such decay can happen simultaneously with other changes to the learning rate from outside this scheduler. When last_epoch=-1, sets initial lr as lr.

Usage

lr_step(optimizer, step_size, gamma = 0.1, last_epoch = -1)

Arguments

optimizer (Optimizer): Wrapped optimizer.
step_size (int): Period of learning rate decay.
gamma (float): Multiplicative factor of learning rate decay. Default: 0.1.
last_epoch (int): The index of last epoch. Default: -1.

Examples

if (torch_is_installed()) {
## Not run:
# Assuming optimizer uses lr = 0.05 for all groups
# lr = 0.05 if epoch < 30
# lr = 0.005 if 30 <= epoch < 60
# lr = 0.0005 if 60 <= epoch < 90
# ...
  scheduler <- lr_step(optimizer, step_size = 30, gamma = 0.1)
  for (epoch in 1:100) {
    train(...)
    validate(...)
    scheduler$step()
  }

## End(Not run)
}
nnf_adaptive_avg_pool1d

*Adaptive_avg_pool1d*

---

**Description**

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

**Usage**

```
nnf_adaptive_avg_pool1d(input, output_size)
```

**Arguments**

- **input**: input tensor of shape (minibatch, in_channels, iW)
- **output_size**: the target output size (single integer)

---

nnf_adaptive_avg_pool2d

*Adaptive_avg_pool2d*

---

**Description**

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

**Usage**

```
nnf_adaptive_avg_pool2d(input, output_size)
```

**Arguments**

- **input**: input tensor (minibatch, in_channels, iH, iW)
- **output_size**: the target output size (single integer or double-integer tuple)
nnf_adaptive_avg_pool3d

*Adaptive_avg_pool3d*

---

**Description**

Applies a 3D adaptive average pooling over an input signal composed of several input planes.

**Usage**

```
nnf_adaptive_avg_pool3d(input, output_size)
```

**Arguments**

- `input`: input tensor (minibatch, in_channels, iT * iH, iW)
- `output_size`: the target output size (single integer or triple-integer tuple)

---

nnf_adaptive_max_pool1d

*Adaptive_max_pool1d*

---

**Description**

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

**Usage**

```
nnf_adaptive_max_pool1d(input, output_size, return_indices = FALSE)
```

**Arguments**

- `input`: input tensor of shape (minibatch, in_channels, iW)
- `output_size`: the target output size (single integer)
- `return_indices`: whether to return pooling indices. Default: FALSE
nnf_adaptive_max_pool2d

Adaptive_max_pool2d

Description

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

Usage

nnf_adaptive_max_pool2d(input, output_size, return_indices = FALSE)

Arguments

input | input tensor (minibatch, in_channels, iH, iW)
output_size | the target output size (single integer or double-integer tuple)
return_indices | whether to return pooling indices. Default: FALSE

nnf_adaptive_max_pool3d

Adaptive_max_pool3d

Description

Applies a 3D adaptive max pooling over an input signal composed of several input planes.

Usage

nnf_adaptive_max_pool3d(input, output_size, return_indices = FALSE)

Arguments

input | input tensor (minibatch, in_channels, iT * iH, iW)
output_size | the target output size (single integer or triple-integer tuple)
return_indices | whether to return pooling indices. Default: FALSE
nnf_affine_grid  Affine_grid

Description
Generates a 2D or 3D flow field (sampling grid), given a batch of affine matrices theta.

Usage
nnf_affine_grid(theta, size, align_corners = FALSE)

Arguments
theta  (Tensor) input batch of affine matrices with shape \((N \times 2 \times 3)\) for 2D or \((N \times 3 \times 4)\) for 3D
size  (torch.Size) the target output image size. \((N \times C \times H \times W)\) for 2D or \((N \times C \times D \times H \times W)\) for 3D) Example: torch.Size((32, 3, 24, 24))
align_corners  (bool, optional) if True, consider -1 and 1 to refer to the centers of the corner pixels rather than the image corners. Refer to nnf_grid_sample() for a more complete description. A grid generated by nnf_affine_grid() should be passed to nnf_grid_sample() with the same setting for this option. Default: False

Note
This function is often used in conjunction with nnf_grid_sample() to build Spatial Transformer Networks.

nnf_alpha_dropout  Alpha_dropout

Description
Applies alpha dropout to the input.

Usage
nnf_alpha_dropout(input, p = 0.5, training = FALSE, inplace = FALSE)

Arguments
input  the input tensor
p  probability of an element to be zeroed. Default: 0.5
training  apply dropout if is TRUE. Default: TRUE
inplace  If set to TRUE, will do this operation in-place. Default: FALSE
nnf_avg_pool1d

Description

Applies a 1D average pooling over an input signal composed of several input planes.

Usage

```python
nnf_avg_pool1d(
    input,
    kernel_size,
    stride = NULL,
    padding = 0,
    ceil_mode = FALSE,
    count_include_pad = TRUE
)
```

Arguments

- **input**: input tensor of shape (minibatch, in_channels, iW)
- **kernel_size**: the size of the window. Can be a single number or a tuple (kW,).
- **stride**: the stride of the window. Can be a single number or a tuple (sW,). Default: kernel_size
- **padding**: implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0
- **ceil_mode**: when True, will use ceil instead of floor to compute the output shape. Default: FALSE
- **count_include_pad**: when True, will include the zero-padding in the averaging calculation. Default: TRUE

nnf_avg_pool2d

Description

Applies 2D average-pooling operation in $kH \times kW$ regions by step size $sH \times sW$ steps. The number of output features is equal to the number of input planes.
nnf_avg_pool3d

**Usage**

```python
nnf_avg_pool3d(
    input,
    kernel_size,
    stride = NULL,
    padding = 0,
    ceil_mode = FALSE,
    count_include_pad = TRUE,
    divisor_override = NULL
)
```

**Arguments**

- **input**: input tensor (minibatch, in_channels, iH, iW)
- **kernel_size**: size of the pooling region. Can be a single number or a tuple (kH, kW)
- **stride**: stride of the pooling operation. Can be a single number or a tuple (sH, sW). Default: kernel_size
- **padding**: implicit zero paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0
- **ceil_mode**: when True, will use ceil instead of floor in the formula to compute the output shape. Default: FALSE
- **count_include_pad**: when True, will include the zero-padding in the averaging calculation. Default: TRUE
- **divisor_override**: if specified, it will be used as divisor, otherwise size of the pooling region will be used. Default: NULL

---

**nnf_avg_pool3d**

**Avg_pool3d**

---

**Description**

Applies 3D average-pooling operation in kT * kH * kW regions by step size sT * sH * sW steps. The number of output features is equal to \[ \left\lfloor \frac{\text{input planes}}{sT \cdot sH \cdot sW} \right\rfloor \].

**Usage**

```python
nnf_avg_pool3d(
    input,
    kernel_size,
    stride = NULL,
    padding = 0,
    ceil_mode = FALSE,
    count_include_pad = TRUE,
    divisor_override = NULL
)
```
**Arguments**

- **input**: input tensor (minibatch, in_channels, iT, iH, iW)
- **kernel_size**: size of the pooling region. Can be a single number or a tuple (kT, kH, kW)
- **stride**: stride of the pooling operation. Can be a single number or a tuple (sT, sH, sW). Default: kernel_size
- **padding**: implicit zero paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0
- **ceil_mode**: when True, will use ceil instead of floor in the formula to compute the output shape
- **count_include_pad**: when True, will include the zero-padding in the averaging calculation
- **divisor_override**: NA if specified, it will be used as divisor, otherwise size of the pooling region will be used. Default: NULL

**Description**

Applies Batch Normalization for each channel across a batch of data.

**Usage**

```r
nnf_batch_norm(
  input,
  running_mean,
  running_var,
  weight = NULL,
  bias = NULL,
  training = FALSE,
  momentum = 0.1,
  eps = 1e-05
)
```

**Arguments**

- **input**: input tensor
- **running_mean**: the running_mean tensor
- **running_var**: the running_var tensor
- **weight**: the weight tensor
- **bias**: the bias tensor
- **training**: bool wether it’s training. Default: FALSE
nnf_binary_cross_entropy

momentum the value used for the running_mean and running_var computation. Can be set to None for cumulative moving average (i.e. simple average). Default: 0.1
eps a value added to the denominator for numerical stability. Default: 1e-5

nnf_bilinear Bilinear

Description
Applies a bilinear transformation to the incoming data: \( y = x_1Ax_2 + b \)

Usage

nnf_bilinear(input1, input2, weight, bias = NULL)

Arguments

input1 \((N, *, H_{in1})\) where \(H_{in1} = \text{in1_features}\) and * means any number of additional dimensions. All but the last dimension of the inputs should be the same.
input2 \((N, *, H_{in2})\) where \(H_{in2} = \text{in2_features}\)
weight \((\text{out_features}, \text{in1_features}, \text{in2_features})\)
bias \((\text{out_features})\)

Value
output \((N, *, H_{out})\) where \(H_{out} = \text{out_features}\) and all but the last dimension are the same shape as the input.

nnf_binary_cross_entropy Binary_cross_entropy

Description
Function that measures the Binary Cross Entropy between the target and the output.

Usage

nnf_binary_cross_entropy(
    input,
    target,
    weight = NULL,
    reduction = c("mean", "sum", "none")
)
nnf_binary_cross_entropy_with_logits

**Arguments**

- **input**: tensor (N,*) where ** means, any number of additional dimensions
- **target**: tensor (N,*) , same shape as the input
- **weight**: (tensor) weight for each value.
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

**Description**

Function that measures Binary Cross Entropy between target and output logits.

**Usage**

```r
nnf_binary_cross_entropy_with_logits(
  input, target,
  weight = NULL,
  reduction = c("mean", "sum", "none"),
  pos_weight = NULL
)
```

**Arguments**

- **input**: Tensor of arbitrary shape
- **target**: Tensor of the same shape as input
- **weight**: (Tensor, optional) a manual rescaling weight if provided it’s repeated to match input tensor shape.
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
- **pos_weight**: (Tensor, optional) a weight of positive examples. Must be a vector with length equal to the number of classes.
nnf_contrib_sparsemax

nnf_celu

### Description

Applies element-wise, \( CELU(x) = \max(0, x) + \min(0, \alpha \times (\exp(x \alpha) - 1)) \).

### Usage

```r
nnf_celu(input, alpha = 1, inplace = FALSE)
nnf_celu_(input, alpha = 1)
```

### Arguments

- **input**: \((N, *)\) tensor, where * means, any number of additional dimensions
- **alpha**: the alpha value for the CELU formulation. Default: 1.0
- **inplace**: can optionally do the operation in-place. Default: FALSE

nnf_contrib_sparsemax

### Description

Applies the SparseMax activation.

### Usage

```r
nnf_contrib_sparsemax(input, dim = -1)
```

### Arguments

- **input**: the input tensor
- **dim**: The dimension over which to apply the sparsemax function. (-1)

### Details

The SparseMax activation is described in 'From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification' The implementation is based on aced125/sparsemax
**nnf_conv1d**

### Description

Applies a 1D convolution over an input signal composed of several input planes.

### Usage

```r
nnf_conv1d(
  input,
  weight,
  bias = NULL,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1
)
```

### Arguments

- **input**: input tensor of shape (minibatch, in_channels, iW)
- **weight**: filters of shape (out_channels, in_channels/groups, kW)
- **bias**: optional bias of shape (out_channels). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a one-element tuple (sW,). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a one-element tuple (padW,). Default: 0
- **dilation**: the spacing between kernel elements. Can be a single number or a one-element tuple (dW,). Default: 1
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1

---

**nnf_conv2d**

### Description

Applies a 2D convolution over an input image composed of several input planes.
Usage

```r
nnf_conv3d(
  input,
  weight,
  bias = NULL,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1
)
```

Arguments

- **input**: input tensor of shape (minibatch, in_channels, iH, iW)
- **weight**: filters of shape (out_channels, in_channels/groups, kH, kW)
- **bias**: optional bias tensor of shape (out_channels). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a tuple (sH, sW). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0
- **dilation**: the spacing between kernel elements. Can be a single number or a tuple (dH, dW). Default: 1
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1

---

**nnf_conv3d**  
*Conv3d*

**Description**

Applies a 3D convolution over an input image composed of several input planes.

**Usage**

```r
nnf_conv3d(
  input,
  weight,
  bias = NULL,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1
)
```
nnf_conv_tbc

Arguments

input  
input tensor of shape (minibatch, in_channels, iT, iH, iW)

weight  
filters of shape (out_channels, in_channels/groups, kT, kH, kW)

bias  
optional bias tensor of shape (out_channels). Default: NULL

stride  
the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sW). Default: 1

padding  
implicit paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0

dilation  
the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1

groups  
split input into groups, in_channels should be divisible by the number of groups. Default: 1

nnf_conv_tbc  
Conv_tbc

Description

Applies a 1-dimensional sequence convolution over an input sequence. Input and output dimensions are (Time, Batch, Channels) - hence TBC.

Usage

nnf_conv_tbc(input, weight, bias, pad = 0)

Arguments

input  
input tensor of shape (sequence length \times batch \times in_channels)

weight  
filter of shape (kernel width \times in_channels \times out_channels)

bias  
bias of shape (out_channels)

pad  
number of timesteps to pad. Default: 0
**nnf_conv_transpose1d**  
*Conv_transpose1d*

**Description**

Applies a 1D transposed convolution operator over an input signal composed of several input planes, sometimes also called “deconvolution”.

**Usage**

```r
nnf_conv_transpose1d(
  input,  
  weight,  
  bias = NULL,  
  stride = 1,  
  padding = 0,  
  output_padding = 0,  
  groups = 1,  
  dilation = 1
)
```

**Arguments**

- **input**: input tensor of shape (minibatch, in_channels, iW)
- **weight**: filters of shape (out_channels, in_channels/groups, kW)
- **bias**: optional bias of shape (out_channels). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a one-element tuple (sW,). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a one-element tuple (padW,). Default: 0
- **output_padding**: padding applied to the output
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation**: the spacing between kernel elements. Can be a single number or a one-element tuple (dW,). Default: 1
nnf_conv_transpose2d

**Description**

Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called “deconvolution”.

**Usage**

```r
nnf_conv_transpose2d(
  input,
  weight,
  bias = NULL,
  stride = 1,
  padding = 0,
  output_padding = 0,
  groups = 1,
  dilation = 1
)
```

**Arguments**

- **input**: input tensor of shape (minibatch, in_channels, iH, iW)
- **weight**: filters of shape (out_channels, in_channels/groups, kH, kW)
- **bias**: optional bias tensor of shape (out_channels). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a tuple (sH, sW). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0
- **output_padding**: padding applied to the output
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation**: the spacing between kernel elements. Can be a single number or a tuple (dH, dW). Default: 1
nnf_conv_transpose3d  

Description

Applies a 3D transposed convolution operator over an input image composed of several input planes, sometimes also called “deconvolution”

Usage

nnf_conv_transpose3d(
    input,
    weight,
    bias = NULL,
    stride = 1,
    padding = 0,
    output_padding = 0,
    groups = 1,
    dilation = 1
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input tensor of shape (minibatch, in_channels, iT, iH, iW)</td>
</tr>
<tr>
<td>weight</td>
<td>filters of shape (out_channels, in_channels/groups, kT, kH, kW)</td>
</tr>
<tr>
<td>bias</td>
<td>optional bias tensor of shape (out_channels). Default: NULL</td>
</tr>
<tr>
<td>stride</td>
<td>the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sW). Default: 1</td>
</tr>
<tr>
<td>padding</td>
<td>implicit paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0</td>
</tr>
<tr>
<td>output_padding</td>
<td>padding applied to the output</td>
</tr>
<tr>
<td>groups</td>
<td>split input into groups, in_channels should be divisible by the number of groups. Default: 1</td>
</tr>
<tr>
<td>dilation</td>
<td>the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1</td>
</tr>
</tbody>
</table>
**nnf_cosine_embedding_loss**  

*Cosine_embedding_loss*

**Description**

Creates a criterion that measures the loss given input tensors $x_1$, $x_2$ and a Tensor label $y$ with values 1 or -1. This is used for measuring whether two inputs are similar or dissimilar, using the cosine distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

**Usage**

```r
nnf_cosine_embedding_loss(
  input1, 
  input2, 
  target, 
  margin = 0, 
  reduction = c("mean", "sum", "none")
)
```

**Arguments**

- **input1**: the input $x_1$ tensor
- **input2**: the input $x_2$ tensor
- **target**: the target tensor
- **margin**: Should be a number from -1 to 1, 0 to 0.5 is suggested. If margin is missing, the default value is 0.
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

**nnf_cosine_similarity**  

*Cosine_similarity*

**Description**

Returns cosine similarity between $x_1$ and $x_2$, computed along dim.

**Usage**

```r
nnf_cosine_similarity(x1, x2, dim = 2, eps = 1e-08)
```
Arguments

- **x1** (Tensor) First input.
- **x2** (Tensor) Second input (of size matching x1).
- **dim** (int, optional) Dimension of vectors. Default: 2
- **eps** (float, optional) Small value to avoid division by zero. Default: 1e-8

Details

\[
\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}
\]

\[
\text{nnf_cross_entropy} \quad \text{Cross_entropy}
\]

Description

This criterion combines log_softmax and nll_loss in a single function.

Usage

```r
nnf_cross_entropy(
  input,
  target,
  weight = NULL,
  ignore_index = -100,
  reduction = c("mean", "sum", "none")
)
```

Arguments

- **input** (Tensor) \((N, C)\) where \(C = \text{number of classes}\) or \((N, C, H, W)\) in case of 2D loss, or \((N, C, d_1, d_2, ..., d_K)\) where \(K \geq 1\) in the case of K-dimensional loss.
- **target** (Tensor) \((N)\) where each value is \(0 \leq \text{targets}[i] \leq C - 1\), or \((N, d_1, d_2, ..., d_K)\) where \(K \geq 1\) for K-dimensional loss.
- **weight** (Tensor, optional) a manual rescaling weight given to each class. If given, has to be a Tensor of size \(C\)
- **ignore_index** (int, optional) Specifies a target value that is ignored and does not contribute to the input gradient.
- **reduction** (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
nnf_ctc_loss

Ctc_loss

Description

The Connectionist Temporal Classification loss.

Usage

nnf_ctc_loss(
  log_probs,
  targets,
  input_lengths,
  target_lengths,
  blank = 0,
  reduction = c("mean", "sum", "none"),
  zero_infinity = FALSE
)

Arguments

log_probs (T, N, C) where C = number of characters in alphabet including blank, T = input length, and N = batch size. The logarithmized probabilities of the outputs (e.g. obtained with nnf_log_softmax).

targets (N, S) or (sum(target_lengths)). Targets cannot be blank. In the second form, the targets are assumed to be concatenated.

input_lengths (N). Lengths of the inputs (must each be ≤ T)

target_lengths (N). Lengths of the targets

blank (int, optional) Blank label. Default 0.

reduction (string, optional) – Specifies the reduction to apply to the output: 'none' l 'mean' l 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

zero_infinity (bool, optional) Whether to zero infinite losses and the associated gradients. Default: FALSE Infinite losses mainly occur when the inputs are too short to be aligned to the targets.
nnf_dropout

**Description**

During training, randomly zeroes some of the elements of the input tensor with probability $p$ using samples from a Bernoulli distribution.

**Usage**

```python
nnf_dropout(input, p = 0.5, training = TRUE, inplace = FALSE)
```

**Arguments**

- `input`: the input tensor
- `p`: probability of an element to be zeroed. Default: 0.5
- `training`: apply dropout if is `TRUE`. Default: `TRUE`
- `inplace`: If set to `TRUE`, will do this operation in-place. Default: `FALSE`

nnf_dropout2d

**Description**

Randomly zero out entire channels (a channel is a 2D feature map, e.g., the $j$-th channel of the $i$-th sample in the batched input is a 2D tensor $input[i,j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability $p$ using samples from a Bernoulli distribution.

**Usage**

```python
nnf_dropout2d(input, p = 0.5, training = TRUE, inplace = FALSE)
```

**Arguments**

- `input`: the input tensor
- `p`: probability of a channel to be zeroed. Default: 0.5
- `training`: apply dropout if is `TRUE`. Default: `TRUE`
- `inplace`: If set to `TRUE`, will do this operation in-place. Default: `FALSE`
**nnf_dropout3d**

* **Description**
Randomly zero out entire channels (a channel is a 3D feature map, e.g., the $j$-th channel of the $i$-th sample in the batched input is a 3D tensor $input[i,j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability $p$ using samples from a Bernoulli distribution.

* **Usage**
```
nnf_dropout3d(input, p = 0.5, training = TRUE, inplace = FALSE)
```

* **Arguments**
  - `input`: the input tensor
  - `p`: probability of a channel to be zeroed. Default: 0.5
  - `training`: apply dropout if is TRUE. Default: TRUE.
  - `inplace`: If set to TRUE, will do this operation in-place. Default: FALSE

**nnf_elu**

* **Description**
Applies element-wise, 
\[
ELU(x) = \max(0, x) + \min(0, \alpha \ast (\exp(x) - 1))
\]

* **Usage**
```
nnf_elu(input, alpha = 1, inplace = FALSE)
nnf_elu_(input, alpha = 1)
```

* **Arguments**
  - `input`: (N,*) tensor, where * means, any number of additional dimensions
  - `alpha`: the alpha value for the ELU formulation. Default: 1.0
  - `inplace`: can optionally do the operation in-place. Default: FALSE
Examples

```r
if (torch_is_installed()) {
  x <- torch_randn(2, 2)
  y <- nnf_elu(x, alpha = 1)
  nnf_elu_(x, alpha = 1)
  torch_equal(x, y)
}
```

nnf_embedding

Description

A simple lookup table that looks up embeddings in a fixed dictionary and size.

Usage

```r
nnf_embedding(
  input,
  weight,
  padding_idx = NULL,
  max_norm = NULL,
  norm_type = 2,
  scale_grad_by_freq = FALSE,
  sparse = FALSE
)
```

Arguments

- `input` (LongTensor) Tensor containing indices into the embedding matrix
- `weight` (Tensor) The embedding matrix with number of rows equal to the maximum possible index + 1, and number of columns equal to the embedding size
- `padding_idx` (int, optional) If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- `max_norm` (float, optional) If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`. Note: this will modify `weight` in-place.
- `norm_type` (float, optional) The p of the p-norm to compute for the `max_norm` option. Default 2.
- `scale_grad_by_freq` (boolean, optional) If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default FALSE.
- `sparse` (bool, optional) If TRUE, gradient w.r.t. weight will be a sparse tensor. See Notes under `nn_embedding` for more details regarding sparse gradients.
Details

This module is often used to retrieve word embeddings using indices. The input to the module is a list of indices, and the embedding matrix, and the output is the corresponding word embeddings.

Description

Computes sums, means or maxes of bags of embeddings, without instantiating the intermediate embeddings.

Usage

```r
nnf_embedding_bag(
  input,
  weight,
  offsets = NULL,
  max_norm = NULL,
  norm_type = 2,
  scale_grad_by_freq = FALSE,
  mode = "mean",
  sparse = FALSE,
  per_sample_weights = NULL,
  include_last_offset = FALSE,
  padding_idx = NULL
)
```

Arguments

- **input**: (LongTensor) Tensor containing bags of indices into the embedding matrix
- **weight**: (Tensor) The embedding matrix with number of rows equal to the maximum possible index + 1, and number of columns equal to the embedding size
- **offsets**: (LongTensor, optional) Only used when `input` is 1D. `offsets` determines the starting index position of each bag (sequence) in `input`.
- **max_norm**: (float, optional) If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`. Note: this will modify `weight` in-place.
- **norm_type**: (float, optional) The `p` in the `p`-norm to compute for the `max_norm` option. Default 2.
- **scale_grad_by_freq**: (boolean, optional) if given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `FALSE`. Note: this option is not supported when `mode="max"`. 
**nnf_fold**

<table>
<thead>
<tr>
<th>mode</th>
<th>(string, optional) &quot;sum&quot;, &quot;mean&quot; or &quot;max&quot;. Specifies the way to reduce the bag. Default: 'mean'</th>
</tr>
</thead>
<tbody>
<tr>
<td>sparse</td>
<td>(bool, optional) if TRUE, gradient w.r.t. weight will be a sparse tensor. See Notes under nn_embedding for more details regarding sparse gradients. Note: this option is not supported when mode=&quot;max&quot;.</td>
</tr>
<tr>
<td>per_sample_weights</td>
<td>(Tensor, optional) a tensor of float / double weights, or NULL to indicate all weights should be taken to be 1. If specified, per_sample_weights must have exactly the same shape as input and is treated as having the same offsets, if those are not NULL.</td>
</tr>
<tr>
<td>include_last_offset</td>
<td>(bool, optional) if TRUE, the size of offsets is equal to the number of bags + 1.</td>
</tr>
<tr>
<td>padding_idx</td>
<td>(int, optional) If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.</td>
</tr>
</tbody>
</table>

---

**Description**

Combines an array of sliding local blocks into a large containing tensor.

**Usage**

```c
nnf_fold(
    input,
    output_size,
    kernel_size,
    dilation = 1,
    padding = 0,
    stride = 1
)
```

**Arguments**

- **input**
  the input tensor
- **output_size**
  the shape of the spatial dimensions of the output (i.e., output$sizes()[-c(1,2)]
- **kernel_size**
  the size of the sliding blocks
- **dilation**
  a parameter that controls the stride of elements within the neighborhood. Default: 1
- **padding**
  implicit zero padding to be added on both sides of input. Default: 0
- **stride**
  the stride of the sliding blocks in the input spatial dimensions. Default: 1

**Warning**

Currently, only 4-D output tensors (batched image-like tensors) are supported.
Fractions_max_pool2d

**Description**

Applies 2D fractional max pooling over an input signal composed of several input planes.

**Usage**

```python
nnf_fractional_max_pool2d(
    input,
    kernel_size,
    output_size = NULL,
    output_ratio = NULL,
    return_indices = FALSE,
    random_samples = NULL
)
```

**Arguments**

- **input** the input tensor
- **kernel_size** the size of the window to take a max over. Can be a single number \(k\) (for a square kernel of \(k \times k\)) or a tuple \((kH, kW)\)
- **output_size** the target output size of the image of the form \(oH \times oW\). Can be a tuple \((oH, oW)\) or a single number \(oH\) for a square image \(oH \times oH\)
- **output_ratio** If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range \((0, 1)\)
- **return_indices** if True, will return the indices along with the outputs.
- **random_samples** optional random samples.

**Details**

Fractional MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

The max-pooling operation is applied in \(kH \times kW\) regions by a stochastic step size determined by the target output size. The number of output features is equal to the number of input planes.
**nnf_fractional_max_pool3d**

*Fractional_max_pool3d*

### Description

Applies 3D fractional max pooling over an input signal composed of several input planes.

### Usage

```python
nnf_fractional_max_pool3d(
    input,
    kernel_size,
    output_size = NULL,
    output_ratio = NULL,
    return_indices = FALSE,
    random_samples = NULL
)
```

### Arguments

- **input**: the input tensor
- **kernel_size**: the size of the window to take a max over. Can be a single number $k$ (for a square kernel of $k \times k \times k$) or a tuple $(k_T, k_H, k_W)$
- **output_size**: the target output size of the form $o_T \times o_H \times o_W$. Can be a tuple $(o_T, o_H, o_W)$ or a single number $o_H$ for a cubic output $o_T \times o_H \times o_H$
- **output_ratio**: If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range $(0, 1)$
- **return_indices**: if True, will return the indices along with the outputs.
- **random_samples**: undocumented argument.

### Details

Fractional MaxPooling is described in detail in the paper *Fractional MaxPooling* by Ben Graham

The max-pooling operation is applied in $kT \times kH \times kW$ regions by a stochastic step size determined by the target output size. The number of output features is equal to the number of input planes.
nnf_gelu

Description
Gelu

Usage
nnf_gelu(input, approximate = "none")

Arguments
- input (N,*) tensor, where * means, any number of additional dimensions
- approximate By default it’s none, and applies element-wise x*pnorm(x), if ‘tanh’, then GELU is estimated. See GELU for more info.

gelu(input) -> Tensor
Applies element-wise the function \( \text{GELU}(x) = x \times \Phi(x) \)
where \( \Phi(x) \) is the Cumulative Distribution Function for Gaussian Distribution.
See Gaussian Error Linear Units (GELUs).

nnf_glu

Description
The gated linear unit. Computes:

Usage
nnf_glu(input, dim = -1)

Arguments
- input (Tensor) input tensor
- dim (int) dimension on which to split the input. Default: -1

Details
\[
\text{GLU}(a, b) = a \otimes \sigma(b)
\]
where input is split in half along dim to form a and b, \( \sigma \) is the sigmoid function and \( \otimes \) is the element-wise product between matrices.
See Language Modeling with Gated Convolutional Networks.
**nnf_grid_sample**

**Description**

Given an input and a flow-field grid, computes the output using input values and pixel locations from grid.

**Usage**

```r
nnf_grid_sample(
  input,
  grid,
  mode = c("bilinear", "nearest"),
  padding_mode = c("zeros", "border", "reflection"),
  align_corners = FALSE
)
```

**Arguments**

- **input** (Tensor) input of shape \((N, C, H_{in}, W_{in})\) (4-D case) or \((N, C, D_{in}, H_{in}, W_{in})\) (5-D case)
- **grid** (Tensor) flow-field of shape \((N, H_{out}, W_{out}, 2)\) (4-D case) or \((N, D_{out}, H_{out}, W_{out}, 3)\) (5-D case)
- **mode** (str) interpolation mode to calculate output values 'bilinear' | 'nearest'. Default: 'bilinear'
- **padding_mode** (str) padding mode for outside grid values 'zeros' | 'border' | 'reflection'. Default: 'zeros'
- **align_corners** (bool, optional) Geometrically, we consider the pixels of the input as squares rather than points. If set to True, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to False, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. This option parallels the align_corners option in `nnf_interpolate()`, and so whichever option is used here should also be used there to resize the input image before grid sampling. Default: False

**Details**

Currently, only spatial (4-D) and volumetric (5-D) input are supported.

In the spatial (4-D) case, for input with shape \((N, C, H_{in}, W_{in})\) and grid with shape \((N, H_{out}, W_{out}, 2)\), the output will have shape \((N, C, H_{out}, W_{out})\).

For each output location \(output[n, :, h, w]\), the size-2 vector \(grid[n, h, w]\) specifies input pixel locations \(x\) and \(y\), which are used to interpolate the output value \(output[n, :, h, w]\).

In the case of 5D inputs, \(grid[n, d, h, w]\) specifies the \(x, y, z\) pixel locations for interpolating
output[n, :, d, h, w]. mode argument specifies nearest or bilinear interpolation method to sample the input pixels.

grid specifies the sampling pixel locations normalized by the input spatial dimensions. Therefore, it should have most values in the range of [-1, 1]. For example, values x = -1, y = -1 is the left-top pixel of input, and values x = 1, y = 1 is the right-bottom pixel of input.

If grid has values outside the range of [-1, 1], the corresponding outputs are handled as defined by padding_mode. Options are

- padding_mode="zeros": use 0 for out-of-bound grid locations,
- padding_mode="border": use border values for out-of-bound grid locations,
- padding_mode="reflection": use values at locations reflected by the border for out-of-bound grid locations. For location far away from the border, it will keep being reflected until becoming in bound, e.g., (normalized) pixel location \( x = -3.5 \) reflects by border -1 and becomes \( x' = 1.5 \), then reflects by border 1 and becomes \( x'' = -0.5 \).

Note

This function is often used in conjunction with nnf_affine_grid() to build Spatial Transformer Networks.

nnf_group_norm

**Group Norm**

### Description

Applies Group Normalization for last certain number of dimensions.

### Usage

```r
nnf_group_norm(input, num_groups, weight = NULL, bias = NULL, eps = 1e-05)
```

### Arguments

- **input**: the input tensor
- **num_groups**: number of groups to separate the channels into
- **weight**: the weight tensor
- **bias**: the bias tensor
- **eps**: a value added to the denominator for numerical stability. Default: 1e-5
**nnf_gumbel_softmax**  
*Gumbel_softmax*

**Description**
Samples from the Gumbel-Softmax distribution and optionally discretizes.

**Usage**
```
nnf_gumbel_softmax(logits, tau = 1, hard = FALSE, dim = -1)
```

**Arguments**
- `logits` [..., num_features] unnormalized log probabilities
- `tau` non-negative scalar temperature
- `hard` if True, the returned samples will be discretized as one-hot vectors, but will be differentiated as if it is the soft sample in autograd
- `dim` (int) A dimension along which softmax will be computed. Default: -1.

**nnf_hardshrink**  
*Hardshrink*

**Description**
Applies the hard shrinkage function element-wise

**Usage**
```
nnf_hardshrink(input, lambd = 0.5)
```

**Arguments**
- `input` (N,*) tensor, where * means, any number of additional dimensions
- `lambd` the lambda value for the Hardshrink formulation. Default: 0.5
nnf_hardsigmoid

**Description**

Applies the element-wise function $\text{Hardsigmoid}(x) = \frac{\text{ReLU}(x+3)}{6}$

**Usage**

nnf_hardsigmoid(input, inplace = FALSE)

**Arguments**

- **input**: (N,*) tensor, where * means, any number of additional dimensions
- **inplace**: NA If set to True, will do this operation in-place. Default: False

nnf_hardswish

**Description**

Applies the hardswish function, element-wise, as described in the paper: Searching for MobileNetV3.

**Usage**

nnf_hardswish(input, inplace = FALSE)

**Arguments**

- **input**: (N,*) tensor, where * means, any number of additional dimensions
- **inplace**: can optionally do the operation in-place. Default: FALSE

**Details**

$$\text{Hardswish}(x) = \begin{cases} 
0 & \text{if } x \leq -3, \\
x & \text{if } x \geq +3, \\
\frac{x \cdot (x + 3)}{6} & \text{otherwise}
\end{cases}$$
**nnf_hardtanh**  
*Hardtanh*

**Description**
Applies the HardTanh function element-wise.

**Usage**
```
nnf_hardtanh(input, min_val = -1, max_val = 1, inplace = FALSE)
nnf_hardtanh_(input, min_val = -1, max_val = 1)
```

**Arguments**
- **input**: \((N,*)\) tensor, where * means, any number of additional dimensions
- **min_val**: minimum value of the linear region range. Default: -1
- **max_val**: maximum value of the linear region range. Default: 1
- **inplace**: can optionally do the operation in-place. Default: FALSE

---

**nnf_hinge_embedding_loss**  
*Hinge_embedding_loss*

**Description**
Measures the loss given an input tensor \(xx\) and a labels tensor \(yy\) (containing 1 or -1). This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance as \(xx\), and is typically used for learning nonlinear embeddings or semi-supervised learning.

**Usage**
```
nnf_hinge_embedding_loss(input, target, margin = 1, reduction = "mean")
```

**Arguments**
- **input**: tensor \((N,*)\) where ** means, any number of additional dimensions
- **target**: tensor \((N,*)\), same shape as the input
- **margin**: Has a default value of 1.
- **reduction** (string, optional) – Specifies the reduction to apply to the output: ’none’ | ’mean’ | ’sum’. ’none’: no reduction will be applied, ’mean’: the sum of the output will be divided by the number of elements in the output, ’sum’: the output will be summed. Default: ’mean’
**nnf_instance_norm**  
*Instance_norm*

### Description

Applies Instance Normalization for each channel in each data sample in a batch.

### Usage

```r
nnf_instance_norm(
  input,
  running_mean = NULL,
  running_var = NULL,
  weight = NULL,
  bias = NULL,
  use_input_stats = TRUE,
  momentum = 0.1,
  eps = 1e-05
)
```

### Arguments

- **input**: the input tensor
- **running_mean**: the running_mean tensor
- **running_var**: the running_var tensor
- **weight**: the weight tensor
- **bias**: the bias tensor
- **use_input_stats**: whether to use input stats
- **momentum**: a double for the momentum
- **eps**: an eps double for numerical stability

---

**nnf_interpolate**  
*Interpolate*

### Description

Down/up samples the input to either the given size or the given scale_factor
Usage

```python
nnf_interpolate(
    input,
    size = NULL,
    scale_factor = NULL,
    mode = "nearest",
    align_corners = FALSE,
    recompute_scale_factor = NULL
)
```

Arguments

- **input**: (Tensor) the input tensor
- **size**: (int or Tuple[int] or Tuple[int, int] or Tuple[int, int, int]) output spatial size.
- **scale_factor**: (float or Tuple[float]) multiplier for spatial size. Has to match input size if it is a tuple.
- **mode**: (str) algorithm used for upsampling: 'nearest' | 'linear' | 'bilinear' | 'bicubic' | 'trilinear' | 'area' Default: 'nearest'
- **align_corners**: (bool, optional) Geometrically, we consider the pixels of the input and output as squares rather than points. If set to TRUE, the input and output tensors are aligned by the center points of their corner pixels, preserving the values at the corner pixels. If set to False, the input and output tensors are aligned by the corner points of their corner pixels, and the interpolation uses edge value padding for out-of-boundary values, making this operation independent of input size when scale_factor is kept the same. This only has an effect when mode is 'linear', 'bilinear', 'bicubic' or 'trilinear'. Default: False
- **recompute_scale_factor**: (bool, optional) recompute the scale_factor for use in the interpolation calculation. When scale_factor is passed as a parameter, it is used to compute the output_size. If recompute_scale_factor is “True” or not specified, a new scale_factor will be computed based on the output and input sizes for use in the interpolation computation (i.e. the computation will be identical to if the computed ‘output_size’ were passed-in explicitly). Otherwise, the passed-in ‘scale_factor’ will be used in the interpolation computation. Note that when ‘scale_factor’ is floating-point, the recomputed scale_factor may differ from the one passed in due to rounding and precision issues.

Details

The algorithm used for interpolation is determined by mode.

Currently temporal, spatial and volumetric sampling are supported, i.e. expected inputs are 3-D, 4-D or 5-D in shape.

The input dimensions are interpreted in the form: mini-batch x channels x [optional depth] x [optional height] x width.

The modes available for resizing are: nearest, linear (3D-only), bilinear, bicubic (4D-only), trilinear (5D-only), area
nnf_kl_div

**Kl_div**

**Description**

The Kullback-Leibler divergence Loss.

**Usage**

```python
nnf_kl_div(input, target, reduction = "mean")
```

**Arguments**

- `input`: tensor (N,*), where ** means, any number of additional dimensions
- `target`: tensor (N,*), same shape as the input
- `reduction`: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

nnf_l1_loss

**L1_loss**

**Description**

Function that takes the mean element-wise absolute value difference.

**Usage**

```python
nnf_l1_loss(input, target, reduction = "mean")
```

**Arguments**

- `input`: tensor (N,*), where ** means, any number of additional dimensions
- `target`: tensor (N,*), same shape as the input
- `reduction`: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
nnf_layer_norm  

**Description**

Applies Layer Normalization for last certain number of dimensions.

**Usage**

```r
nnf_layer_norm(
    input,
    normalized_shape,
    weight = NULL,
    bias = NULL,
    eps = 1e-05
)
```

**Arguments**

- `input`: the input tensor
- `normalized_shape`: input shape from an expected input of size. If a single integer is used, it is treated as a singleton list, and this module will normalize over the last dimension which is expected to be of that specific size.
- `weight`: the weight tensor
- `bias`: the bias tensor
- `eps`: a value added to the denominator for numerical stability. Default: 1e-5

nnf_leaky_relu  

**Description**

Applies element-wise, \( \text{LeakyReLU}(x) = \max(0, x) + \text{negative_slope} \times \min(0, x) \)

**Usage**

```r
nnf_leaky_relu(input, negative_slope = 0.01, inplace = FALSE)
```

**Arguments**

- `input`: (N,*) tensor, where * means, any number of additional dimensions
- `negative_slope`: Controls the angle of the negative slope. Default: 1e-2
- `inplace`: can optionally do the operation in-place. Default: FALSE
nnf_linear

**Description**

Applies a linear transformation to the incoming data: \( y = xA^T + b \).

**Usage**

```r
nnf_linear(input, weight, bias = NULL)
```

**Arguments**

- **input** \((N, *, in\_features)\) where * means any number of additional dimensions
- **weight** \((out\_features, in\_features)\) the weights tensor.
- **bias** optional tensor \((out\_features)\)

nnf_local_response_norm

**Description**

Applies local response normalization over an input signal composed of several input planes, where channels occupy the second dimension. Applies normalization across channels.

**Usage**

```r
nnf_local_response_norm(input, size, alpha = 1e-04, beta = 0.75, k = 1)
```

**Arguments**

- **input** the input tensor
- **size** amount of neighbouring channels used for normalization
- **alpha** multiplicative factor. Default: 0.0001
- **beta** exponent. Default: 0.75
- **k** additive factor. Default: 1
**nnf_logsoftmax**

Description

Applies element-wise LogSigmoid\(x_i = \log\left(\frac{1}{1 + \exp(-x_i)}\right)\)

Usage

```python
nnf_logsoftmax(input)
```

Arguments

- `input` (N,*) tensor, where * means, any number of additional dimensions

**nnf_logsigmoid**

Description

Applies a softmax followed by a logarithm.

Usage

```python
nnf_logsoftmax(input, dim = NULL, dtype = NULL)
```

Arguments

- `input` (Tensor) input
- `dim` (int) A dimension along which log_softmax will be computed.
- `dtype` (torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

Details

While mathematically equivalent to log(softmax(x)), doing these two operations separately is slower, and numerically unstable. This function uses an alternative formulation to compute the output and gradient correctly.
nnf_lp_pool1d  

**Description**

Applies a 1D power-average pooling over an input signal composed of several input planes. If the sum of all inputs to the power of $p$ is zero, the gradient is set to zero as well.

**Usage**

```
nnf_lp_pool1d(input, norm_type, kernel_size, stride = NULL, ceil_mode = FALSE)
```

**Arguments**

- **input**: the input tensor
- **norm_type**: if inf than one gets max pooling if 0 you get sum pooling (proportional to the avg pooling)
- **kernel_size**: a single int, the size of the window
- **stride**: a single int, the stride of the window. Default value is kernel_size
- **ceil_mode**: when True, will use ceil instead of floor to compute the output shape

---

nnf_lp_pool2d  

**Description**

Applies a 2D power-average pooling over an input signal composed of several input planes. If the sum of all inputs to the power of $p$ is zero, the gradient is set to zero as well.

**Usage**

```
nnf_lp_pool2d(input, norm_type, kernel_size, stride = NULL, ceil_mode = FALSE)
```

**Arguments**

- **input**: the input tensor
- **norm_type**: if inf than one gets max pooling if 0 you get sum pooling (proportional to the avg pooling)
- **kernel_size**: a single int, the size of the window
- **stride**: a single int, the stride of the window. Default value is kernel_size
- **ceil_mode**: when True, will use ceil instead of floor to compute the output shape
**nnf_margin_ranking_loss**  
*Margin_ranking_loss*

**Description**

Creates a criterion that measures the loss given inputs \( \mathbf{x}_1, \mathbf{x}_2 \), two 1D mini-batch Tensors, and a label 1D mini-batch tensor \( \mathbf{y} \) (containing 1 or -1).

**Usage**

```python
nnf_margin_ranking_loss(input1, input2, target, margin = 0, reduction = "mean")
```

**Arguments**

- **input1**: the first tensor
- **input2**: the second input tensor
- **target**: the target tensor
- **margin**: Has a default value of 0.
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

**nnf_max_pool1d**  
*Max_pool1d*

**Description**

Applies a 1D max pooling over an input signal composed of several input planes.

**Usage**

```python
nnf_max_pool1d(
    input,
    kernel_size,
    stride = NULL,
    padding = 0,
    dilation = 1,
    ceil_mode = FALSE,
    return_indices = FALSE
)
```
nff_max_pool2d

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input tensor of shape (minibatch, in_channels, iW)</td>
</tr>
<tr>
<td>kernel_size</td>
<td>the size of the window. Can be a single number or a tuple (kW,).</td>
</tr>
<tr>
<td>stride</td>
<td>the stride of the window. Can be a single number or a tuple (sW,). Default: kernel_size</td>
</tr>
<tr>
<td>padding</td>
<td>implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0</td>
</tr>
<tr>
<td>dilation</td>
<td>controls the spacing between the kernel points; also known as the à trous algorithm.</td>
</tr>
<tr>
<td>ceil_mode</td>
<td>when True, will use ceil instead of floor to compute the output shape. Default: FALSE</td>
</tr>
<tr>
<td>return_indices</td>
<td>whether to return the indices where the max occurs.</td>
</tr>
</tbody>
</table>

Description

Applies a 2D max pooling over an input signal composed of several input planes.

Usage

```python
nff_max_pool2d(
    input,
    kernel_size,
    stride = kernel_size,
    padding = 0,
    dilation = 1,
    ceil_mode = FALSE,
    return_indices = FALSE
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input tensor (minibatch, in_channels, iH, iW)</td>
</tr>
<tr>
<td>kernel_size</td>
<td>size of the pooling region. Can be a single number or a tuple (kH, kW)</td>
</tr>
<tr>
<td>stride</td>
<td>stride of the pooling operation. Can be a single number or a tuple (sH, sW). Default: kernel_size</td>
</tr>
<tr>
<td>padding</td>
<td>implicit zero paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0</td>
</tr>
<tr>
<td>dilation</td>
<td>controls the spacing between the kernel points; also known as the à trous algorithm.</td>
</tr>
<tr>
<td>ceil_mode</td>
<td>when True, will use ceil instead of floor in the formula to compute the output shape. Default: FALSE</td>
</tr>
<tr>
<td>return_indices</td>
<td>whether to return the indices where the max occurs.</td>
</tr>
</tbody>
</table>
nnf_max_pool3d  

**Description**

Applies a 3D max pooling over an input signal composed of several input planes.

**Usage**

```
nnf_max_pool3d(
    input,
    kernel_size,
    stride = NULL,
    padding = 0,
    dilation = 1,
    ceil_mode = FALSE,
    return_indices = FALSE
)
```

**Arguments**

- **input**: input tensor (minibatch, in_channels, iT, iH, iW)
- **kernel_size**: size of the pooling region. Can be a single number or a tuple (kT, kH, kW)
- **stride**: stride of the pooling operation. Can be a single number or a tuple (sT, sH, sW). Default: kernel_size
- **padding**: implicit zero paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0
- **dilation**: controls the spacing between the kernel points; also known as the à trous algorithm.
- **ceil_mode**: when True, will use ceil instead of floor in the formula to compute the output shape
- **return_indices**: whether to return the indices where the max occurs.

nnf_max_unpool1d  

**Description**

Computes a partial inverse of MaxPool1d.
Usage

```c
nnf_max_unpool1d(
    input,
    indices,
    kernel_size,
    stride = NULL,
    padding = 0,
    output_size = NULL
)
```

Arguments

- **input**: the input Tensor to invert
- **indices**: the indices given out by max pool
- **kernel_size**: Size of the max pooling window.
- **stride**: Stride of the max pooling window. It is set to kernel_size by default.
- **padding**: Padding that was added to the input
- **output_size**: the targeted output size

---

Description

Computes a partial inverse of MaxPool2d.

Usage

```c
nnf_max_unpool2d(
    input,
    indices,
    kernel_size,
    stride = NULL,
    padding = 0,
    output_size = NULL
)
```

Arguments

- **input**: the input Tensor to invert
- **indices**: the indices given out by max pool
- **kernel_size**: Size of the max pooling window.
- **stride**: Stride of the max pooling window. It is set to kernel_size by default.
- **padding**: Padding that was added to the input
- **output_size**: the targeted output size
nnf_max_unpool3d  \textit{Max\_unpool3d}

\textbf{Description}

Computes a partial inverse of MaxPool3d.

\textbf{Usage}

\begin{verbatim}
nnf_max_unpool3d(
    input,
    indices,
    kernel_size,
    stride = NULL,
    padding = 0,
    output_size = NULL
)
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
    \item \textit{input} the input Tensor to invert
    \item \textit{indices} the indices given out by max pool
    \item \textit{kernel_size} Size of the max pooling window.
    \item \textit{stride} Stride of the max pooling window. It is set to kernel_size by default.
    \item \textit{padding} Padding that was added to the input
    \item \textit{output_size} the targeted output size
\end{itemize}

nnf_mse_loss  \textit{Mse\_loss}

\textbf{Description}

Measures the element-wise mean squared error.

\textbf{Usage}

\begin{verbatim}
nnf_mse_loss(input, target, reduction = "mean")
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
    \item \textit{input} tensor (N,*), where ** means, any number of additional dimensions
    \item \textit{target} tensor (N,*), same shape as the input
    \item \textit{reduction} (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
**nnf_multilabel_margin_loss**

*Multilabel_margin_loss*

**Description**

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input x (a 2D mini-batch Tensor) and output y (which is a 2D Tensor of target class indices).

**Usage**

```plaintext
nnf_multilabel_margin_loss(input, target, reduction = "mean")
```

**Arguments**

- **input**: tensor (N,*) where ** means, any number of additional dimensions
- **target**: tensor (N,*) , same shape as the input
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: ‘none’ | ‘mean’ | ‘sum’. ’none’: no reduction will be applied, ‘mean’: the sum of the output will be divided by the number of elements in the output, ’sum’: the output will be summed. Default: ’mean’

**nnf_multilabel_soft_margin_loss**

*Multilabel_soft_margin_loss*

**Description**

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input x and target y of size (N, C).

**Usage**

```plaintext
nnf_multilabel_soft_margin_loss(input, target, weight = NULL, reduction = "mean")
```
Arguments

- **input**: tensor (N,*) where ** means, any number of additional dimensions
- **target**: tensor (N,*) , same shape as the input
- **weight**: weight tensor to apply on the loss.
- **reduction**: (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

Note

It takes a one hot encoded target vector as input.

---

**nnf_multi_head_attention_forward**  
*Multi head attention forward*

**Description**

Allows the model to jointly attend to information from different representation subspaces. See reference: Attention Is All You Need

**Usage**

```plaintext
nnf_multi_head_attention_forward(
    query,  
    key,  
    value,  
    embed_dim_to_check,  
    num_heads,  
    in_proj_weight,  
    in_proj_bias,  
    bias_k,  
    bias_v,  
    add_zero_attn,  
    dropout_p,  
    out_proj_weight,  
    out_proj_bias,  
    training = TRUE,  
    key_padding_mask = NULL,  
    need_weights = TRUE,  
    attn_mask = NULL,  
    avg_weights = TRUE,  
    use_separate_proj_weight = FALSE,  
    q_proj_weight = NULL,  
    k_proj_weight = NULL,  
```
\begin{verbatim}
  v_proj_weight = NULL,
  static_k = NULL,
  static_v = NULL,
  batch_first = FALSE
)

Arguments

query \((L, N, E)\) where \(L\) is the target sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. If batch_first is TRUE, the first two dimensions are transposed.

key \((S, N, E)\), where \(S\) is the source sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. If batch_first is TRUE, the first two dimensions are transposed.

value \((S, N, E)\) where \(S\) is the source sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. If batch_first is TRUE, the first two dimensions are transposed.

embed_dim_to_check
total dimension of the model.

num_headsparallel attention heads.
in_proj_weightinput projection weight and bias.
in_proj_biastrue currently undocumented.
bias_kbias of the key and value sequences to be added at dim=0.
bias_vcurrenly undocumented.
add_zero_attnadd a new batch of zeros to the key and value sequences at dim=1.
dropout_pprobability of an element to be zeroed.
out_proj_weightthe output projection weight and bias.
out_proj_biastrue currently undocumented.
trainingapply dropout if is TRUE.
key_padding_mask\((N, S)\) where \(N\) is the batch size, \(S\) is the source sequence length. If a ByteTensor is provided, the non-zero positions will be ignored while the position with the zero positions will be unchanged. If a BoolTensor is provided, the positions with the value of True will be ignored while the position with the value of False will be unchanged.

need_weightsoutput attn_output_weights.

attn_mask2D mask \((L, S)\) where \(L\) is the target sequence length, \(S\) is the source sequence length. 3D mask \((N * num_heads, L, S)\) where \(N\) is the batch size, \(L\) is the target sequence length, \(S\) is the source sequence length. attn_mask ensure that position i is allowed to attend the unmasked positions. If a ByteTensor is provided, the non-zero positions are not allowed to attend while the zero positions will be unchanged. If a BoolTensor is provided, positions with True is not allowed to attend while False values will be unchanged. If a FloatTensor is provided, it will be added to the attention weight.
\end{verbatim}
nnf_multi_margin_loss

**avg_weights**  Logical; whether to average attn_output_weights over the attention heads before outputting them. This doesn’t change the returned value of attn_output; it only affects the returned attention weight matrix.

**use_separate_proj_weight**  the function accepts the proj. weights for query, key, and value in different forms. If false, in_proj_weight will be used, which is a combination of q_proj_weight, k_proj_weight, v_proj_weight.

**q_proj_weight**  input projection weight and bias.

**k_proj_weight**  currently undocumented.

**v_proj_weight**  currently undocumented.

**static_k**  static key and value used for attention operators.

**static_v**  currently undocumented.

**batch_first**  Logical; whether to expect query, key, and value to have batch as their first parameter, and to return output with batch first.

---

**nnf_multi_margin_loss**  *Multi_margin_loss*

---

**Description**

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch Tensor) and output y (which is a 1D tensor of target class indices, 0 <= y <= x$size(2) - 1$).

**Usage**

```r
nnf_multi_margin_loss(
  input,
  target,
  p = 1,
  margin = 1,
  weight = NULL,
  reduction = "mean"
)
```

**Arguments**

- **input**  tensor (N,*) where ** means, any number of additional dimensions
- **target**  tensor (N,*) , same shape as the input
- **p**  Has a default value of 1. 1 and 2 are the only supported values.
- **margin**  Has a default value of 1.
- **weight**  a manual rescaling weight given to each class. If given, it has to be a Tensor of size C. Otherwise, it is treated as if having all ones.
**nnf_nll_loss**

(string, optional) – Specifies the reduction to apply to the output: 'none', 'mean', 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

---

**nnf_nll_loss**  
**Nll_loss**

**Description**

The negative log likelihood loss.

**Usage**

```python
nnf_nll_loss(
    input,
    target,
    weight = NULL,
    ignore_index = -100,
    reduction = "mean"
)
```

**Arguments**

- **input**  
  
  \((N, C)\) where \(C = \text{number of classes}\) or \((N, C, H, W)\) in case of 2D Loss, or \((N, C, d_1, d_2, \ldots, d_K)\) where \(K \geq 1\) in the case of K-dimensional loss.

- **target**  
  
  \((N)\) where each value is \(0 \leq \text{targets}[i] \leq C - 1\), or \((N, d_1, d_2, \ldots, d_K)\) where \(K \geq 1\) for K-dimensional loss.

- **weight**  
  
  (Tensor, optional) a manual rescaling weight given to each class. If given, has to be a Tensor of size \(C\)

- **ignore_index**  
  
  (int, optional) Specifies a target value that is ignored and does not contribute to the input gradient.

- **reduction**  
  
  (string, optional) – Specifies the reduction to apply to the output: 'none', 'mean', 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
**nnf_normalize**

**Normalize**

**Description**

Performs $L_p$ normalization of inputs over specified dimension.

**Usage**

```python
nnf_normalize(input, p = 2, dim = 2, eps = 1e-12, out = NULL)
```

**Arguments**

- **input**
  - input tensor of any shape
- **p**
  - (float) the exponent value in the norm formulation. Default: 2
- **dim**
  - (int) the dimension to reduce. Default: 1
- **eps**
  - (float) small value to avoid division by zero. Default: 1e-12
- **out**
  - (Tensor, optional) the output tensor. If out is used, this operation won’t be differentiable.

**Details**

For a tensor input of sizes $(n_0, ..., n_{dim}, ..., n_k)$, each $n_{dim}$-element vector $v$ along dimension $dim$ is transformed as

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}.$$  

With the default arguments it uses the Euclidean norm over vectors along dimension 1 for normalization.

---

**nnf_one_hot**

**One_hot**

**Description**

Takes LongTensor with index values of shape (*) and returns a tensor of shape (*, num_classes) that have zeros everywhere except where the index of last dimension matches the corresponding value of the input tensor, in which case it will be 1.

**Usage**

```python
nnf_one_hot(tensor, num_classes = -1)
```
Arguments

tensor  (LongTensor) class values of any shape.
num_classes  (int) Total number of classes. If set to -1, the number of classes will be inferred as one greater than the largest class value in the input tensor.

Details

One-hot on Wikipedia: https://en.wikipedia.org/wiki/One-hot

nnf_pad

Description

Pads tensor.

Usage

nnf_pad(input, pad, mode = "constant", value = NULL)

Arguments

input  (Tensor) N-dimensional tensor
pad  (tuple) m-elements tuple, where \( \frac{m}{2} \leq \text{input dimensions} \) and \( m \) is even.
mode  'constant', 'reflect', 'replicate' or 'circular'. Default: 'constant'
value  fill value for 'constant' padding. Default: 0.

Padding size

The padding size by which to pad some dimensions of input are described starting from the last dimension and moving forward. \( \left\lfloor \frac{\text{len}(\text{pad})}{2} \right\rfloor \) dimensions of input will be padded. For example, to pad only the last dimension of the input tensor, then pad has the form \((\text{padding\_left}, \text{padding\_right})\); to pad the last 2 dimensions of the input tensor, then use \((\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom})\); to pad the last 3 dimensions, use \((\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom}, \text{padding\_front}, \text{padding\_back})\).

Padding mode

See nn_constant_pad_2d, nn_reflection_pad_2d, and nn_replication_pad_2d for concrete examples on how each of the padding modes works. Constant padding is implemented for arbitrary dimensions. tensor, or the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor. Reflect padding is only implemented for padding the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor.
### nnf_pairwise_distance

**Description**

Computes the batchwise pairwise distance between vectors using the p-norm.

**Usage**

```r
nnf_pairwise_distance(x1, x2, p = 2, eps = 1e-06, keepdim = FALSE)
```

**Arguments**

- **x1**: (Tensor) First input.
- **x2**: (Tensor) Second input (of size matching x1).
- **p**: the norm degree. Default: 2
- **eps**: (float, optional) Small value to avoid division by zero. Default: 1e-8
- **keepdim**: Determines whether or not to keep the vector dimension. Default: False

### nnf_pdist

**Description**

Computes the p-norm distance between every pair of row vectors in the input. This is identical to the upper triangular portion, excluding the diagonal, of `torch.norm(input[:, None] - input, dim=2, p=p)`.

This function will be faster if the rows are contiguous.

**Usage**

```r
nnf_pdist(input, p = 2)
```

**Arguments**

- **input**: input tensor of shape \( N \times M \).
- **p**: p value for the p-norm distance to calculate between each vector pair \( \in [0, \infty) \).

**Details**

If input has shape \( N \times M \) then the output will have shape \( \frac{1}{2}N(N - 1) \).
**nnf_pixel_shuffle**  
*Pixel_shuffle*

**Description**
Rearranges elements in a tensor of shape \(( *, C \times r^2, H, W)\) to a tensor of shape \(( *, C, H \times r, W \times r)\).

**Usage**

```python
nnf_pixel_shuffle(input, upscale_factor)
```

**Arguments**
- **input** (Tensor) the input tensor
- **upscale_factor** (int) factor to increase spatial resolution by

**nnf_poisson_nll_loss**  
*Poisson_nll_loss*

**Description**
Poisson negative log likelihood loss.

**Usage**

```python
nnf_poisson_nll_loss(
    input,
    target,
    log_input = TRUE,
    full = FALSE,
    eps = 1e-08,
    reduction = "mean"
)
```

**Arguments**
- **input** tensor (N,*) where ** means, any number of additional dimensions
- **target** tensor (N,*), same shape as the input
- **log_input** if TRUE the loss is computed as \(\exp(input) - target \times input\), if FALSE then loss is \(input - target \times \log(input + eps)\). Default: TRUE.
- **full** whether to compute full loss, i.e. to add the Stirling approximation term. Default: FALSE.
- **eps** (float, optional) Small value to avoid evaluation of \(\log(0)\) when \(log\_input = FALSE\). Default: 1e-8
**nnf_prelu**

Prelu

**Description**

Applies element-wise the function \( PReLU(x) = \max(0, x) + weight \cdot \min(0, x) \) where weight is a learnable parameter.

**Usage**

nnf_prelu(input, weight)

**Arguments**

- **input** (N,*) tensor, where * means, any number of additional dimensions
- **weight** (Tensor) the learnable weights

---

**nnf_relu**

Relu

**Description**

Applies the rectified linear unit function element-wise.

**Usage**

nnf_relu(input, inplace = FALSE)

nnf_relu_(input)

**Arguments**

- **input** (N,*) tensor, where * means, any number of additional dimensions
- **inplace** can optionally do the operation in-place. Default: FALSE

**reduction**

(string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'
**nnf_relu6**  

**Description**  

Applies the element-wise function $ReLU_6(x) = \min(\max(0, x), 6)$.

**Usage**  

```r
nnf_relu6(input, inplace = FALSE)
```

**Arguments**  

- **input**  
  (N,*) tensor, where * means, any number of additional dimensions
- **inplace**  
  can optionally do the operation in-place. Default: FALSE

---

**nnf_rrelu**  

**Description**  

Randomized leaky ReLU.

**Usage**  

```r
nnf_rrelu(input, lower = 1/8, upper = 1/3, training = FALSE, inplace = FALSE)
nnf_rrelu_(input, lower = 1/8, upper = 1/3, training = FALSE)
```

**Arguments**  

- **input**  
  (N,*) tensor, where * means, any number of additional dimensions
- **lower**  
  lower bound of the uniform distribution. Default: 1/8
- **upper**  
  upper bound of the uniform distribution. Default: 1/3
- **training**  
  bool whether it’s a training pass. Default: FALSE
- **inplace**  
  can optionally do the operation in-place. Default: FALSE
### nnf_selu

**Selu**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies element-wise,</td>
</tr>
<tr>
<td>$SELU(x) = scale \cdot (max(0, x) + min(0, \alpha \cdot (exp(x) - 1)))$</td>
</tr>
<tr>
<td>, with $\alpha = 1.6732632423543772848170429916717$ and $scale = 1.0507009873554804934193349852946$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnf_selu(input, inplace = FALSE)</td>
</tr>
<tr>
<td>nnf_selu_(input)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong></td>
</tr>
<tr>
<td><strong>inplace</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (torch_is_installed()) {</td>
</tr>
<tr>
<td>x &lt;- torch_randn(2, 2)</td>
</tr>
<tr>
<td>y &lt;- nnf_selu(x)</td>
</tr>
<tr>
<td>nnf_selu_(x)</td>
</tr>
<tr>
<td>torch_equal(x, y)</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

### nnf_sigmoid

**Sigmoid**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applies element-wise $Sigmoid(x_i) = \frac{1}{1+exp(-x_i)}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnf_sigmoid(input)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong></td>
</tr>
</tbody>
</table>
nnf_silu

Applies the Sigmoid Linear Unit (SiLU) function, element-wise. See nn_silu() for more information.

Description
Applies the Sigmoid Linear Unit (SiLU) function, element-wise. See nn_silu() for more information.

Usage
nnf_silu(input, inplace = FALSE)

Arguments
input (N,*) tensor, where * means, any number of additional dimensions
inplace can optionally do the operation in-place. Default: FALSE

See Also
nn_silu().

nnf_smooth_l1_loss  Smooth_L1_loss

Description
Function that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise.

Usage
nnf_smooth_l1_loss(input, target, reduction = "mean")

Arguments
input tensor (N,*) where ** means, any number of additional dimensions
target tensor (N,*) , same shape as the input
reduction (string, optional) – Specifies the reduction to apply to the output: ‘none’ | ‘mean’ | ‘sum’. ‘none’: no reduction will be applied, ‘mean’: the sum of the output will be divided by the number of elements in the output, ‘sum’: the output will be summed. Default: ‘mean’
nnf_softmax

**Description**

Applies a softmax function.

**Usage**

```python
nnf_softmax(input, dim, dtype = NULL)
```

**Arguments**

- `input` (Tensor) input
- `dim` (int) A dimension along which softmax will be computed.
- `dtype` (torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

**Details**

Softmax is defined as:

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range [0, 1] and sum to 1.

nnf_softmin

**Description**

Applies a softmin function.

**Usage**

```python
nnf_softmin(input, dim, dtype = NULL)
```
nnf_softplus

Arguments

<table>
<thead>
<tr>
<th>input</th>
<th>(Tensor) input</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>(int) A dimension along which softmin will be computed (so every slice along dim will sum to 1).</td>
</tr>
<tr>
<td>dtype</td>
<td>(torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.</td>
</tr>
</tbody>
</table>

Details

Note that

\[
\text{Softmin}(x) = \text{Softmax}(-x)
\]

See nnf_softmax definition for mathematical formula.

nnf_softplus Softplus

Description

Applies element-wise, the function \( \text{Softplus}(x) = \frac{1}{\beta} \log(1 + \exp(\beta \times x)) \).

Usage

nnf_softplus(input, beta = 1, threshold = 20)

Arguments

<table>
<thead>
<tr>
<th>input</th>
<th>(N,*) tensor, where * means, any number of additional dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta</td>
<td>the beta value for the Softplus formulation. Default: 1</td>
</tr>
<tr>
<td>threshold</td>
<td>values above this revert to a linear function. Default: 20</td>
</tr>
</tbody>
</table>

Details

For numerical stability the implementation reverts to the linear function when \( \text{input} \times \beta > \text{threshold} \).
nnf_softshrink | Softshrink

**Description**

Applies the soft shrinkage function elementwise

**Usage**

nnf_softshrink(input, lambd = 0.5)

**Arguments**

- **input** (N,*) tensor, where * means, any number of additional dimensions
- **lambd** the lambda (must be no less than zero) value for the Softshrink formulation. Default: 0.5

nnf_softsign | Softsign

**Description**

Applies element-wise, the function $SoftSign(x) = x / (1 + |x|)$

**Usage**

nnf_softsign(input)

**Arguments**

- **input** (N,*) tensor, where * means, any number of additional dimensions
nnf_soft_margin_loss  

**Soft_margin_loss**

**Description**

Creates a criterion that optimizes a two-class classification logistic loss between input tensor \( x \) and target tensor \( y \) (containing 1 or -1).

**Usage**

```python
nnf_soft_margin_loss(input, target, reduction = "mean")
```

**Arguments**

- **input**: tensor \((N,*)\) where ** means, any number of additional dimensions
- **target**: tensor \((N,*)\), same shape as the input
- **reduction** (string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

nnf_tanhshrink  

**Tanhshrink**

**Description**

Applies element-wise, \( \text{Tanhshrink}(x) = x - \text{Tanh}(x) \)

**Usage**

```python
nnf_tanhshrink(input)
```

**Arguments**

- **input**: \((N,*)\) tensor, where * means, any number of additional dimensions
nnf_threshold

**Description**

Thresholds each element of the input Tensor.

**Usage**

```
nnf_threshold(input, threshold, value, inplace = FALSE)
nnf_threshold_(input, threshold, value)
```

**Arguments**

- **input**: (N,*) tensor, where * means, any number of additional dimensions
- **threshold**: The value to threshold at
- **value**: The value to replace with
- **inplace**: can optionally do the operation in-place. Default: FALSE

nnf_triplet_margin_loss

**Description**

Creates a criterion that measures the triplet loss given an input tensors \( x_1, x_2, x_3 \) and a margin with a value greater than 0. This is used for measuring a relative similarity between samples. A triplet is composed by \( a, p \) and \( n \) (i.e., anchor, positive examples and negative examples respectively). The shapes of all input tensors should be (N, D).

**Usage**

```
nnf_triplet_margin_loss(
    anchor,
    positive,
    negative,
    margin = 1,
    p = 2,
    eps = 1e-06,
    swap = FALSE,
    reduction = "mean"
)
```
nnf_triplet_margin_with_distance_loss

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>the anchor input tensor</td>
</tr>
<tr>
<td>positive</td>
<td>the positive input tensor</td>
</tr>
<tr>
<td>negative</td>
<td>the negative input tensor</td>
</tr>
<tr>
<td>margin</td>
<td>Default: 1.</td>
</tr>
<tr>
<td>p</td>
<td>The norm degree for pairwise distance. Default: 2.</td>
</tr>
<tr>
<td>eps</td>
<td>(float, optional) Small value to avoid division by zero.</td>
</tr>
<tr>
<td>swap</td>
<td>The distance swap is described in detail in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al. Default: FALSE.</td>
</tr>
<tr>
<td>reduction</td>
<td>(string, optional) – Specifies the reduction to apply to the output: 'none'</td>
</tr>
</tbody>
</table>

nnf_triplet_margin_with_distance_loss


nnf_triplet_margin_with_distance_loss

Triplet margin with distance loss

Description

See nn_triplet_margin_with_distance_loss()

Usage

nnf_triplet_margin_with_distance_loss(
    anchor,
    positive,
    negative,
    distance_function = NULL,
    margin = 1,
    swap = FALSE,
    reduction = "mean"
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>the anchor input tensor</td>
</tr>
<tr>
<td>positive</td>
<td>the positive input tensor</td>
</tr>
<tr>
<td>negative</td>
<td>the negative input tensor</td>
</tr>
<tr>
<td>distance_function</td>
<td>(callable, optional): A nonnegative, real-valued function that quantifies the closeness of two tensors. If not specified, nn_pairwise_distance() will be used. Default: None</td>
</tr>
</tbody>
</table>

nnf_triplet_margin_with_distance_loss


nnf_triplet_margin_with_distance_loss

Triplet margin with distance loss

Description

See nn_triplet_margin_with_distance_loss()

Usage

nnf_triplet_margin_with_distance_loss(
    anchor,
    positive,
    negative,
    distance_function = NULL,
    margin = 1,
    swap = FALSE,
    reduction = "mean"
)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor</td>
<td>the anchor input tensor</td>
</tr>
<tr>
<td>positive</td>
<td>the positive input tensor</td>
</tr>
<tr>
<td>negative</td>
<td>the negative input tensor</td>
</tr>
<tr>
<td>distance_function</td>
<td>(callable, optional): A nonnegative, real-valued function that quantifies the closeness of two tensors. If not specified, nn_pairwise_distance() will be used. Default: None</td>
</tr>
</tbody>
</table>
nnf_unfold

Unfold

Description

Extracts sliding local blocks from an batched input tensor.

Usage

nnf_unfold(input, kernel_size, dilation = 1, padding = 0, stride = 1)

Arguments

- **input**: the input tensor
- **kernel_size**: the size of the sliding blocks
- **dilation**: a parameter that controls the stride of elements within the neighborhood. Default: 1
- **padding**: implicit zero padding to be added on both sides of input. Default: 0
- **stride**: the stride of the sliding blocks in the input spatial dimensions. Default: 1

Warning

More than one element of the unfolded tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensor, please clone it first.
nn_adaptive_avg_pool1d

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

Description

The output size is H, for any input size. The number of output features is equal to the number of input planes.

Usage

nn_adaptive_avg_pool1d(output_size)

Arguments

output_size the target output size H

Examples

if (torch_is_installed()) {
  # target output size of 5
  m <- nn_adaptive_avg_pool1d(5)
  input <- torch_randn(1, 64, 8)
  output <- m(input)
}

nn_adaptive_avg_pool2d

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

Description

The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

Usage

nn_adaptive_avg_pool2d(output_size)

Arguments

output_size the target output size of the image of the form H x W. Can be a tuple (H, W) or a single H for a square image H x H. H and W can be either an int, or NULL which means the size will be the same as that of the input.
Examples

```r
if (torch_is_installed()) {
    # target output size of 5x7
    m <- nn_adaptive_avg_pool2d(c(5, 7))
    input <- torch_randn(1, 64, 8, 9)
    output <- m(input)
    # target output size of 7x7 (square)
    m <- nn_adaptive_avg_pool2d(7)
    input <- torch_randn(1, 64, 10, 9)
    output <- m(input)
}
```

---

nn_adaptive_avg_pool3d

*Applies a 3D adaptive average pooling over an input signal composed of several input planes.*

Description

The output is of size D x H x W, for any input size. The number of output features is equal to the number of input planes.

Usage

```
nn_adaptive_avg_pool3d(output_size)
```

Arguments

- `output_size`: the target output size of the form D x H x W. Can be a tuple (D, H, W) or a single number D for a cube D x D x D. D, H and W can be either a `int`, or `None` which means the size will be the same as that of the input.

Examples

```r
if (torch_is_installed()) {
    # target output size of 5x7x9
    m <- nn_adaptive_avg_pool3d(c(5, 7, 9))
    input <- torch_randn(1, 64, 8, 9, 10)
    output <- m(input)
    # target output size of 7x7x7 (cube)
    m <- nn_adaptive_avg_pool3d(7)
    input <- torch_randn(1, 64, 10, 9, 8)
    output <- m(input)
}
```
**nn_adaptive_log_softmax_with_loss**

*AdaptiveLogSoftmaxWithLoss module*

---

**Description**

Efficient softmax approximation as described in [Efficient softmax approximation for GPUs by Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou](#).

**Usage**

```python
nn_adaptive_log_softmax_with_loss(
    in_features,
    n_classes,
    cutoffs,
    div_value = 4,
    head_bias = FALSE
)
```

**Arguments**

- `in_features` (int): Number of features in the input tensor
- `n_classes` (int): Number of classes in the dataset
- `cutoffs` (Sequence): Cutoffs used to assign targets to their buckets
- `div_value` (float, optional): Value used as an exponent to compute sizes of the clusters. Default: 4.0
- `head_bias` (bool, optional): If True, adds a bias term to the 'head' of the adaptive softmax. Default: False

**Details**

Adaptive softmax is an approximate strategy for training models with large output spaces. It is most effective when the label distribution is highly imbalanced, for example in natural language modelling, where the word frequency distribution approximately follows the Zipf’s law.

Adaptive softmax partitions the labels into several clusters, according to their frequency. These clusters may contain different number of targets each.

Additionally, clusters containing less frequent labels assign lower dimensional embeddings to those labels, which speeds up the computation. For each minibatch, only clusters for which at least one target is present are evaluated.

The idea is that the clusters which are accessed frequently (like the first one, containing most frequent labels), should also be cheap to compute – that is, contain a small number of assigned labels. We highly recommend taking a look at the original paper for more details.
nn_adaptive_log_softmax_with_loss

- **cutoffs** should be an ordered Sequence of integers sorted in the increasing order. It controls number of clusters and the partitioning of targets into clusters. For example setting `cutoffs = c(10, 100, 1000)` means that first 10 targets will be assigned to the 'head' of the adaptive softmax, targets 11, 12, ..., 100 will be assigned to the first cluster, and targets 101, 102, ..., 1000 will be assigned to the second cluster, while targets 1001, 1002, ..., n_classes - 1 will be assigned to the last, third cluster.

- **div_value** is used to compute the size of each additional cluster, which is given as $\left\lfloor \frac{\text{in_features}}{\text{div_value}^{idx}} \right\rfloor$, where $idx$ is the cluster index (with clusters for less frequent words having larger indices, and indices starting from 1).

- **head_bias** if set to True, adds a bias term to the 'head' of the adaptive softmax. See paper for details. Set to False in the official implementation.

**Value**

`NamedTuple` with output and loss fields:

- **output** is a Tensor of size N containing computed target log probabilities for each example
- **loss** is a Scalar representing the computed negative log likelihood loss

**Warning**

Labels passed as inputs to this module should be sorted according to their frequency. This means that the most frequent label should be represented by the index 0, and the least frequent label should be represented by the index n_classes - 1.

**Shape**

- input: (N, in_features)
- target: (N) where each value satisfies $0 \leq \text{target}[i] \leq \text{n_classes}$
- output1: (N)
- output2: Scalar

**Note**

This module returns a `NamedTuple` with output and loss fields. See further documentation for details.

To compute log-probabilities for all classes, the `log_prob` method can be used.
nn_adaptive_max_pool1d

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

Description

The output size is H, for any input size. The number of output features is equal to the number of input planes.

Usage

nn_adaptive_max_pool1d(output_size, return_indices = FALSE)

Arguments

output_size the target output size H

return_indices if TRUE, will return the indices along with the outputs. Useful to pass to nn_max_unpool1d(). Default: FALSE

Examples

if (torch_is_installed()) {
  # target output size of 5
  m <- nn_adaptive_max_pool1d(5)
  input <- torch_randn(1, 64, 8)
  output <- m(input)
}

nn_adaptive_max_pool2d

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

Description

The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

Usage

nn_adaptive_max_pool2d(output_size, return_indices = FALSE)
nn_adaptive_max_pool3d

Applies a 3D adaptive max pooling over an input signal composed of several input planes.

Description

The output is of size D x H x W, for any input size. The number of output features is equal to the number of input planes.

Usage

nn_adaptive_max_pool3d(output_size, return_indices = FALSE)

Arguments

output_size       the target output size of the image of the form D x H x W. Can be a tuple (D, H, W) or a single D for a cube D x D x D. D, H and W can be either a int, or None which means the size will be the same as that of the input.

return_indices   if TRUE, will return the indices along with the outputs. Useful to pass to nn_max_unpool3d(). Default: FALSE

Examples

```r
if (torch_is_installed()) {
  # target output size of 5x7
  m <- nn_adaptive_max_pool2d(c(5, 7))
  input <- torch_randn(1, 64, 8, 9)
  output <- m(input)
  # target output size of 7x7 (square)
  m <- nn_adaptive_max_pool2d(7)
  input <- torch_randn(1, 64, 10, 9)
  output <- m(input)
}
```
Examples

```r
if (torch_is_installed()) {
  # target output size of 5x7x9
  m <- nn_adaptive_max_pool3d(c(5, 7, 9))
  input <- torch_randn(1, 64, 8, 9, 10)
  output <- m(input)
  # target output size of 7x7x7 (cube)
  m <- nn_adaptive_max_pool3d(7)
  input <- torch_randn(1, 64, 10, 9, 8)
  output <- m(input)
}
```

---

**nn_avg_pool1d**

Applies a 1D average pooling over an input signal composed of several input planes.

---

**Description**

In the simplest case, the output value of the layer with input size \((N, C, L)\), output \((N, C, L_{out})\) and kernel size \(k\) can be precisely described as:

\[
out(N_i, C_j, l) = \frac{1}{k} \sum_{m=0}^{k-1} input(N_i, C_j, stride \times l + m)
\]

**Usage**

```r
nn_avg_pool1d(
  kernel_size,
  stride = NULL,
  padding = 0,
  ceil_mode = FALSE,
  count_include_pad = TRUE
)
```

**Arguments**

- `kernel_size`: the size of the window
- `stride`: the stride of the window. Default value is `kernel_size`
- `padding`: implicit zero padding to be added on both sides
- `ceil_mode`: when TRUE, will use ceiling instead of floor to compute the output shape
- `count_include_pad`: when TRUE, will include the zero-padding in the averaging calculation

**Details**

\[
out(N_i, C_j, l) = \frac{1}{k} \sum_{m=0}^{k-1} input(N_i, C_j, stride \times l + m)
\]
If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points.

The parameters kernel_size, stride, padding can each be an int or a one-element tuple.

Shape

- Input: \((N, C, L_{\text{in}})\)
- Output: \((N, C, L_{\text{out}})\), where

\[
L_{\text{out}} = \left\lfloor \frac{L_{\text{in}} + 2 \times \text{padding} - \text{kernel}\_size}{\text{stride}} + 1 \right\rfloor
\]

Examples

```r
if (torch_is_installed()) {
  # pool with window of size=3, stride=2
  m <- nn_avg_pool1d(3, stride = 2)
  m(torch_randn(1, 1, 8))
}
```

**nn_avg_pool2d**

Applies a 2D average pooling over an input signal composed of several input planes.

Description

In the simplest case, the output value of the layer with input size \((N, C, H, W)\), output \((N, C, H_{\text{out}}, W_{\text{out}})\) and kernel_size \((kH, kW)\) can be precisely described as:

Usage

```r
nn_avg_pool2d(
  kernel_size = ,
  stride = NULL,
  padding = 0,
  ceil_mode = FALSE,
  count_include_pad = TRUE,
  divisor_override = NULL
)
```

Arguments

- **kernel_size** the size of the window
- **stride** the stride of the window. Default value is kernel_size
- **padding** implicit zero padding to be added on both sides
ceiling_mode when TRUE, will use ceiling instead of floor to compute the output shape

count_include_pad when TRUE, will include the zero-padding in the averaging calculation
deriver_override if specified, it will be used as divisor, otherwise kernel_size will be used

details

\[
\text{out}(N, C, h, w) = \frac{1}{kH \times kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \text{input}(N, C, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)
\]

If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points.

The parameters kernel_size, stride, padding can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

shape

- Input: \((N, C, H_{in}, W_{in})\)
- Output: \((N, C, H_{out}, W_{out})\), where

\[
H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel_size}[0]}{\text{stride}[0]} + 1 \right\rfloor
\]

\[
W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel_size}[1]}{\text{stride}[1]} + 1 \right\rfloor
\]

examples

```r
if (torch_is_installed()) {

# pool of square window of size=3, stride=2
m <- nn_avg_pool2d(3, stride = 2)
# pool of non-square window
m <- nn_avg_pool2d(c(3, 2), stride = c(2, 1))
input <- torch_randn(20, 16, 50, 32)
output <- m(input)
}
```
nn_avg_pool3d

Applies a 3D average pooling over an input signal composed of several input planes.

**Description**

In the simplest case, the output value of the layer with input size \((N, C, D, H, W)\), output \((N, C, D_{out}, H_{out}, W_{out})\) and kernel size \((kD, kH, kW)\) can be precisely described as:

**Usage**

```python
nen_avg_pool3d(
    kernel_size,
    stride = NULL,
    padding = 0,
    ceil_mode = FALSE,
    count_include_pad = TRUE,
    divisor_override = NULL
)
```

**Arguments**

- `kernel_size` the size of the window
- `stride` the stride of the window. Default value is `kernel_size`
- `padding` implicit zero padding to be added on all three sides
- `ceil_mode` when TRUE, will use `ceil` instead of `floor` to compute the output shape
- `count_include_pad` when TRUE, will include the zero-padding in the averaging calculation
- `divisor_override` if specified, it will be used as divisor, otherwise `kernel_size` will be used

**Details**

\[
    \text{out}(N_i, C_j, d, h, w) = \sum_{k=0}^{kD-1} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \frac{\text{input}(N_i, C_j, \text{stride[0]} \times d + k, \text{stride[1]} \times h + m, \text{stride[2]} \times w + n)}{kD \times kH \times kW}
\]

If padding is non-zero, then the input is implicitly zero-padded on all three sides for padding number of points.

The parameters `kernel_size`, `stride` can either be:

- a single `int` – in which case the same value is used for the depth, height and width dimension
- a tuple of three `ints` – in which case, the first `int` is used for the depth dimension, the second `int` for the height dimension and the third `int` for the width dimension
nn\_batch\_norm1d

Shape

- **Input:** \((N, C, D_{in}, H_{in}, W_{in})\)
- **Output:** \((N, C, D_{out}, H_{out}, W_{out})\), where

\[
D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor
\]
\[
H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor
\]
\[
W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{kernel\_size}[2]}{\text{stride}[2]} + 1 \right\rfloor
\]

Examples

```python
if (torch_is_installed()) {
    # pool of square window of size=3, stride=2
    m <- nn_avg_pool3d(3, stride = 2)
    # pool of non-square window
    m <- nn_avg_pool3d(c(3, 2, 2), stride = c(2, 1, 2))
    input <- torch_randn(20, 16, 50, 44, 31)
    output <- m(input)
}
```

nn\_batch\_norm1d  
**BatchNorm1D module**

Description

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*

Usage

```python
nn\_batch\_norm1d(
    num\_features,  
    eps = 1e-05,  
    momentum = 0.1,  
    affine = TRUE,  
    track\_running\_stats = TRUE
)
```
Arguments

- **num_features**: \(C\) from an expected input of size \((N, C, L)\) or \(L\) from input of size \((N, L)\)
- **eps**: a value added to the denominator for numerical stability. Default: 1e-5
- **momentum**: the value used for the \(\text{running}\_\text{mean}\) and \(\text{running}\_\text{var}\) computation. Can be set to \texttt{NULL} for cumulative moving average (i.e. simple average). Default: 0.1
- **affine**: a boolean value that when set to \texttt{TRUE}, this module has learnable affine parameters. Default: \texttt{TRUE}
- **track_running_stats**: a boolean value that when set to \texttt{TRUE}, this module tracks the running mean and variance, and when set to \texttt{FALSE}, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: \texttt{TRUE}

Details

\[
y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \cdot \gamma + \beta
\]

The mean and standard-deviation are calculated per-dimension over the mini-batches and \(\gamma\) and \(\beta\) are learnable parameter vectors of size \(C\) (where \(C\) is the input size). By default, the elements of \(\gamma\) are set to 1 and the elements of \(\beta\) are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default \texttt{momentum} of 0.1. If \texttt{track_running_stats} is set to \texttt{FALSE}, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

**Note**

This \texttt{momentum} argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is \(\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t\), where \(\hat{x}\) is the estimated statistic and \(x_t\) is the new observed value.

Because the Batch Normalization is done over the \(C\) dimension, computing statistics on \((N, L)\) slices, it’s common terminology to call this Temporal Batch Normalization.

**Shape**

- **Input**: \((N, C)\) or \((N, C, L)\)
- **Output**: \((N, C)\) or \((N, C, L)\) (same shape as input)

**Examples**

```r
if (torch_is_installed()) {
  # With Learnable Parameters
  m <- nn_batch_norm1d(100)
  # Without Learnable Parameters
  m <- nn_batch_norm1d(100, affine = FALSE)
  input <- torch_randn(20, 100)
```
Description

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

Usage

```r
nn_batch_norm2d(
  num_features,
  eps = 1e-05,
  momentum = 0.1,
  affine = TRUE,
  track_running_stats = TRUE
)
```

Arguments

- `num_features`: $C$ from an expected input of size $(N, C, H, W)$
- `eps`: a value added to the denominator for numerical stability. Default: 1e-5
- `momentum`: the value used for the running_mean and running_var computation. Can be set to None for cumulative moving average (i.e. simple average). Default: 0.1
- `affine`: a boolean value that when set to TRUE, this module has learnable affine parameters. Default: TRUE
- `track_running_stats`: a boolean value that when set to TRUE, this module tracks the running mean and variance, and when set to FALSE, this module does not track such statistics and uses batch statistics instead in both training and eval modes if the running mean and variance are None. Default: TRUE

Details

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and $\gamma$ and $\beta$ are learnable parameter vectors of size $C$ (where $C$ is the input size). By default, the elements of $\gamma$ are set to 1 and the elements of $\beta$ are set to 0. The standard-deviation is calculated via the biased estimator, equivalent to `torch_var(input, unbiased=False)`. Also by default, during training
this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default momentum of 0.1.

If track_running_stats is set to FALSE, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Shape

- Input: \((N, C, H, W)\)
- Output: \((N, C, H, W)\) (same shape as input)

Note

This momentum argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is

\[
\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t,
\]

where \(\hat{x}\) is the estimated statistic and \(x_t\) is the new observed value. Because the Batch Normalization is done over the \(C\) dimension, computing statistics on \((N, H, W)\) slices, it’s common terminology to call this Spatial Batch Normalization.

Examples

```r
if (torch_is_installed()) {
  # With Learnable Parameters
  m <- nn_batch_norm2d(100)
  # Without Learnable Parameters
  m <- nn_batch_norm2d(100, affine = FALSE)
  input <- torch_randn(20, 100, 35, 45)
  output <- m(input)
}
```

---

**nn_batch_norm3d**  
*BatchNorm3D*

Description

Applies Batch Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.*

Usage

```r
nn_batch_norm3d(
  num_features,  
  eps = 1e-05,  
  momentum = 0.1,  
  affine = TRUE,  
  track_running_stats = TRUE
)
```
nn_batch_norm3d

Arguments

num_features  
C from an expected input of size \((N, C, D, H, W)\)

eps  
a value added to the denominator for numerical stability. Default: 1e-5

momentum  
the value used for the running_mean and running_var computation. Can be set to None for cumulative moving average (i.e. simple average). Default: 0.1

affine  
a boolean value that when set to TRUE, this module has learnable affine parameters. Default: TRUE

track_running_stats  
a boolean value that when set to TRUE, this module tracks the running mean and variance, and when set to FALSE, this module does not track such statistics and uses batch statistics instead in both training and eval modes if the running mean and variance are None. Default: TRUE

Details

\[
y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \ast \gamma + \beta
\]

The mean and standard-deviation are calculated per-dimension over the mini-batches and \(\gamma\) and \(\beta\) are learnable parameter vectors of size \(C\) (where \(C\) is the input size). By default, the elements of \(\gamma\) are set to 1 and the elements of \(\beta\) are set to 0. The standard-deviation is calculated via the biased estimator, equivalent to \(\text{torch_var(input, unbiased = FALSE)}\).

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default momentum of 0.1.

If track_running_stats is set to FALSE, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Shape

- Input: \((N, C, D, H, W)\)
- Output: \((N, C, D, H, W)\) (same shape as input)

Note

This momentum argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is: \(\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t\), where \(\hat{x}\) is the estimated statistic and \(x_t\) is the new observed value.

Because the Batch Normalization is done over the \(C\) dimension, computing statistics on \((N, D, H, W)\) slices, it’s common terminology to call this Volumetric Batch Normalization or Spatio-temporal Batch Normalization.
Examples

```c
if (torch_is_installed()) {
  # With Learnable Parameters
  m <- nn_batch_norm3d(100)
  # Without Learnable Parameters
  m <- nn_batch_norm3d(100, affine = FALSE)
  input <- torch_randn(20, 100, 35, 45, 55)
  output <- m(input)
}
```

---

**nn_bce_loss**  
*Binary cross entropy loss*

**Description**

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

**Usage**

```c
nn_bce_loss(weight = NULL, reduction = "mean")
```

**Arguments**

- `weight`  
  (Tensor, optional): a manual rescaling weight given to the loss of each batch element. If given, has to be a Tensor of size \( n_{\text{batch}} \).

- `reduction`  
  (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

**Details**

The unreduced (i.e. with \( \text{reduction} = \text{'none'} \)) loss can be described as:

\[
\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \quad l_n = -w_n [y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]
\]

where \( N \) is the batch size. If reduction is not 'none' (default 'mean'), then

\[
\ell(x, y) = \begin{cases} 
  \text{mean}(L), & \text{if} \ \text{reduction} = \text{'mean'}, \\
  \text{sum}(L), & \text{if} \ \text{reduction} = \text{'sum'}.
\end{cases}
\]

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets \( y \) should be numbers between 0 and 1.

Notice that if \( x_n \) is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set \( \log(0) = -\infty \), since \( \lim_{x \to 0} \log(x) = -\infty \).

However, an infinite term in the loss equation is not desirable for several reasons. For one, if either \( y_n = 0 \) or \((1 - y_n) = 0\), then we would be multiplying 0 with infinity. Secondly, if
we have an infinite loss value, then we would also have an infinite term in our gradient, since \( \lim_{x \to 0} \frac{d}{dx} \log(x) = \infty \).

This would make BCELoss’s backward method nonlinear with respect to \( x_n \), and using it for things like linear regression would not be straight-forward. Our solution is that BCELoss clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Target: \((N, *)\), same shape as the input
- Output: scalar. If reduction is 'none', then \((N, *)\), same shape as input.

Examples

```java
if (torch_is_installed()) {
    m <- nn_sigmoid()
    loss <- nn_bce_loss()
    input <- torch_randn(3, requires_grad = TRUE)
    target <- torch_rand(3)
    output <- loss(m(input), target)
    output$backward()
}
```

**nn_bce_with_logits_loss**

*BCE with logits loss*

**Description**

This loss combines a Sigmoid layer and the BCELoss in one single class. This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

**Usage**

```
nn_bce_with_logits_loss(weight = NULL, reduction = "mean", pos_weight = NULL)
```

**Arguments**

- **weight** (Tensor, optional): a manual rescaling weight given to the loss of each batch element. If given, has to be a Tensor of size \( n_{\text{batch}} \).

- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

- **pos_weight** (Tensor, optional): a weight of positive examples. Must be a vector with length equal to the number of classes.
Details

The unreduced (i.e. with reduction set to 'none') loss can be described as:

$$
\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \quad l_n = -w_n [y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))],
$$

where $N$ is the batch size. If reduction is not 'none' (default 'mean'), then

$$
\ell(x, y) = \begin{cases} 
\text{mean}(L), & \text{if reduction = 'mean'}; \\
\sum(L), & \text{if reduction = 'sum'}. 
\end{cases}
$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets $t[i]$ should be numbers between 0 and 1. It’s possible to trade off recall and precision by adding weights to positive examples. In the case of multi-label classification the loss can be described as:

$$
\ell_c(x, y) = L_c = \{l_{1,c}, \ldots, l_{N,c}\}^T, \quad l_{n,c} = -w_{n,c} [p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))],
$$

where $c$ is the class number ($c > 1$ for multi-label binary classification, $c = 1$ for single-label binary classification), $n$ is the number of the sample in the batch and $p_c$ is the weight of the positive answer for the class $c$. $p_c > 1$ increases the recall, $p_c < 1$ increases the precision. For example, if a dataset contains 100 positive and 300 negative examples of a single class, then pos_weight for the class should be equal to \(\frac{300}{100} = 3\). The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

Shape

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If reduction is 'none', then $(N, *)$, same shape as input.

Examples

```r
if (torch_is_installed()) {
  loss <- nn_bce_with_logits_loss()
  input <- torch_randn(3, requires_grad = TRUE)
  target <- torch_empty(3)$random_(1, 2)
  output <- loss(input, target)
  output$backward()
  target <- torch_ones(10, 64, dtype = torch_float32()) # 64 classes, batch size = 10
  output <- torch_full(c(10, 64), 1.5) # A prediction (logit)
  pos_weight <- torch_ones(64) # All weights are equal to 1
  criterion <- nn_bce_with_logits_loss(pos_weight = pos_weight)
  criterion(output, target) # -log(sigmoid(1.5))
}
```
**nn_bilinear**

**Bilinear module**

**Description**
Applies a bilinear transformation to the incoming data \( y = x_1^T A x_2 + b \)

**Usage**
\[
nn\_bilinear(in1\_features, in2\_features, out\_features, bias = TRUE)
\]

**Arguments**
- `in1_features`: size of each first input sample
- `in2_features`: size of each second input sample
- `out_features`: size of each output sample
- `bias`: If set to `FALSE`, the layer will not learn an additive bias. Default: `TRUE`

**Shape**
- Input 1: \((N, *, H_{in1}) \) where \( H_{in1} = \) `in1_features` and `*` means any number of additional dimensions. All but the last dimension of the inputs should be the same.
- Input 2: \((N, *, H_{in2}) \) where \( H_{in2} = \) `in2_features`.
- Output: \((N, *, H_{out}) \) where \( H_{out} = \) `out_features` and all but the last dimension are the same shape as the input.

**Attributes**
- `weight`: the learnable weights of the module of shape \((\text{out\_features}, \text{in1\_features}, \text{in2\_features})\).
The values are initialized from \( \mathcal{U}( -\sqrt{k}, \sqrt{k}) \), where \( k = \frac{1}{\text{in1\_features}} \)
- `bias`: the learnable bias of the module of shape \((\text{out\_features})\). If `bias` is `TRUE`, the values are initialized from \( \mathcal{U}(-\sqrt{k}, \sqrt{k}) \), where \( k = \frac{1}{\text{in1\_features}} \)

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_bilinear(20, 30, 50)
  input1 <- torch_randn(128, 20)
  input2 <- torch_randn(128, 30)
  output <- m(input1, input2)
  print(output$size())
}
```
nn_buffer

Description

Indicates that a tensor is a buffer in a nn_module

Usage

`nn_buffer(x, persistent = TRUE)`

Arguments

- `x` the tensor that will be converted to nn_buffer
- `persistent` whether the buffer should be persistent or not.

nn_celu

Description

Applies the element-wise function:

Usage

`nn_celu(alpha = 1, inplace = FALSE)`

Arguments

- `alpha` the $\alpha$ value for the CELU formulation. Default: 1.0
- `inplace` can optionally do the operation in-place. Default: FALSE

Details

$$CELU(x) = \max(0, x) + \min(0, \alpha \ast (\exp(x/\alpha) - 1))$$

More details can be found in the paper Continuously Differentiable Exponential Linear Units.

Shape

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input
Examples

```r
if (torch_is_installed()) {
  m <- nn_celu()
  input <- torch_randn(2)
  output <- m(input)
}
```

**nn_contrib_sparsemax**  
*Sparsemax activation*  

**Description**  
Sparsemax activation module.

**Usage**

```r
nn_contrib_sparsemax(dim = -1)
```

**Arguments**

- **dim**  
The dimension over which to apply the sparsemax function. (-1)

**Details**

The SparseMax activation is described in 'From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification' The implementation is based on aced125/sparsemax

**nn_conv1d**  
*Conv1D module*  

**Description**

Applies a 1D convolution over an input signal composed of several input planes. In the simplest case, the output value of the layer with input size \((N, C_{in}, L)\) and output \((N, C_{out}, L_{out})\) can be precisely described as:

**Usage**

```r
nn_conv1d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1,
  bias = TRUE,
  padding_mode = "zeros"
)
```
Arguments

- `in_channels` (int): Number of channels in the input image
- `out_channels` (int): Number of channels produced by the convolution
- `kernel_size` (int or tuple): Size of the convolving kernel
- `stride` (int or tuple, optional): Stride of the convolution. Default: 1
- `padding` (int, tuple or str, optional) – Padding added to both sides of the input. Default: 0
- `dilation` (int, tuple, optional): Spacing between kernel elements. Default: 1
- `groups` (int, optional): Number of blocked connections from input channels to output channels. Default: 1
- `bias` (bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
- `padding_mode` (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

\[
\text{out}(N_i, C_{out,j}) = \text{bias}(C_{out,j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out,j}, k) * \text{input}(N_i, k)
\]

where * is the valid cross-correlation operator, \(N\) is a batch size, \(C\) denotes a number of channels, \(L\) is a length of signal sequence.

- `stride` controls the stride for the cross-correlation, a single number or a one-element tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for padding number of points.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At `groups`=1, all inputs are convolved to all outputs.
  - At `groups`=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At `groups`= `in_channels`, each input channel is convolved with its own set of filters, of size \( \left\lceil \frac{\text{out\_channels}}{\text{in\_channels}} \right\rceil \).

Note

Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

When `groups == in_channels` and `out_channels == K * in_channels`, where \(K\) is a positive integer, this operation is also termed in literature as depthwise convolution. In other words, for an input of size \((N, C_{in}, L_{in})\), a depthwise convolution with a depthwise multiplier \(K\), can be constructed by arguments \((C_{in} = C_{in}, C_{out} = C_{in} \times K, ..., groups = C_{in})\).
Shape

- Input: \((N, C_{in}, L_{in})\)
- Output: \((N, C_{out}, L_{out})\) where

\[
L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel}_{\text{size}} - 1) - 1}{\text{stride}} + 1 \right\rfloor
\]

Attributes

- \text{weight} (Tensor): the learnable weights of the module of shape \((\text{out}_{\text{channels}}, \frac{\text{in}_{\text{channels}}}{\text{groups}}, \text{kernel}_{\text{size}})\). The values of these weights are sampled from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{groups}}{\text{in}_{\text{channels}} \times \text{kernel}_{\text{size}}}\).
- \text{bias} (Tensor): the learnable bias of the module of shape \((\text{out}_{\text{channels}})\). If \text{bias} is TRUE, then the values of these weights are sampled from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{groups}}{\text{in}_{\text{channels}} \times \text{kernel}_{\text{size}}}\).

Examples

```r
if (torch_is_installed()) {
  m <- nn_conv1d(16, 33, 3, stride = 2)
  input <- torch_randn(20, 16, 50)
  output <- m(input)
}
```

nn_conv2d

Conv2D module

Description

Applies a 2D convolution over an input signal composed of several input planes.

Usage

```r
nn_conv2d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1,
  bias = TRUE,
  padding_mode = "zeros"
)
```
Arguments

- `in_channels` (int): Number of channels in the input image
- `out_channels` (int): Number of channels produced by the convolution
- `kernel_size` (int or tuple): Size of the convolving kernel
- `stride` (int or tuple, optional): Stride of the convolution. Default: 1
- `padding` (int or tuple or string, optional): Zero-padding added to both sides of the input. It can be either a string 'valid', 'same' or a tuple of ints giving the amount of implicit padding applied on both sides. Default: 0
- `dilation` (int or tuple, optional): Spacing between kernel elements. Default: 1
- `groups` (int, optional): Number of blocked connections from input channels to output channels. Default: 1
- `bias` (bool, optional): If True, adds a learnable bias to the output. Default: True
- `padding_mode` (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

In the simplest case, the output value of the layer with input size \((N, C_{in}, H, W)\) and output \((N, C_{out}, H_{out}, W_{out})\) can be precisely described as:

\[
out(N_i, C_{out,j}) = bias(C_{out,j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out,j}, k) \star input(N_i, k)
\]

where \(\star\) is the valid 2D cross-correlation operator, \(N\) is a batch size, \(C\) denotes a number of channels, \(H\) is a height of input planes in pixels, and \(W\) is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for padding number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At `groups`=1, all inputs are convolved to all outputs.
  - At `groups`=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At `groups`= `in_channels`, each input channel is convolved with its own set of filters, of size: \(\left\lfloor \frac{out_channels}{in_channels} \right\rfloor\).

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension
**Note**

Depending on the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

When \(\text{groups} == \text{in\_channels}\) and \(\text{out\_channels} == K \times \text{in\_channels}\), where \(K\) is a positive integer, this operation is also termed in literature as depthwise convolution. In other words, for an input of size :math:`(N, C_{in}, H_{in}, W_{in})`, a depthwise convolution with a depthwise multiplier \(K\), can be constructed by arguments \((\text{in\_channels} = C_{in}, \text{out\_channels} = C_{in} \times K, ..., \text{groups} = C_{in})\).

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting \(\text{backends\_cudnn\_deterministic} = \text{TRUE}\).

**Shape**

- **Input**: \((N, C_{in}, H_{in}, W_{in})\)
- **Output**: \((N, C_{out}, H_{out}, W_{out})\) where

  \[
  H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor
  \]

  \[
  W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor
  \]

**Attributes**

- **weight** (Tensor): the learnable weights of the module of shape \((\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])\). The values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{C_{in} \times \Pi_{i=0}^{\text{groups}} \text{kernel\_size}[i]}{\text{groups}}\).

- **bias** (Tensor): the learnable bias of the module of shape \((\text{out\_channels})\). If \(\text{bias} = \text{TRUE}\), then the values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{C_{in} \times \Pi_{i=0}^{\text{groups}} \text{kernel\_size}[i]}{\text{groups}}\).

**Examples**

```python
if (torch_is_installed()) {

    # With square kernels and equal stride
    m <- nn_conv2d(16, 33, 3, stride = 2)
    # non-square kernels and unequal stride and with padding
    m <- nn_conv2d(16, 33, c(3, 5), stride = c(2, 1), padding = c(4, 2))
    # non-square kernels and unequal stride and with padding and dilation
    m <- nn_conv2d(16, 33, c(3, 5), stride = c(2, 1), padding = c(4, 2), dilation = c(3, 1))
    input <- torch_randn(20, 16, 50, 100)
    output <- m(input)
}
```
nn_conv3d  

**Description**

Applies a 3D convolution over an input signal composed of several input planes. In the simplest case, the output value of the layer with input size \((N, C_{in}, D, H, W)\) and output \((N, C_{out}, D_{out}, H_{out}, W_{out})\) can be precisely described as:

\[
\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \ast \text{input}(N_i, k)
\]

where \(\ast\) is the valid 3D cross-correlation operator.

**Usage**

```python
nn_conv3d(
in_channels,
out_channels,
kernel_size,
stride = 1,
padding = 0,
dilation = 1,
groups = 1,
bias = TRUE,
padding_mode = "zeros"
)
```

**Arguments**

- `in_channels` (int): Number of channels in the input image
- `out_channels` (int): Number of channels produced by the convolution
- `kernel_size` (int or tuple): Size of the convolving kernel
- `stride` (int or tuple, optional): Stride of the convolution. Default: 1
- `padding` (int, tuple or str, optional): Padding added to all six sides of the input. Default: 0
- `dilation` (int or tuple, optional): Spacing between kernel elements. Default: 1
- `groups` (int, optional): Number of blocked connections from input channels to output channels. Default: 1
- `bias` (bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
- `padding_mode` (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

**Details**

\[
\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \ast \text{input}(N_i, k)
\]
• **stride** controls the stride for the cross-correlation.

• **padding** controls the amount of implicit zero-paddings on both sides for padding number of points for each dimension.

• **dilation** controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.

• **groups** controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,

  • At `groups`=1, all inputs are convolved to all outputs.

  • At `groups`=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

  • At `groups`= `in_channels`, each input channel is convolved with its own set of filters, of size `out_channels / in_channels`.

The parameters **kernel_size**, **stride**, **padding**, **dilation** can either be:

• a single int – in which case the same value is used for the depth, height and width dimension

• a tuple of three ints – in which case, the first int is used for the depth dimension, the second int for the height dimension and the third int for the width dimension

**Shape**

• Input: `(N, C_{in}, D_{in}, H_{in}, W_{in})`

• Output: `(N, C_{out}, D_{out}, H_{out}, W_{out})` where

\[
D_{out} = \left\lceil \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rceil
\]

\[
H_{out} = \left\lceil \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rceil
\]

\[
W_{out} = \left\lceil \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rceil
\]

**Attributes**

• **weight** (Tensor): the learnable weights of the module of shape `(out_channels, in_channels \_ groups, kernel_size[0], kernel_size[1], kernel_size[2])`. The values of these weights are sampled from \( \mathcal{U}(-\sqrt{k}, \sqrt{k}) \) where \( k = \frac{C_{in} \times \prod_{i=0}^{groups} \text{kernel_size}[i]}{C_{out} \times \prod_{i=0}^{groups} \text{kernel_size}[i]} \)

• **bias** (Tensor): the learnable bias of the module of shape `(out_channels)`. If `bias` is True, then the values of these weights are sampled from \( \mathcal{U}(-\sqrt{k}, \sqrt{k}) \) where \( k = \frac{C_{in} \times \prod_{i=0}^{groups} \text{kernel_size}[i]}{C_{out} \times \prod_{i=0}^{groups} \text{kernel_size}[i]} \)
**Note**

Depending on the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

When `groups == in_channels` and `out_channels == K * in_channels`, where `K` is a positive integer, this operation is also termed in literature as depthwise convolution. In other words, for an input of size \(N, C_{in}, D_{in}, H_{in}, W_{in}\), a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments \(in\_channels = C_{in}, out\_channels = C_{in} \times K, ..., groups = C_{in}\).

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = TRUE`. Please see the notes on :doc:`/notes/randomness` for background.

**Examples**

```r
if (torch_is_installed()) {
  # With square kernels and equal stride
  m <- nn_conv3d(16, 33, 3, stride = 2)
  # non-square kernels and unequal stride and with padding
  m <- nn_conv3d(16, 33, c(3, 5, 2), stride = c(2, 1, 1), padding = c(4, 2, 0))
  input <- torch_randn(20, 16, 10, 50, 100)
  output <- m(input)
}
```

### nn_conv_transpose1d  
**ConvTranspose1D**

**Description**

Applies a 1D transposed convolution operator over an input image composed of several input planes.

**Usage**

```r
nen_conv_transpose1d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  output_padding = 0,
  groups = 1,
  bias = TRUE,
  dilation = 1,
  padding_mode = "zeros"
)
```
Arguments

- **in_channels** (int): Number of channels in the input image
- **out_channels** (int): Number of channels produced by the convolution
- **kernel_size** (int or tuple): Size of the convolving kernel
- **stride** (int or tuple, optional): Stride of the convolution. Default: 1
- **padding** (int or tuple, optional): dilation * (kernel_size - 1) - padding zero-padding will be added to both sides of the input. Default: 0
- **output_padding** (int or tuple, optional): Additional size added to one side of the output shape. Default: 0
- **groups** (int, optional): Number of blocked connections from input channels to output channels. Default: 1
- **bias** (bool, optional): If True, adds a learnable bias to the output. Default: TRUE
- **dilation** (int or tuple, optional): Spacing between kernel elements. Default: 1
- **padding_mode** (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

This module can be seen as the gradient of Conv1d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- **stride** controls the stride for the cross-correlation.
- **padding** controls the amount of implicit zero-paddings on both sides for dilation * (kernel_size - 1) - padding number of points. See note below for details.
- **output_padding** controls the additional size added to one side of the output shape. See note below for details.
- **dilation** controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.
- **groups** controls the connections between inputs and outputs. in_channels and out_channels must both be divisible by groups. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups=in_channels, each input channel is convolved with its own set of filters (of size \( \frac{out\_channels}{in\_channels} \)).

Shape

- **Input:** \((N, C_{in}, L_{in})\)
- **Output:** \((N, C_{out}, L_{out})\)

\[
L_{out} = (L_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{dilation} \times (\text{kernel}\_\text{size} - 1) + \text{output}\_\text{padding} + 1
\]
nn_conv_transpose2d

Attributes

- weight (Tensor): the learnable weights of the module of shape \((\text{in_channels}, \frac{\text{out_channels}}{\text{groups}}, \text{kernel_size})\). The values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{groups}}{\text{out_channels} \times \text{kernel_size}}\)
- bias (Tensor): the learnable bias of the module of shape (out_channels). If bias is TRUE, then the values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{groups}}{\text{out_channels} \times \text{kernel_size}}\)

Note

Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

The padding argument effectively adds \(\text{dilation} \times (\text{kernel_size} - 1) - \text{padding}\) amount of zero padding to both sizes of the input. This is set so that when a \texttt{torch.nn.Conv1d} and a \texttt{torch.nn.ConvTranspose1d} are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when stride > 1, \texttt{torch.nn.Conv1d} maps multiple input shapes to the same output shape. \text{output_padding} is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that \text{output_padding} is only used to find output shape, but does not actually add zero-padding to output.

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting \texttt{torch.backends.cudnn.deterministic = TRUE}.

Examples

```python
if (torch_is_installed()) {
    m <- nn_conv_transpose1d(32, 16, 2)
    input <- torch_randn(10, 32, 2)
    output <- m(input)
}
```

nn_conv_transpose2d  \(\text{ConvTranspose2D module}\)

Description

Applies a 2D transposed convolution operator over an input image composed of several input planes.

Usage

```python
nn_conv_transpose2d(
    in_channels,
    out_channels,
    kernel_size,
    stride = 1,
)```
nn_conv_transpose2d

```
padding = 0,
output_padding = 0,
groups = 1,
bias = TRUE,
dilation = 1,
padding_mode = "zeros"
```

Arguments

- **in_channels** (int): Number of channels in the input image
- **out_channels** (int): Number of channels produced by the convolution
- **kernel_size** (int or tuple): Size of the convolving kernel
- **stride** (int or tuple, optional): Stride of the convolution. Default: 1
- **padding** (int or tuple, optional): dilation * (kernel_size - 1) - padding zero-padding will be added to both sides of each dimension in the input. Default: 0
- **output_padding** (int or tuple, optional): Additional size added to one side of each dimension in the output shape. Default: 0
- **groups** (int, optional): Number of blocked connections from input channels to output channels. Default: 1
- **bias** (bool, optional): If True, adds a learnable bias to the output. Default: True
- **dilation** (int or tuple, optional): Spacing between kernel elements. Default: 1
- **padding_mode** (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- **stride** controls the stride for the cross-correlation.
- **padding** controls the amount of implicit zero-paddings on both sides for dilation * (kernel_size - 1) - padding number of points. See note below for details.
- **output_padding** controls the additional size added to one side of the output shape. See note below for details.
- **dilation** controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what dilation does.
- **groups** controls the connections between inputs and outputs. in_channels and out_channels must both be divisible by groups. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
At \texttt{groups} = \texttt{in\_channels}, each input channel is convolved with its own set of filters (of size \[ \frac{\text{out\_channels}}{\text{in\_channels}} \]).

The parameters kernel\_size, stride, padding, output\_padding can either be:

- a single \texttt{int} – in which case the same value is used for the height and width dimensions
- a tuple of two \texttt{ints} – in which case, the first \texttt{int} is used for the height dimension, and the second \texttt{int} for the width dimension

\section*{Shape}

\begin{itemize}
  \item Input: \((N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})\)
  \item Output: \((N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})\) where
    \[ H_{\text{out}} = (H_{\text{in}} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1 \]
    \[ W_{\text{out}} = (W_{\text{in}} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1 \]
\end{itemize}

\section*{Attributes}

\begin{itemize}
  \item weight (Tensor): the learnable weights of the module of shape \((\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])\). The values of these weights are sampled from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{C_{\text{out}} \times \Pi_{i=0}^{\text{groups}} \text{kernel\_size}[i]}{\text{groups}}\)
  \item bias (Tensor): the learnable bias of the module of shape \((\text{out\_channels})\) If \texttt{bias} is \texttt{True}, then the values of these weights are sampled from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{C_{\text{out}} \times \Pi_{i=0}^{\text{groups}} \text{kernel\_size}[i]}{\text{groups}}\)
\end{itemize}

\section*{Note}

Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

The padding argument effectively adds \(\text{dilation} \times (\text{kernel\_size} - 1) - \text{padding}\) amount of zero padding to both sizes of the input. This is set so that when a \texttt{nn\_conv2d} and a \texttt{nn\_conv\_transpose2d} are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when \texttt{stride} > 1, \texttt{nn\_conv2d} maps multiple input shapes to the same output shape. \texttt{output\_padding} is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that \texttt{output\_padding} is only used to find output shape, but does not actually add zero-padding to output.

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting \texttt{torch.backends.cudnn.deterministic = TRUE}. 

**nn_conv_transpose3d**

*ConvTranpose3D module*

**Description**

Applies a 3D transposed convolution operator over an input image composed of several input planes.

**Usage**

```r
nn_conv_transpose3d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  output_padding = 0,
  groups = 1,
  bias = TRUE,
  dilation = 1,
  padding_mode = "zeros"
)
```

**Arguments**

- `in_channels` (int): Number of channels in the input image
- `out_channels` (int): Number of channels produced by the convolution
- `kernel_size` (int or tuple): Size of the convolving kernel
- `stride` (int or tuple, optional): Stride of the convolution. Default: 1

**Examples**

```r
if (torch_is_installed()) {
    # With square kernels and equal stride
    m <- nn_conv_transpose2d(16, 33, 3, stride = 2)
    # non-square kernels and unequal stride and with padding
    m <- nn_conv_transpose2d(16, 33, c(3, 5), stride = c(2, 1), padding = c(4, 2))
    input <- torch_randn(20, 16, 50, 100)
    output <- m(input)
    # exact output size can be also specified as an argument
    input <- torch_randn(1, 16, 12, 12)
    downsample <- nn_conv2d(16, 16, 3, stride = 2, padding = 1)
    upsample <- nn_conv_transpose2d(16, 16, 3, stride = 2, padding = 1)
    h <- downsample(input)
    h$size()
    output <- upsample(h, output_size = input$size())
    output$size()
}
```
nn_conv_transpose3d

padding (int or tuple, optional): dilation * (kernel_size - 1) - padding zero-padding will be added to both sides of each dimension in the input. Default: 0

output_padding (int or tuple, optional): Additional size added to one side of each dimension in the output shape. Default: 0

groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1

bias (bool, optional): If True, adds a learnable bias to the output. Default: True

dilation (int or tuple, optional): Spacing between kernel elements. Default: 1

padding_mode (string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

The transposed convolution operator multiplies each input value element-wise by a learnable kernel, and sums over the outputs from all input feature planes.

This module can be seen as the gradient of Conv3d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- stride controls the stride for the cross-correlation.
- padding controls the amount of implicit zero-paddings on both sides for dilation * (kernel_size - 1) - padding number of points. See note below for details.
- output_padding controls the additional size added to one side of the output shape. See note below for details.
- dilation controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link_ has a nice visualization of what dilation does.
- groups controls the connections between inputs and outputs. in_channels and out_channels must both be divisible by groups. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups= in_channels, each input channel is convolved with its own set of filters (of size \( \left\lfloor \frac{\text{out_channels}}{\text{in_channels}} \right\rfloor \))

The parameters kernel_size, stride, padding, output_padding can either be:

- a single int – in which case the same value is used for the depth, height and width dimensions
- a tuple of three ints – in which case, the first int is used for the depth dimension, the second int for the height dimension and the third int for the width dimension
Shape

- Input: \((N, C_{\text{in}}, D_{\text{in}}, H_{\text{in}}, W_{\text{in}})\)
- Output: \((N, C_{\text{out}}, D_{\text{out}}, H_{\text{out}}, W_{\text{out}})\) where

\[
\begin{align*}
D_{\text{out}} &= (D_{\text{in}} - 1) \times \text{stride}_0 - 2 \times \text{padding}_0 + \text{dilation}_0 \times (\text{kernel}_0 - 1) + \text{output_padding}_0 + 1 \\
H_{\text{out}} &= (H_{\text{in}} - 1) \times \text{stride}_1 - 2 \times \text{padding}_1 + \text{dilation}_1 \times (\text{kernel}_1 - 1) + \text{output_padding}_1 + 1 \\
W_{\text{out}} &= (W_{\text{in}} - 1) \times \text{stride}_2 - 2 \times \text{padding}_2 + \text{dilation}_2 \times (\text{kernel}_2 - 1) + \text{output_padding}_2 + 1
\end{align*}
\]

Attributes

- weight (Tensor): the learnable weights of the module of shape \((\text{in_channels}, \text{out_channels} / \text{groups}, \text{kernel}_0, \text{kernel}_1, \text{kernel}_2)\). The values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{out_channels} \times \text{in_channels} \times \text{product of kernel sizes}}{\text{groups}}\)
- bias (Tensor): the learnable bias of the module of shape \((\text{out_channels})\). If bias is True, then the values of these weights are sampled from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{\text{out_channels} \times \text{in_channels} \times \text{product of kernel sizes}}{\text{groups}}\)

Note

Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

The padding argument effectively adds \(\text{dilation} \times (\text{kernel} - 1) - \text{padding}\) amount of zero padding to both sizes of the input. This is set so that when a \texttt{"torch.nn.Conv3d"} and a \texttt{"torch.nn.ConvTranspose3d"} are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when \texttt{stride} > 1, \texttt{"torch.nn.Conv3d"} maps multiple input shapes to the same output shape. \texttt{output_padding} is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that \texttt{output_padding} is only used to find output shape, but does not actually add zero-padding to output.

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting \texttt{torch.backends.cudnn.deterministic = TRUE}.

Examples

```python
if (torch_is_installed()) {
    ## Not run:
    # With square kernels and equal stride
    m <- nn_conv_transpose3d(16, 33, 3, stride = 2)
    # non-square kernels and unequal stride and with padding
    m <- nn_conv_transpose3d(16, 33, c(3, 5, 2), stride = c(2, 1, 1), padding = c(0, 4, 2))
    input <- torch_randn(20, 16, 10, 50, 100)
    output <- m(input)
}
```

## End(Not run)
nn_cosine_embedding_loss

Cosine embedding loss

Description

Creates a criterion that measures the loss given input tensors $x_1, x_2$ and a Tensor label $y$ with values 1 or -1. This is used for measuring whether two inputs are similar or dissimilar, using the cosine distance, and is typically used for learning nonlinear embeddings or semi-supervised learning. The loss function for each sample is:

Usage

```python
nn_cosine_embedding_loss(margin = 0, reduction = "mean")
```

Arguments

- **margin** (float, optional): Should be a number from $-1$ to $1$, $0$ to $0.5$ is suggested. If `margin` is missing, the default value is $0$.
- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

$$loss(x, y) = \begin{cases} 
1 - \cos(x_1, x_2), & \text{if } y = 1 \\
\max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 
\end{cases}$$

nn_cross_entropy_loss

CrossEntropyLoss module

Description

This criterion combines `nn_log_softmax()` and `nn_nll_loss()` in one single class. It is useful when training a classification problem with $C$ classes.

Usage

```python
nn_cross_entropy_loss(weight = NULL, ignore_index = -100, reduction = "mean")
```
nn_cross_entropy_loss

Arguments

weight (Tensor, optional): a manual rescaling weight given to each class. If given, has to be a Tensor of size \( C \)

ignore_index (int, optional): Specifies a target value that is ignored and does not contribute to the input gradient. When size_average is \(\text{TRUE}\), the loss is averaged over non-ignored targets.

reduction (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

If provided, the optional argument weight should be a 1D Tensor assigning weight to each of the classes.

This is particularly useful when you have an unbalanced training set. The input is expected to contain raw, unnormalized scores for each class. Input has to be a Tensor of size either \((\text{minibatch}, C)\) or \((\text{minibatch}, C, d_1, d_2, ..., d_K)\) with \( K \geq 1 \) for the K-dimensional case (described later).

This criterion expects a class index in the range \([0, C - 1]\) as the target for each value of a 1D tensor of size minibatch; if ignore_index is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

\[
\text{loss}(x, \text{class}) = - \log \left( \frac{\exp(x[\text{class}])}{\sum_j \exp(x[j])} \right) = -x[\text{class}] + \log \left( \sum_j \exp(x[j]) \right)
\]

or in the case of the weight argument being specified:

\[
\text{loss}(x, \text{class}) = \text{weight}[\text{class}] \left( -x[\text{class}] + \log \left( \sum_j \exp(x[j]) \right) \right)
\]

The losses are averaged across observations for each minibatch. Can also be used for higher dimension inputs, such as 2D images, by providing an input of size \((\text{minibatch}, C, d_1, d_2, ..., d_K)\) with \( K \geq 1 \), where \( K \) is the number of dimensions, and a target of appropriate shape (see below).

Shape

- Input: \((N, C)\) where \( C = \text{number of classes} \), or \((N, C, d_1, d_2, ..., d_K)\) with \( K \geq 1 \) in the case of K-dimensional loss.
- Target: \((N)\) where each value is \( 0 \leq \text{targets}[i] \leq C - 1 \), or \((N, d_1, d_2, ..., d_K)\) with \( K \geq 1 \) in the case of K-dimensional loss.
- Output: scalar. If reduction is 'none', then the same size as the target: \((N)\), or \((N, d_1, d_2, ..., d_K)\) with \( K \geq 1 \) in the case of K-dimensional loss.
Examples

```r
if (torch_is_installed()) {
  loss <- nn_cross_entropy_loss()
  input <- torch_randn(3, 5, requires_grad = TRUE)
  target <- torch_randint(low = 1, high = 5, size = 3, dtype = torch_long())
  output <- loss(input, target)
  output$backward()
}
```

nn_ctc_loss

*The Connectionist Temporal Classification loss.*

**Description**

Calculates loss between a continuous (unsegmented) time series and a target sequence. CTCLoss sums over the probability of possible alignments of input to target, producing a loss value which is differentiable with respect to each input node. The alignment of input to target is assumed to be "many-to-one", which limits the length of the target sequence such that it must be $\leq$ the input length.

**Usage**

```r
nn_ctc_loss(blank = 0, reduction = "mean", zero_infinity = FALSE)
```

**Arguments**

- **blank** (int, optional): blank label. Default 0.
- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the output losses will be divided by the target lengths and then the mean over the batch is taken. Default: 'mean'
- **zero_infinity** (bool, optional): Whether to zero infinite losses and the associated gradients. Default: FALSE Infinite losses mainly occur when the inputs are too short to be aligned to the targets.

**Shape**

- **Log_probs**: Tensor of size $(T, N, C)$, where $T = \text{input length}$, $N = \text{batch size}$, and $C = \text{number of classes (including blank)}$. The logarithmized probabilities of the outputs (e.g. obtained with `nnflog_softmax()`).
- **Targets**: Tensor of size $(N, S)$ or (sum(target_lengths)), where $N = \text{batch size}$ and $S = \text{max target length}$, if shape is $(N, S)$. It represent the target sequences. Each element in the target sequence is a class index. And the target index cannot be blank (default=0). In the $(N, S)$ form, targets are padded to the length of the longest sequence, and stacked. In the (sum(target_lengths)) form, the targets are assumed to be un-padded and concatenated within 1 dimension.
Input_lengths: Tuple or tensor of size \((N)\), where \(N = \text{batch size}\). It represents the lengths of the inputs (must each be \(\leq T\)). And the lengths are specified for each sequence to achieve masking under the assumption that sequences are padded to equal lengths.

Target_lengths: Tuple or tensor of size \((N)\), where \(N = \text{batch size}\). It represents lengths of the targets. Lengths are specified for each sequence to achieve masking under the assumption that sequences are padded to equal lengths. If target shape is \((N, S)\), target_lengths are effectively the stop index \(s_n\) for each target sequence, such that \(\text{target}_n = \text{targets}[n, 0:s_n]\) for each target in a batch. Lengths must each be \(\leq S\). If the targets are given as a 1d tensor that is the concatenation of individual targets, the target_lengths must add up to the total length of the tensor.

Output: scalar. If reduction is 'none', then \((N)\), where \(N = \text{batch size}\).

\[[\text{nnf})\log\text{softmax}()]: \text{R:nnf})\log\text{softmax}()[n,0:s_n]: \text{R:n,0:s_n}\]

Note

In order to use CuDNN, the following must be satisfied: targets must be in concatenated format, all input_lengths must be \(T\). blank = 0, target_lengths \(\leq 256\), the integer arguments must be of The regular implementation uses the (more common in PyTorch) torch_long dtype. dtype torch_int32.

In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting torch.backends.cudnn.deterministic = TRUE.

References


Examples

```r
if (torch_is_installed()) {
  # Target are to be padded
  T <- 50  # Input sequence length
  C <- 20  # Number of classes (including blank)
  N <- 16  # Batch size
  S <- 30  # Target sequence length of longest target in batch (padding length)
  S_min <- 10  # Minimum target length, for demonstration purposes

  # Initialize random batch of input vectors, for *size = (T,N,C)*
  input <- torch_randn(T, N, C)$log_softmax(2)$detach()$requires_grad()

  # Initialize random batch of targets (0 = blank, 1:C = classes)
  target <- torch_randint(low = 1, high = C, size = c(N, S), dtype = torch_long())

  input_lengths <- torch_full(size = c(N), fill_value = TRUE, dtype = torch_long())
  target_lengths <- torch_randint(low = S_min, high = S, size = c(N), dtype = torch_long())
  ctc_loss <- nn_ctc_loss()
  loss <- ctc_loss(input, target, input_lengths, target_lengths)
```
nn_dropout

# Target are to be un-padded
T <- 50 # Input sequence length
C <- 20 # Number of classes (including blank)
N <- 16 # Batch size

# Initialize random batch of input vectors, for *size = (T,N,C)
input <- torch_randn(T, N, C)$log_softmax(2)$detach()$requires_grad_()
input_lengths <- torch_full(size = c(N), fill_value = TRUE, dtype = torch_long())

# Initialize random batch of targets (0 = blank, 1:C = classes)
target_lengths <- torch_randint(low = 1, high = T, size = c(N), dtype = torch_long())
target <- torch_randint(low = 1, high = C, size = as.integer(sum(target_lengths)),
dtype = torch_long() )
ctc_loss <- nn_ctc_loss()
loss <- ctc_loss(input, target, input_lengths, target_lengths)
loss$backward()

nn_dropout

Dropout module

Description

During training, randomly zeroes some of the elements of the input tensor with probability p using
samples from a Bernoulli distribution. Each channel will be zeroed out independently on every
forward call.

Usage

nn_dropout(p = 0.5, inplace = FALSE)

Arguments

 p probability of an element to be zeroed. Default: 0.5
 inplace If set to TRUE, will do this operation in-place. Default: FALSE.

Details

This has proven to be an effective technique for regularization and preventing the co-adaptation of
neurons as described in the paper Improving neural networks by preventing co-adaptation of feature
detectors.

Furthermore, the outputs are scaled by a factor of :math:`\frac{1}{1-p}` during training. This
means that during evaluation the module simply computes an identity function.
**nn_dropout2d**

**Shape**
- Input: (*). Input can be of any shape
- Output: (*). Output is of the same shape as input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_dropout(p = 0.2)
  input <- torch_randn(20, 16)
  output <- m(input)
}
```

---

**nn_dropout2d**

*Dropout2D module*

**Description**

Randomly zero out entire channels (a channel is a 2D feature map, e.g., the \( j \)-th channel of the \( i \)-th sample in the batched input is a 2D tensor \( \text{input}[i,j] \)).

**Usage**

```r
nn_dropout2d(p = 0.5, inplace = FALSE)
```

**Arguments**

- **p**
  - (float, optional): probability of an element to be zero-ed.
- **inplace**
  - (bool, optional): If set to TRUE, will do this operation in-place

**Details**

Each channel will be zeroed out independently on every forward call with probability \( p \) using samples from a Bernoulli distribution. Usually the input comes from `nn_conv2d` modules.

As described in the paper *Efficient Object Localization Using Convolutional Networks*, if adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then i.i.d. dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, `nn_dropout2d` will help promote independence between feature maps and should be used instead.

**Shape**

- Input: \( (N, C, H, W) \)
- Output: \( (N, C, H, W) \) (same shape as input)
Examples

```r
if (torch_is_installed()) {
  m <- nn_dropout2d(p = 0.2)
  input <- torch_randn(20, 16, 32, 32)
  output <- m(input)
}
```

---

### nn_dropout3d

**Dropout3D module**

**Description**

Randomly zero out entire channels (a channel is a 3D feature map, e.g., the \( j \)-th channel of the \( i \)-th sample in the batched input is a 3D tensor \( \text{input}[i,j] \)).

**Usage**

```r
nn_dropout3d(p = 0.5, inplace = FALSE)
```

**Arguments**

- `p` (float, optional): probability of an element to be zeroed.
- `inplace` (bool, optional): If set to `TRUE`, will do this operation in-place

**Details**

Each channel will be zeroed out independently on every forward call with probability \( p \) using samples from a Bernoulli distribution. Usually the input comes from `nn_conv2d` modules.

As described in the paper *Efficient Object Localization Using Convolutional Networks*, if adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then i.i.d. dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease.

In this case, `nn_dropout3d` will help promote independence between feature maps and should be used instead.

**Shape**

- **Input:** \( (N, C, D, H, W) \)
- **Output:** \( (N, C, D, H, W) \) (same shape as input)

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_dropout3d(p = 0.2)
  input <- torch_randn(20, 16, 4, 32, 32)
  output <- m(input)
}
```
nn_elu

**ELU module**

**Description**

Applies the element-wise function:

**Usage**

```r
nn_elu(alpha = 1, inplace = FALSE)
```

**Arguments**

- `alpha` the $\alpha$ value for the ELU formulation. Default: 1.0
- `inplace` can optionally do the operation in-place. Default: FALSE

**Details**

\[
\text{ELU}(x) = \max(0, x) + \min(0, \alpha \cdot (\exp(x) - 1))
\]

**Shape**

- Input: $(N, \ast)$ where $\ast$ means, any number of additional dimensions
- Output: $(N, \ast)$, same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_elu()
  input <- torch_randn(2)
  output <- m(input)
}
```

nn_embedding

**Embedding module**

**Description**

A simple lookup table that stores embeddings of a fixed dictionary and size. This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.
Usage

```
nn_embedding(
    num_embeddings,
    embedding_dim,
    padding_idx = NULL,
    max_norm = NULL,
    norm_type = 2,
    scale_grad_by_freq = FALSE,
    sparse = FALSE,
    .weight = NULL
)
```

Arguments

- `num_embeddings` (int): size of the dictionary of embeddings
- `embedding_dim` (int): the size of each embedding vector
- `padding_idx` (int, optional): If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- `max_norm` (float, optional): If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- `norm_type` (float, optional): The p of the p-norm to compute for the `max_norm` option. Default 2.
- `scale_grad_by_freq` (boolean, optional): If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default False.
- `sparse` (bool, optional): If True, gradient w.r.t. weight matrix will be a sparse tensor.
- `.weight` (Tensor): embeddings weights (in case you want to set it manually)

See Notes for more details regarding sparse gradients.

Attributes

- `weight` (Tensor): the learnable weights of the module of shape `(num_embeddings, embedding_dim)` initialized from $\mathcal{N}(0, 1)$

Shape

- Input: (*) LongTensor of arbitrary shape containing the indices to extract
- Output: (*, H), where * is the input shape and $H = \text{embedding_dim}$

Note

Keep in mind that only a limited number of optimizers support sparse gradients: currently it’s `optim.SGD` (CUDA and CPU), `optim.SparseAdam` (CUDA and CPU) and `optim.Adagrad` (CPU).

With `padding_idx` set, the embedding vector at `padding_idx` is initialized to all zeros. However, note that this vector can be modified afterwards, e.g., using a customized initialization method, and thus changing the vector used to pad the output. The gradient for this vector from `nn_embedding` is always zero.
**nn_embedding_bag**

**Embedding bag module**

**Examples**

```r
if (torch_is_installed()) {
  # an Embedding module containing 10 tensors of size 3
  embedding <- nn_embedding(10, 3)
  # a batch of 2 samples of 4 indices each
  input <- torch_tensor(rbind(c(1, 2, 4, 5), c(4, 3, 2, 9)), dtype = torch_long())
  embedding(input)
  # example with padding_idx
  embedding <- nn_embedding(10, 3, padding_idx = 1)
  input <- torch_tensor(matrix(c(1, 3, 1, 6), nrow = 1), dtype = torch_long())
  embedding(input)
}
```

**Description**

Computes sums, means or maxes of bags of embeddings, without instantiating the intermediate embeddings.

**Usage**

```r
nn_embedding_bag(
  num_embeddings, embedding_dim, 
  max_norm = NULL, norm_type = 2, scale_grad_by_freq = FALSE, mode = "mean", 
  sparse = FALSE, include_last_offset = FALSE, padding_idx = NULL, 
  .weight = NULL
)
```

**Arguments**

- `num_embeddings` (int): size of the dictionary of embeddings
- `embedding_dim` (int): the size of each embedding vector
- `max_norm` (float, optional): If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- `norm_type` (float, optional): The p of the p-norm to compute for the `max_norm` option. Default `2`.
- `scale_grad_by_freq` (boolean, optional): If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`. 
**nn_flatten**

Flattens a contiguous range of dims into a tensor.

**Description**

For use with nn_sequential.

**Usage**

nn_flatten(start_dim = 2, end_dim = -1)
nn_fractional_max_pool2d

Arguments

- **start_dim**: first dim to flatten (default = 2).
- **end_dim**: last dim to flatten (default = -1).

Shape

- Input: (*, S_start,..., S_i, ..., S_end, *), where S_i is the size at dimension i and * means any number of dimensions including none.
- Output: (*, S_start*...*S_i*...S_end, *).

See Also

- [nn_unflatten](#)

Examples

```r
if (torch_is_installed()) {
  input <- torch_randn(32, 1, 5, 5)
  m <- nn_flatten()
  m(input)
}
```

nn_fractional_max_pool2d

Applies a 2D fractional max pooling over an input signal composed of several input planes.

Description

Fractional MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

Usage

```r
nn_fractional_max_pool2d(
  kernel_size,
  output_size = NULL,
  output_ratio = NULL,
  return_indices = FALSE
)
```

Arguments

- **kernel_size**: the size of the window to take a max over. Can be a single number k (for a square kernel of k x k) or a tuple (kh, kw)
- **output_size**: the target output size of the image of the form oH x oW. Can be a tuple (oH, oW) or a single number oH for a square image oH x oH
nn_fractional_max_pool3d

Applies a 3D fractional max pooling over an input signal composed of several input planes.

Details

The max-pooling operation is applied in $kH \times kW$ regions by a stochastic step size determined by the target output size. The number of output features is equal to the number of input planes.

Examples

```r
if (torch_is_installed()) {
  # pool of square window of size=3, and target output size 13x12
  m <- nn_fractional_max_pool2d(3, output_size = c(13, 12))
  # pool of square window and target output size being half of input image size
  m <- nn_fractional_max_pool2d(3, output_ratio = c(0.5, 0.5))
  input <- torch_randn(20, 16, 50, 32)
  output <- m(input)
}
```

nn_fractional_max_pool3d

Applies a 3D fractional max pooling over an input signal composed of several input planes.

Description

Fractional MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

Usage

```r
nn_fractional_max_pool3d(
  kernel_size,
  output_size = NULL,
  output_ratio = NULL,
  return_indices = FALSE
)
```

Arguments

- `kernel_size`: the size of the window to take a max over. Can be a single number $k$ (for a square kernel of $k \times k \times k$) or a tuple ($kt \times kh \times kw$)
- `output_size`: the target output size of the image of the form $oT \times oH \times oW$. Can be a tuple ($oT, oH, oW$) or a single number $oH$ for a square image $oH \times oH \times oH$
- `output_ratio`: If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range $(0, 1)$
- `return_indices`: if TRUE, will return the indices along with the outputs. Useful to pass to `nn_max_unpool3d()`. Default: FALSE
The max-pooling operation is applied in $kT \times kH \times kW$ regions by a stochastic step size determined by the target output size. The number of output features is equal to the number of input planes.

Examples

```r
if (torch_is_installed()) {
  # pool of cubic window of size=3, and target output size 13x12x11
  m <- nn_fractional_max_pool3d(3, output_size = c(13, 12, 11))
  # pool of cubic window and target output size being half of input size
  m <- nn_fractional_max_pool3d(3, output_ratio = c(0.5, 0.5, 0.5))
  input <- torch_randn(20, 16, 50, 32, 16)
  output <- m(input)
}
```

---

**GELU module**

**Description**

Applies the Gaussian Error Linear Units function:

$$GELU(x) = x \cdot \Phi(x)$$

**Usage**

```r
nn_gelu(approximate = "none")
```

**Arguments**

- `approximate`: the gelu approximation algorithm to use: 'none' or 'tanh'. Default: 'none'.

**Details**

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

**Shape**

- Input: $(N, \ast)$ where $\ast$ means, any number of additional dimensions
- Output: $(N, \ast)$, same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_gelu()
  input <- torch_randn(2)
  output <- m(input)
}
```
nn_glu | \textit{GLU module}
--- | ---

### Description

Applies the gated linear unit function $GLU(a, b) = a \otimes \sigma(b)$ where $a$ is the first half of the input matrices and $b$ is the second half.

### Usage

nn_glu(dim = -1)

### Arguments

- **dim** (int): the dimension on which to split the input. Default: -1

### Shape

- Input: $(*)_1, N, (*)_2$ where $*$ means, any number of additional dimensions
- Output: $(*)_1, M, (*)_2$ where $M = N/2$

### Examples

```r
if (torch_is_installed()) {
  m <- nn_glu()
  input <- torch_randn(4, 2)
  output <- m(input)
}
```

nn_group_norm | \textit{Group normalization}
--- | ---

### Description

Applies Group Normalization over a mini-batch of inputs as described in the paper \textit{Group Normalization}.

### Usage

nn_group_norm(num_groups, num_channels, eps = 1e-05, affine = TRUE)
**Arguments**

- `num_groups` (int): number of groups to separate the channels into
- `num_channels` (int): number of channels expected in input
- `eps` a value added to the denominator for numerical stability. Default: 1e-5
- `affine` a boolean value that when set to `TRUE`, this module has learnable per-channel affine parameters initialized to ones (for weights) and zeros (for biases). Default: `TRUE`.

**Details**

\[ y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \ast \gamma + \beta \]

The input channels are separated into `num_groups` groups, each containing \(\frac{\text{num_channels}}{\text{num_groups}}\) channels. The mean and standard-deviation are calculated separately over the each group. \(\gamma\) and \(\beta\) are learnable per-channel affine transform parameter vectors of size `num_channels` if `affine` is `TRUE`. The standard-deviation is calculated via the biased estimator, equivalent to `torch_var(input, unbiased=FALSE)`.

**Shape**

- Input: \((N, C, *)\) where \(C = \text{num_channels}\)
- Output: \((N, C, *)\) (same shape as input)

**Note**

This layer uses statistics computed from input data in both training and evaluation modes.

**Examples**

```r
if (torch_is_installed()) {

  input <- torch_randn(20, 6, 10, 10)
  # Separate 6 channels into 3 groups
  m <- nn_group_norm(3, 6)
  # Separate 6 channels into 6 groups (equivalent with [nn_instance_morm])
  m <- nn_group_norm(6, 6)
  # Put all 6 channels into a single group (equivalent with [nn_layer_norm])
  m <- nn_group_norm(1, 6)
  # Activating the module
  output <- m(input)
}
```
nn_gru

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

Description

For each element in the input sequence, each layer computes the following function:

Usage

```
n_gru(
    input_size,
    hidden_size,
    num_layers = 1,
    bias = TRUE,
    batch_first = FALSE,
    dropout = 0,
    bidirectional = FALSE,
    ...
  )
```

Arguments

- `input_size`: The number of expected features in the input x
- `hidden_size`: The number of features in the hidden state h
- `num_layers`: Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a stacked GRU, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- `bias`: If `FALSE`, then the layer does not use bias weights $b_{ih}$ and $b_{hh}$. Default: `TRUE`
- `batch_first`: If `TRUE`, then the input and output tensors are provided as (batch, seq, feature). Default: `FALSE`
- `dropout`: If non-zero, introduces a Dropout layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: `0`
- `bidirectional`: If `TRUE`, becomes a bidirectional GRU. Default: `FALSE`
- `...`: currently unused.

Details

\[
\begin{align*}
  r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
  z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
  n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\
  h_t &= (1 - z_t)n_t + z_t h_{(t-1)}
\end{align*}
\]

where $h_t$ is the hidden state at time $t$, $x_t$ is the input at time $t$, $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time 0, and $r_t$, $z_t$, $n_t$ are the reset, update, and new gates, respectively. $\sigma$ is the sigmoid function.
**nn_gru**

**Inputs**

Inputs: input, h_0

- **input** of shape (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `nn_utils.rnn_pack_padded_sequence()` for details.
- **h_0** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

**Outputs**

Outputs: output, h_n

- **output** of shape (seq_len, batch, num_directions * hidden_size): tensor containing the output features h_t from the last layer of the GRU, for each t. If a PackedSequence has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using `output$\text{view}(c(seq_len, batch, num_directions, hidden_size))`, with forward and backward being direction 0 and 1 respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for t = seq_len. Like output, the layers can be separated using `h_n$\text{view}(num_layers, num_directions, batch, hidden_size)`.

**Attributes**

- **weight_ih_l[k]**: the learnable input-hidden weights of the k\textsuperscript{th} layer (W_ir|W_iz|W_in), of shape (3*hidden_size x input_size)
- **weight_hh_l[k]**: the learnable hidden-hidden weights of the k\textsuperscript{th} layer (W_hr|W_hz|W_hn), of shape (3*hidden_size x hidden_size)
- **bias_ih_l[k]**: the learnable input-hidden bias of the k\textsuperscript{th} layer (b_ir|b_iz|b_in), of shape (3*hidden_size)
- **bias_hh_l[k]**: the learnable hidden-hidden bias of the k\textsuperscript{th} layer (b_hr|b_hz|b_hn), of shape (3*hidden_size)

**Note**

All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden_size}}$

**Examples**

```r
if (torch_is_installed()) {

  rnn <- nn_gru(10, 20, 2)
  input <- torch_randn(5, 3, 10)
  h0 <- torch_randn(2, 3, 20)
  output <- rnn(input, h0)
}
```
nn_hardsigmoid

Hardsigmoid module

Description

Applies the element-wise function:

Usage

nn_hardsigmoid()

Arguments

None

Details

HardSigmoid

Examples

if (torch_is_installed()) {
  m <- nn_hardsigmoid()
  input <- torch_randn(2)
  output <- m(input)
}
nn_hardswish

Details

\[
\text{Hardsigmoid}(x) = \begin{cases} 
0 & \text{if } x \leq -3, \\
1 & \text{if } x \geq +3, \\
x/6 + 1/2 & \text{otherwise}
\end{cases}
\]

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_hardsigmoid()
  input <- torch_randn(2)
  output <- m(input)
}
```

nn_hardswish  

Hardswish module

Description

Applies the hardswish function, element-wise, as described in the paper: Searching for MobileNetV3

Usage

nn_hardswish()

Details

\[
\text{Hardswish}(x) = \begin{cases} 
0 & \text{if } x \leq -3, \\
x & \text{if } x \geq +3, \\
x \cdot (x + 3)/6 & \text{otherwise}
\end{cases}
\]

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input
**Examples**

```r
if (torch_is_installed()) {
  ## Not run:
  m <- nn_hardswish()
  input <- torch_randn(2)
  output <- m(input)

  ## End(Not run)
}
```

**nn_hardtanh**

*Hardtanh module*

**Description**

Applies the HardTan function element-wise. HardTan is defined as:

**Usage**

```r
nn_hardtanh(min_val = -1, max_val = 1, inplace = FALSE)
```

**Arguments**

- `min_val`: minimum value of the linear region range. Default: -1
- `max_val`: maximum value of the linear region range. Default: 1
- `inplace`: can optionally do the operation in-place. Default: FALSE

**Details**

\[
\text{HardTanh}(x) = \begin{cases} 
  1 & \text{if } x > 1 \\
  -1 & \text{if } x < -1 \\
  x & \text{otherwise}
\end{cases}
\]

The range of the linear region \([1, 1]\) can be adjusted using `min_val` and `max_val`.

**Shape**

- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_hardtanh(-2, 2)
  input <- torch_randn(2)
  output <- m(input)
}
```
nn_hinge_embedding_loss

Hinge embedding loss

Description

Measures the loss given an input tensor \( x \) and a labels tensor \( y \) (containing 1 or -1).

Usage

```python
nn_hinge_embedding_loss(margin = 1, reduction = "mean")
```

Arguments

- **margin** (float, optional): Has a default value of 1.
- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance as \( x \), and is typically used for learning nonlinear embeddings or semi-supervised learning. The loss function for \( n \)-th sample in the mini-batch is

\[
l_n = x_n, \quad \text{if } y_n = 1, \\
\max\{0, \Delta - x_n\}, \quad \text{if } y_n = -1,
\]

and the total loss functions is

\[
\ell(x, y) = \begin{cases} 
\frac{\text{mean}(L)}{\text{sum}(L)}, & \text{if reduction = 'mean'}; \\
\frac{\text{mean}(L)}{\text{sum}(L)}, & \text{if reduction = 'sum'}.
\end{cases}
\]

where \( L = \{l_1, \ldots, l_N\}^\top \).

Shape

- Input: \((*)\) where * means, any number of dimensions. The sum operation operates over all the elements.
- Target: \((*)\), same shape as the input
- Output: scalar. If reduction is 'none', then same shape as the input
nn_identity

Identity module

Description

A placeholder identity operator that is argument-insensitive.

Usage

nn_identity(…)

Arguments

... any arguments (unused)

Examples

if (torch_is_installed()) {
  m <- nn_identity(54, unused_argument1 = 0.1, unused_argument2 = FALSE)
  input <- torch_randn(128, 20)
  output <- m(input)
  print(output$size())
}

nn_init_calculate_gain

Calculate gain

Description

Return the recommended gain value for the given nonlinearity function.

Usage

nn_init_calculate_gain(nonlinearity, param = NULL)

Arguments

nonlinearity the non-linear function
param optional parameter for the non-linear function
**nn_init_constant_**

*Constant initialization*

**Description**

Fills the input Tensor with the value val.

**Usage**

nn_init_constant_(tensor, val)

**Arguments**

tensor: an n-dimensional Tensor
val: the value to fill the tensor with

**Examples**

```r
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_constant_(w, 0.3)
}
```

---

**nn_init_dirac_**

*Dirac initialization*

**Description**

Fills the {3, 4, 5}-dimensional input Tensor with the Dirac delta function. Preserves the identity of the inputs in Convolutional layers, where as many input channels are preserved as possible. In case of groups>1, each group of channels preserves identity.

**Usage**

nn_init_dirac_(tensor, groups = 1)

**Arguments**

tensor: a {3, 4, 5}-dimensional torch.Tensor
groups: (optional) number of groups in the conv layer (default: 1)
Examples

```r
if (torch_is_installed()) {
  ## Not run:
  w <- torch_empty(3, 16, 5, 5)
  nn_init_dirac_(w)

  ## End(Not run)
}
```

**nn_init_eye_**  
*Eye initialization*

**Description**

Fills the 2-dimensional input Tensor with the identity matrix. Preserves the identity of the inputs in Linear layers, where as many inputs are preserved as possible.

**Usage**

```r
nn_init_eye_(tensor)
```

**Arguments**

- **tensor** a 2-dimensional torch tensor.

**Examples**

```r
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_eye_(w)
}
```

---

**nn_init_kaiming_normal_**

*Kaiming normal initialization*

**Description**

Fills the input Tensor with values according to the method described in Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015), using a normal distribution.
nn_init_kaiming_uniform_

Usage

nn_init_kaiming_normal_(
    tensor,
    a = 0,
    mode = "fan_in",
    nonlinearity = "leaky_relu"
)

Arguments

tensor an n-dimensional torch.Tensor
a the negative slope of the rectifier used after this layer (only used with 'leaky_relu')
mode either 'fan_in' (default) or 'fan_out'. Choosing 'fan_in' preserves the magnitude of the variance of the weights in the forward pass. Choosing 'fan_out' preserves the magnitudes in the backwards pass.
nonlinearity the non-linear function. recommended to use only with 'relu' or 'leaky_relu' (default).

Examples

if (torch_is_installed()) {
    w <- torch_empty(3, 5)
    nn_init_kaiming_normal_(w, mode = "fan_in", nonlinearity = "leaky_relu")
}

nn_init_kaiming_uniform_

Kaiming uniform initialization

Description

Fills the input Tensor with values according to the method described in Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015), using a uniform distribution.

Usage

nn_init_kaiming_uniform_(
    tensor,
    a = 0,
    mode = "fan_in",
    nonlinearity = "leaky_relu"
)
Arguments

tensor  an n-dimensional torch.Tensor

a  the negative slope of the rectifier used after this layer (only used with 'leaky_relu')

mode  either 'fan_in' (default) or 'fan_out'. Choosing 'fan_in' preserves the magnitude of the variance of the weights in the forward pass. Choosing 'fan_out' preserves the magnitudes in the backwards pass.

nonlinearity  the non-linear function. recommended to use only with 'relu' or 'leaky_relu' (default).

Examples

if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_kaiming_uniform_(w, mode = "fan_in", nonlinearity = "leaky_relu")
}

nn_init_normal_  Normal initialization

Description

Fills the input Tensor with values drawn from the normal distribution

Usage

nn_init_normal_(tensor, mean = 0, std = 1)

Arguments

tensor  an n-dimensional Tensor

mean  the mean of the normal distribution

std  the standard deviation of the normal distribution

Examples

if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_normal_(w)
}
nn_init_ones_  

**Ones initialization**

**Description**

Fills the input Tensor with the scalar value 1

**Usage**

```
nn_init_ones_(tensor)
```

**Arguments**

- **tensor** an n-dimensional Tensor

**Examples**

```
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_ones_(w)
}
```

---

nn_init_orthogonal_  

**Orthogonal initialization**

**Description**

Fills the input Tensor with a (semi) orthogonal matrix, as described in *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* - Saxe, A. et al. (2013). The input tensor must have at least 2 dimensions, and for tensors with more than 2 dimensions the trailing dimensions are flattened.

**Usage**

```
nn_init_orthogonal_(tensor, gain = 1)
```

**Arguments**

- **tensor** an n-dimensional Tensor
- **gain** optional scaling factor

**Examples**

```
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_orthogonal_(w)
}
```
nn_init_sparse_  Sparse initialization

Description

Fills the 2D input Tensor as a sparse matrix, where the non-zero elements will be drawn from the normal distribution as described in Deep learning via Hessian-free optimization - Martens, J. (2010).

Usage

```
n_init_sparse_(tensor, sparsity, std = 0.01)
```

Arguments

- `tensor`: an n-dimensional Tensor
- `sparsity`: The fraction of elements in each column to be set to zero
- `std`: the standard deviation of the normal distribution used to generate the non-zero values

Examples

```
if (torch_is_installed()) {
  ## Not run:
  w <- torch_empty(3, 5)
  nn_init_sparse_(w, sparsity = 0.1)
  ## End(Not run)
}
```

nn_init_trunc_normal_  Truncated normal initialization

Description

Fills the input Tensor with values drawn from a truncated normal distribution.

Usage

```
n_init_trunc_normal_(tensor, mean = 0, std = 1, a = -2, b = 2)
```
\textbf{nn_init_uniform_} \hspace{1cm} \textit{Uniform initialization}

\textbf{Description}
Fills the input Tensor with values drawn from the uniform distribution

\textbf{Usage}
\begin{verbatim}
nn_init_uniform_(tensor, a = 0, b = 1)
\end{verbatim}

\textbf{Arguments}
\begin{itemize}
\item tensor \hspace{0.5cm} \text{an n-dimensional Tensor}
\item a \hspace{0.5cm} \text{the lower bound of the uniform distribution}
\item b \hspace{0.5cm} \text{the upper bound of the uniform distribution}
\end{itemize}

\textbf{Examples}
\begin{verbatim}
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_uniform_(w)
}
\end{verbatim}
nn_init_xavier_normal_

**Xavier normal initialization**

**Description**

Fills the input Tensor with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010), using a normal distribution.

**Usage**

```python
nn_init_xavier_normal_(tensor, gain = 1)
```

**Arguments**

- **tensor**: an n-dimensional Tensor
- **gain**: an optional scaling factor

**Examples**

```python
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_xavier_normal_(w)
}
```

---

nn_init_xavier_uniform_

**Xavier uniform initialization**

**Description**

Fills the input Tensor with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010), using a uniform distribution.

**Usage**

```python
nn_init_xavier_uniform_(tensor, gain = 1)
```

**Arguments**

- **tensor**: an n-dimensional Tensor
- **gain**: an optional scaling factor
nn_init_zeros_  Zeros initialization

Description
Fills the input Tensor with the scalar value 0

Usage
nn_init_zeros_(tensor)

Arguments
tensor  an n-dimensional tensor

Examples
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_xavier_uniform_(w)
}

nn_kl_div_loss  Kullback-Leibler divergence loss

Description
The Kullback-Leibler divergence loss measure Kullback-Leibler divergence is a useful distance measure for continuous distributions and is often useful when performing direct regression over the space of (discretely sampled) continuous output distributions.

Usage
nn_kl_div_loss(reduction = "mean")

Arguments
reduction  (string, optional): Specifies the reduction to apply to the output: 'none' | 'batchmean' | 'sum' | 'mean'. 'none': no reduction will be applied. 'batchmean': the sum of the output will be divided by batchsize. 'sum': the output will be summed. 'mean': the output will be divided by the number of elements in the output. Default: 'mean'
Details

As with `nn_nll_loss()`, the input given is expected to contain log-probabilities and is not restricted to a 2D Tensor.

The targets are interpreted as probabilities by default, but could be considered as log-probabilities with log_target set to TRUE.

This criterion expects a target Tensor of the same size as the input Tensor.

The unreduced (i.e. with reduction set to 'none') loss can be described as:

\[ l(x, y) = L = \{l_1, \ldots, l_N\}, \quad l_n = y_n \cdot (\log y_n - x_n) \]

where the index \(N\) spans all dimensions of input and \(L\) has the same shape as input. If reduction is not 'none' (default 'mean'), then:

\[ \ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases} \]

In default reduction mode 'mean', the losses are averaged for each minibatch over observations as well as over dimensions. 'batchmean' mode gives the correct KL divergence where losses are averaged over batch dimension only. 'mean' mode's behavior will be changed to the same as 'batchmean' in the next major release.

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Target: \((N, *)\), same shape as the input
- Output: scalar by default. If reduction is 'none', then \((N, *)\), the same shape as the input

Note

reduction = 'mean' doesn't return the true kl divergence value, please use reduction = 'batchmean' which aligns with KL math definition. In the next major release, 'mean' will be changed to be the same as 'batchmean'.

---

### nn_l1_loss

**L1 loss**

**Description**

Creates a criterion that measures the mean absolute error (MAE) between each element in the input \(x\) and target \(y\).

**Usage**

```python
nn_l1_loss(reduction = "mean")
```
Arguments

reduction (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

The unreduced (i.e. with reduction set to 'none') loss can be described as:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \quad l_n = |x_n - y_n|,$$

where $N$ is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \frac{\text{mean}(L)}{\text{sum}(L)}, \quad \text{if reduction = 'mean'};$$

$$\text{sum}(L), \quad \text{if reduction = 'sum'}.$$

$x$ and $y$ are tensors of arbitrary shapes with a total of $n$ elements each.

The sum operation still operates over all the elements, and divides by $n$. The division by $n$ can be avoided if one sets reduction = 'sum'.

Shape

- Input: $(N, \ast)$ where $\ast$ means, any number of additional dimensions
- Target: $(N, \ast)$, same shape as the input
- Output: scalar. If reduction is 'none', then $(N, \ast)$, same shape as the input

Examples

```r
if (torch_is_installed()) {
  loss <- nn_l1_loss()
  input <- torch_randn(3, 5, requires_grad = TRUE)
  target <- torch_randn(3, 5)
  output <- loss(input, target)
  output$backward()
}
```

Description

Applies Layer Normalization over a mini-batch of inputs as described in the paper Layer Normalization.
Usage

nn_layer_norm(normalized_shape, eps = 1e-05, elementwise_affine = TRUE)

Arguments

normalized_shape
(int or list): input shape from an expected input of size \( \ast \times \text{normalized_shape}[0] \times \text{normalized_shape}[1] \times \ldots \times \text{normalized_shape}[\text{-}1] \) If a single integer is used, it is treated as a singleton list, and this module will normalize over the last dimension which is expected to be of that specific size.

eps
a value added to the denominator for numerical stability. Default: 1e-5

elementwise_affine
a boolean value that when set to TRUE, this module has learnable per-element affine parameters initialized to ones (for weights) and zeros (for biases). Default: TRUE.

Details

\[ y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \cdot \gamma + \beta \]

The mean and standard-deviation are calculated separately over the last certain number dimensions which have to be of the shape specified by normalized_shape.

\( \gamma \) and \( \beta \) are learnable affine transform parameters of normalized_shape if elementwise_affine is TRUE.

The standard-deviation is calculated via the biased estimator, equivalent to torch_var(input, unbiased=FALSE).

Shape

- Input: \((N, \ast)\)
- Output: \((N, \ast)\) (same shape as input)

Note

Unlike Batch Normalization and Instance Normalization, which applies scalar scale and bias for each entire channel/plane with the affine option, Layer Normalization applies per-element scale and bias with elementwise_affine.

This layer uses statistics computed from input data in both training and evaluation modes.

Examples

```r
if (torch_is_installed()) {

input <- torch_randn(20, 5, 10, 10)
# With Learnable Parameters
m <- nn_layer_norm(input$size()[1])
```
nn_leaky_relu

LeakyReLU module

Description

Applies the element-wise function:

Usage

```r
nn_leaky_relu(negative_slope = 0.01, inplace = FALSE)
```

Arguments

- `negative_slope`: Controls the angle of the negative slope. Default: 1e-2
- `inplace`: Can optionally do the operation in-place. Default: FALSE

Details

LeakyReLU\(x\) = \max(0, x) + \text{negative_slope} \times \min(0, x)

or

LeakyReLU\(x\) = \begin{cases}  
  x, & \text{if } x \geq 0 \\
  \text{negative_slope} \times x, & \text{otherwise}
\end{cases}

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_leaky_relu(0.1)
  input <- torch_randn(2)
  output <- m(input)
}
```
**nn_linear**  
*Linear module*

**Description**
Applies a linear transformation to the incoming data: \( y = xA^T + b \)

**Usage**

\[
nn\text{linear}(\text{in\_features}, \text{out\_features}, \text{bias} = \text{TRUE})
\]

**Arguments**
- **in\_features**: size of each input sample
- **out\_features**: size of each output sample
- **bias**: If set to FALSE, the layer will not learn an additive bias. Default: TRUE

**Shape**
- Input: \((N, *, H_{in})\) where * means any number of additional dimensions and \(H_{in} = \text{in\_features}\).
- Output: \((N, *, H_{out})\) where all but the last dimension are the same shape as the input and \(H_{out} = \text{out\_features}\).

**Attributes**
- **weight**: the learnable weights of the module of shape \((\text{out\_features}, \text{in\_features})\). The values are initialized from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{1}{\text{in\_features}}\).
- **bias**: the learnable bias of the module of shape \((\text{out\_features})\). If bias is TRUE, the values are initialized from \(U(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{1}{\text{in\_features}}\).

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_linear(20, 30)
  input <- torch_randn(128, 20)
  output <- m(input)
  print(output$size())
}
```
**nn_log_sigmoid**

*LogSigmoid module*

**Description**

Applies the element-wise function:

\[
\text{LogSigmoid}(x) = \log \left( \frac{1}{1 + \exp(-x)} \right)
\]

**Usage**

\[\text{nn_log_sigmoid}()\]

**Shape**

- Input: \((N, \ast)\) where \(\ast\) means any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

**Examples**

\[
\text{if (torch_is_installed())} \{
    m <- \text{nn_log_sigmoid}()
    \text{input} <- \text{torch_randn(2)}
    \text{output} <- m(\text{input})
\}
\]

---

**nn_log_softmax**

*LogSoftmax module*

**Description**

Applies the \(\log(\text{Softmax}(x))\) function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as:

\[
\text{LogSoftmax}(x_i) = \log \left( \frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)
\]

**Usage**

\[\text{nn_log_softmax}(\text{dim})\]

**Arguments**

- \text{dim} (int): A dimension along which LogSoftmax will be computed.

**Details**

\[
\text{LogSoftmax}(x_i) = \log \left( \frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)
\]
Value

a Tensor of the same dimension and shape as the input with values in the range \([-\infty, 0)\)

Shape

- Input: \((*)\) where * means, any number of additional dimensions
- Output: \((*)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_log_softmax(1)
  input <- torch_randn(2, 3)
  output <- m(input)
}
```

**nn_lp_pool1d**

Applies a 1D power-average pooling over an input signal composed of several input planes.

**Description**

On each window, the function computed is:

**Usage**

```r
nn_lp_pool1d(norm_type, kernel_size, stride = NULL, ceil_mode = FALSE)
```

**Arguments**

- **norm_type**
  - if \( \text{inf} \) than one gets max pooling if 0 you get sum pooling (proportional to the avg pooling)
- **kernel_size**
  - a single int, the size of the window
- **stride**
  - a single int, the stride of the window. Default value is `kernel_size`
- **ceil_mode**
  - when `TRUE`, will use `ceil` instead of `floor` to compute the output shape

**Details**

\[
f(X) = \sqrt[p]{\sum_{x \in X} x^p}
\]

- At \( p = \infty \), one gets Max Pooling
- At \( p = 1 \), one gets Sum Pooling (which is proportional to Average Pooling)
Shape

- Input: \((N, C, L_{in})\)
- Output: \((N, C, L_{out})\), where

\[
L_{out} = \left\lfloor \frac{L_{in} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor
\]

Note

If the sum to the power of \(p\) is zero, the gradient of this function is not defined. This implementation will set the gradient to zero in this case.

Examples

```r
if (torch_is_installed()) {
  # power-2 pool of window of length 3, with stride 2.
  m <- nn_lp_pool2d(2, 3, stride = 2)
  input <- torch_randn(20, 16, 50)
  output <- m(input)
}
```

---

**nn_lp_pool2d**

Applies a 2D power-average pooling over an input signal composed of several input planes.

Description

On each window, the function computed is:

Usage

```
nn_lp_pool2d(norm_type, kernel_size, stride = NULL, ceil_mode = FALSE)
```

Arguments

- **norm_type**
  - if inf than one gets max pooling if 0 you get sum pooling (proportional to the avg pooling)
- **kernel_size**
  - the size of the window
- **stride**
  - the stride of the window. Default value is kernel\_size
- **ceil_mode**
  - when TRUE, will use ceil instead of floor to compute the output shape
Details

\[ f(X) = \sqrt[p]{\sum_{x \in X} x^p} \]

- At \( p = \infty \), one gets Max Pooling
- At \( p = 1 \), one gets Sum Pooling (which is proportional to average pooling)

The parameters \( \text{kernel\_size}, \text{stride} \) can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

Shape

- Input: \((N, C, H_{\text{in}}, W_{\text{in}})\)
- Output: \((N, C, H_{\text{out}}, W_{\text{out}})\), where

\[
H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor \\
W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor
\]

Note

If the sum to the power of \( p \) is zero, the gradient of this function is not defined. This implementation will set the gradient to zero in this case.

Examples

```r
if (torch_is_installed()) {

# power-2 pool of square window of size=3, stride=2
m <- nn_lp_pool2d(2, 3, stride = 2)
# pool of non-square window of power 1.2
m <- nn_lp_pool2d(1.2, c(3, 2), stride = c(2, 1))
input <- torch_randn(20, 16, 50, 32)
output <- m(input)
}
```
nn_lstm

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

Description
For each element in the input sequence, each layer computes the following function:

Usage

nn_lstm(
    input_size,
    hidden_size,
    num_layers = 1,
    bias = TRUE,
    batch_first = FALSE,
    dropout = 0,
    bidirectional = FALSE,
    ...
)

Arguments

input_size The number of expected features in the input x
hidden_size The number of features in the hidden state h
num_layers Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
bias If FALSE, then the layer does not use bias weights b_ih and b_hh. Default: TRUE
batch_first If TRUE, then the input and output tensors are provided as (batch, seq, feature). Default: FALSE
dropout If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to dropout. Default: 0
bidirectional If TRUE, becomes a bidirectional LSTM. Default: FALSE
... currently unused.

Details

\[
\begin{align*}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_tc_{(t-1)} + i_tg_t \\
h_t &= o_t \tanh(c_t)
\end{align*}
\]
where $h_t$ is the hidden state at time $t$, $c_t$ is the cell state at time $t$, $x_t$ is the input at time $t$, $h_{(t-1)}$ is the hidden state of the previous layer at time $t-1$ or the initial hidden state at time $0$, and $i_t$, $f_t$, $g_t$, $o_t$ are the input, forget, cell, and output gates, respectively. $\sigma$ is the sigmoid function.

**Inputs**

Inputs: input, (h_0, c_0)

- **input** of shape (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See `nn_utils.rnn_pack_padded_sequence()` or `nn_utils.rnn_pack_sequence()` for details.
- **h_0** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch.
- **c_0** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the initial cell state for each element in the batch.

If (h_0, c_0) is not provided, both h_0 and c_0 default to zero.

**Outputs**

Outputs: output, (h_n, c_n)

- **output** of shape (seq_len, batch, num_directions * hidden_size): tensor containing the output features (h_t) from the last layer of the LSTM, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using output_view(c(seq_len, batch, num_directions, hidden_size)), with forward and backward being direction 0 and 1 respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for t = seq_len. Like output, the layers can be separated using h_n_view(c(num_layers, num_directions, batch, hidden_size)) and similarly for c_n.
- **c_n** (num_layers * num_directions, batch, hidden_size): tensor containing the cell state for t = seq_len

**Attributes**

- **weight_ih_l[k]**: the learnable input-hidden weights of the k^{th} layer ($W_{ii}|W_{if}|W_{ig}|W_{io}$), of shape (4*hidden_size x input_size)
- **weight_hh_l[k]**: the learnable hidden-hidden weights of the k^{th} layer ($W_{hi}|W_{hf}|W_{hg}|W_{ho}$), of shape (4*hidden_size x hidden_size)
- **bias_ih_l[k]**: the learnable input-hidden bias of the k^{th} layer ($b_{ii}|b_{if}|b_{ig}|b_{io}$), of shape (4*hidden_size)
- **bias_hh_l[k]**: the learnable hidden-hidden bias of the k^{th} layer ($b_{hi}|b_{hf}|b_{hg}|b_{ho}$), of shape (4*hidden_size)

**Note**

All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden_size}}$
nn_margin_ranking_loss

Examples

```r
if (torch_is_installed()) {
  rnn <- nn_lstm(10, 20, 2)
  input <- torch_randn(5, 3, 10)
  h0 <- torch_randn(2, 3, 20)
  c0 <- torch_randn(2, 3, 20)
  output <- rnn(input, list(h0, c0))
}
```

**Description**

Creates a criterion that measures the loss given inputs \( x_1, x_2 \), two 1D mini-batch Tensors, and a label 1D mini-batch tensor \( y \) (containing 1 or -1). If \( y = 1 \) then it assumed the first input should be ranked higher (have a larger value) than the second input, and vice-versa for \( y = -1 \).

**Usage**

```r
nn_margin_ranking_loss(margin = 0, reduction = "mean")
```

**Arguments**

- **margin** (float, optional): Has a default value of 0.
- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none', 'mean', 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

**Details**

The loss function for each pair of samples in the mini-batch is:

\[
loss(x_1, x_2, y) = \max(0, -y \times (x_1 - x_2) + \text{margin})
\]

**Shape**

- Input1: \((N)\) where \(N\) is the batch size.
- Input2: \((N)\), same shape as the Input1.
- Target: \((N)\), same shape as the inputs.
- Output: scalar. If reduction is 'none', then \((N)\).
Examples

```r
if (torch_is_installed()) {
    loss <- nn_margin_ranking_loss()
    input1 <- torch_randn(3, requires_grad = TRUE)
    input2 <- torch_randn(3, requires_grad = TRUE)
    target <- torch_randn(3)$sign()
    output <- loss(input1, input2, target)
    output$backward()
}
```

### nn_max_pool1d

```r
nn_max_pool1d
```

**MaxPool1D module**

**Description**

Applies a 1D max pooling over an input signal composed of several input planes.

**Usage**

```r
nn_max_pool1d(
    kernel_size,
    stride = NULL,
    padding = 0,
    dilation = 1,
    return_indices = FALSE,
    ceil_mode = FALSE
)
```

**Arguments**

- `kernel_size`: the size of the window to take a max over
- `stride`: the stride of the window. Default value is `kernel_size`
- `padding`: implicit zero padding to be added on both sides
- `dilation`: a parameter that controls the stride of elements in the window
- `return_indices`: if TRUE, will return the max indices along with the outputs. Useful for `nn_max_unpool1d()` later.
- `ceil_mode`: when TRUE, will use ceil instead of floor to compute the output shape

**Details**

In the simplest case, the output value of the layer with input size \((N, C, L)\) and output \((N, C, L_{out})\) can be precisely described as:

\[
out(N_i, C_j, k) = \max_{m=0,\ldots, (kernel_size - 1)} input(N_i, C_j, stride \times k + m)
\]

If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points. dilation controls the spacing between the kernel points. It is harder to describe, but this [link](#) has a nice visualization of what dilation does.
Shape

- Input: \((N, C, L_{in})\)
- Output: \((N, C, L_{out})\), where

\[
L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor
\]

Examples

```r
if (torch_is_installed()) {
  # pool of size=3, stride=2
  m <- nn_max_pool1d(3, stride = 2)
  input <- torch_randn(20, 16, 50)
  output <- m(input)
}
```

---

nn_max_pool2d  
MaxPool2D module

Description

Applies a 2D max pooling over an input signal composed of several input planes.

Usage

```r
nn_max_pool2d(
  kernel_size,
  stride = NULL,
  padding = 0,
  dilation = 1,
  return_indices = FALSE,
  ceil_mode = FALSE
)
```

Arguments

- `kernel_size`: the size of the window to take a max over
- `stride`: the stride of the window. Default value is `kernel_size`
- `padding`: implicit zero padding to be added on both sides
- `dilation`: a parameter that controls the stride of elements in the window
- `return_indices`: if `TRUE`, will return the max indices along with the outputs. Useful for `nn_max_unpool2d()` later.
- `ceil_mode`: when `TRUE`, will use `ceil` instead of `floor` to compute the output shape
Details

In the simplest case, the output value of the layer with input size \((N, C, H, W)\), output \((N, C, H_{out}, W_{out})\) and kernel size \((kH, kW)\) can be precisely described as:

\[
\text{out}(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)
\]

If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points. dilation controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what dilation does.

The parameters kernel_size, stride, padding, dilation can either be:

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

Shape

- Input: \((N, C, H_{in}, W_{in})\)
- Output: \((N, C, H_{out}, W_{out})\), where

\[
H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor
\]

\[
W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor
\]

Examples

```r
if (torch_is_installed()) {
    # pool of square window of size=3, stride=2
    m <- nn_max_pool2d(3, stride = 2)
    # pool of non-square window
    m <- nn_max_pool2d(c(3, 2), stride = c(2, 1))
    input <- torch_randn(20, 16, 50, 32)
    output <- m(input)
}
```

---

**nn_max_pool3d**

Applies a 3D max pooling over an input signal composed of several input planes.

Description

In the simplest case, the output value of the layer with input size \((N, C, D, H, W)\), output \((N, C, D_{out}, H_{out}, W_{out})\) and kernel size \((kD, kH, kW)\) can be precisely described as:
Usage

```
nn_max_pool3d(
    kernel_size,
    stride = NULL,
    padding = 0,
    dilation = 1,
    return_indices = FALSE,
    ceil_mode = FALSE
)
```

Arguments

- `kernel_size`: the size of the window to take a max over
- `stride`: the stride of the window. Default value is `kernel_size`
- `padding`: implicit zero padding to be added on all three sides
- `dilation`: a parameter that controls the stride of elements in the window
- `return_indices`: if TRUE, will return the max indices along with the outputs. Useful for `torch.nn.MaxUnpool3d` later
- `ceil_mode`: when TRUE, will use ceil instead of floor to compute the output shape

Details

\[
\text{out}(N_i, C_j, d, h, w) = \max_{k=0, \ldots, kD-1} \max_{m=0, \ldots, kH-1} \max_{n=0, \ldots, kW-1} \text{input}(N_i, C_j, stride[0] \times d + k, stride[1] \times h + m, stride[2] \times w + n)
\]

If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points. dilation controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what dilation does. The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single int – in which case the same value is used for the depth, height and width dimension
- a tuple of three ints – in which case, the first int is used for the depth dimension, the second int for the height dimension and the third int for the width dimension

Shape

- Input: \((N, C, D_{in}, H_{in}, W_{in})\)
- Output: \((N, C, D_{out}, H_{out}, W_{out})\), where

\[
D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor
\]

\[
H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor
\]

\[
W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor
\]
Examples

```r
if (torch_is_installed()) {
  # pool of square window of size=3, stride=2
  m <- nn_max_pool3d(3, stride = 2)
  # pool of non-square window
  m <- nn_max_pool3d(c(3, 2, 2), stride = c(2, 1, 2))
  input <- torch_randn(20, 16, 50, 44, 31)
  output <- m(input)
}
```

**nn_max_unpool1d**

*Computes a partial inverse of MaxPool1d.*

Description

MaxPool1d is not fully invertible, since the non-maximal values are lost. MaxUnpool1d takes in as input the output of MaxPool1d including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

Usage

```r
nn_max_unpool1d(kernel_size, stride = NULL, padding = 0)
```

Arguments

- `kernel_size` (int or tuple): Size of the max pooling window.
- `stride` (int or tuple): Stride of the max pooling window. It is set to `kernel_size` by default.
- `padding` (int or tuple): Padding that was added to the input

Inputs

- `input`: the input Tensor to invert
- `indices`: the indices given out by `nn_max_pool1d()`
- `output_size` (optional): the targeted output size

Shape

- Input: \((N, C, H_{in})\)
- Output: \((N, C, H_{out})\), where
  \[
  H_{out} = (H_{in} - 1) \times stride[0] - 2 \times padding[0] + kernel_size[0]
  \]
  or as given by `output_size` in the call operator
Note

MaxPool1d can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument output_size in the forward call. See the Inputs and Example below.

Examples

```r
if (torch_is_installed()) {
  pool <- nn_max_pool1d(2, stride = 2, return_indices = TRUE)
  unpool <- nn_max_unpool1d(2, stride = 2)

  input <- torch_tensor(array(1:8 / 1, dim = c(1, 1, 8)))
  out <- pool(input)
  unpool(out[[1]], out[[2]])

  # Example showcasing the use of output_size
  input <- torch_tensor(array(1:8 / 1, dim = c(1, 1, 8)))
  out <- pool(input)
  unpool(out[[1]], out[[2]], output_size = input$size())
  unpool(out[[1]], out[[2]])
}
```

nn_max_unpool2d

Computes a partial inverse of MaxPool2d.

Description

MaxPool2d is not fully invertible, since the non-maximal values are lost. MaxUnpool2d takes in as input the output of MaxPool2d including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

Usage

```
n_max_unpool2d(kernel_size, stride = NULL, padding = 0)
```

Arguments

- `kernel_size` (int or tuple): Size of the max pooling window.
- `stride` (int or tuple): Stride of the max pooling window. It is set to kernel_size by default.
- `padding` (int or tuple): Padding that was added to the input

Inputs

- `input`: the input Tensor to invert
- `indices`: the indices given out by `nn_max_pool2d()`
- `output_size` (optional): the targeted output size
**Shape**

- Input: \((N, C, H_{in}, W_{in})\)
- Output: \((N, C, H_{out}, W_{out})\)

\[
H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel_size}[0]
\]
\[
W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel_size}[1]
\]

or as given by `output_size` in the call operator

**Note**

`MaxPool2d` can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument `output_size` in the forward call. See the Inputs and Example below.

**Examples**

```r
if (torch_is_installed()) {
  pool <- nn_max_pool2d(2, stride = 2, return_indices = TRUE)
  unpool <- nn_max_unpool2d(2, stride = 2)
  input <- torch_randn(1, 1, 4, 4)
  out <- pool(input)
  unpool(out[[1]], out[[2]])

  # specify a different output size than input size
  unpool(out[[1]], out[[2]], output_size = c(1, 1, 5, 5))
}
```

---

**nn_max_unpool3d**

*Computes a partial inverse of MaxPool3d.*

**Description**

`MaxPool3d` is not fully invertible, since the non-maximal values are lost. `MaxUnpool3d` takes in as input the output of `MaxPool3d` including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

**Usage**

```
nn_max_unpool3d(kernel_size, stride = NULL, padding = 0)
```

**Arguments**

- `kernel_size` (int or tuple): Size of the max pooling window.
- `stride` (int or tuple): Stride of the max pooling window. It is set to `kernel_size` by default.
- `padding` (int or tuple): Padding that was added to the input
**Inputs**

- **input**: the input Tensor to invert
- **indices**: the indices given out by `nn_max_pool3d()`
- **output_size** (optional): the targeted output size

**Shape**

- **Input**: \((N, C, D_{in}, H_{in}, W_{in})\)
- **Output**: \((N, C, D_{out}, H_{out}, W_{out})\), where

\[
D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel_size}[0]
\]

\[
H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel_size}[1]
\]

\[
W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{padding}[2] + \text{kernel_size}[2]
\]

or as given by `output_size` in the call operator

**Note**

`MaxPool3d` can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument `output_size` in the forward call. See the Inputs section below.

**Examples**

```r
if (torch_is_installed()) {

# pool of square window of size=3, stride=2
pool <- nn_max_pool3d(3, stride = 2, return_indices = TRUE)
unpool <- nn_max_unpool3d(3, stride = 2)
out <- pool(torch_randn(20, 16, 51, 33, 15))
unpooled_output <- unpool(out[[1]], out[[2]])
unpooled_output$size()
}
```

---

**nn_module**

*Base class for all neural network modules.*

**Description**

Your models should also subclass this class.
Usage

```
nn_module(
    classname = NULL,
    inherit = nn_Module,
    ..., 
    private = NULL,
    active = NULL,
    parent_env = parent.frame()
)
```

Arguments

- **classname**: an optional name for the module
- **inherit**: an optional module to inherit from
- **private**: passed to `R6::R6Class()`. 
- **active**: passed to `R6::R6Class()`. 
- **parent_env**: passed to `R6::R6Class()`.

Details

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes.

You are expected to implement the `initialize` and the `forward` to create a new `nn_module`.

Initialize

The `initialize` function will be called whenever a new instance of the `nn_module` is created. We use the `initialize` functions to define submodules and parameters of the module. For example:

```
initialize = function(input_size, output_size) {
    self$conv1 <- nn_conv2d(input_size, output_size, 5)
    self$conv2 <- nn_conv2d(output_size, output_size, 5)
}
```

The `initialize` function can have any number of parameters. All objects assigned to `self$` will be available for other methods that you implement. Tensors wrapped with `nn_parameter()` or `nn_buffer()` and submodules are automatically tracked when assigned to `self$`.

The `initialize` function is optional if the module you are defining doesn’t have weights, submodules or buffers.

Forward

The `forward` method is called whenever an instance of `nn_module` is called. This is usually used to implement the computation that the module does with the weights ad submodules defined in the `initialize` function.

For example:
forward = function(input) {
  input <- self$conv1(input)
  input <- nnf_relu(input)
  input <- self$conv2(input)
  input <- nnf_relu(input)
  input
}

The forward function can use the self$training attribute to make different computations depending whether the model is training or not, for example if you were implementing the dropout module.

Examples

if (torch_is_installed()) {
  model <- nn_module(
    initialize = function() {
      self$conv1 <- nn_conv2d(1, 20, 5)
      self$conv2 <- nn_conv2d(20, 20, 5)
    },
    forward = function(input) {
      input <- self$conv1(input)
      input <- nnf_relu(input)
      input <- self$conv2(input)
      input <- nnf_relu(input)
      input
    }
  )
}

nn_module_dict

| Container that allows named values |

Description

Container that allows named values

Usage

nn_module_dict(dict)

Arguments

dict A named list of submodules that will be saved in that module.

See Also

nn_module_list()
Examples

```r
if (torch_is_installed()) {
  nn_module <- nn_module(
    initialize = function() {
      self$dict <- nn_module_dict(list(
        l1 = nn_linear(10, 20),
        l2 = nn_linear(20, 10)
      ))
    },
    forward = function(x) {
      x <- self$dict$l1(x)
      self$dict$l2(x)
    }
  )
}
```

**nn_module_list**  
*Holds submodules in a list.*

Description

*nn_module_list* can be indexed like a regular R list, but modules it contains are properly registered, and will be visible by all *nn_module* methods.

Usage

```r
nn_module_list(modules = list())
```

Arguments

- **modules**  
a list of modules to add

See Also

- *nn_module_dict()*

Examples

```r
if (torch_is_installed()) {
  my_module <- nn_module(
    initialize = function() {
      self$linears <- nn_module_list(lapply(1:10, function(x) nn_linear(10, 10)))
    },
    forward = function(x) {
      for (i in 1:length(self$linears)) {
        x <- self$linears[[i]](x)
      }
      x
    }
  )
```
**nn_mse_loss**

nn_mse_loss


\[ \text{MSE loss} \]

**Description**

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input \( x \) and target \( y \). The unreduced (i.e. with \texttt{reduction} set to 'none') loss can be described as:

\[
\ell(x, y) = \sum_{n=1}^{N} (x_n - y_n)^2,
\]

where \( N \) is the batch size. If \texttt{reduction} is not 'none' (default 'mean'), then:

\[
\ell(x, y) = \begin{cases} 
\frac{1}{N} \sum_{n=1}^{N} (x_n - y_n)^2, & \text{if reduction = 'mean';} \\
\sum_{n=1}^{N} (x_n - y_n)^2, & \text{if reduction = 'sum'.}
\end{cases}
\]

\( x \) and \( y \) are tensors of arbitrary shapes with a total of \( n \) elements each.

The mean operation still operates over all the elements, and divides by \( n \). The division by \( n \) can be avoided if one sets \texttt{reduction} = 'sum'.

**Usage**

```python
nn_mse_loss(reduction = "mean")
```

**Arguments**

- **reduction**: (string, optional): Specifies the reduction to apply to the output: 'none' \| 'mean' \| 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

**Details**

\[
\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \quad l_n = (x_n - y_n)^2,
\]

**Shape**

- Input: \((N, *)\) where \(*\) means, any number of additional dimensions
- Target: \((N, *)\), same shape as the input
nn_multihead_attention

Examples

```r
if (torch_is_installed()) {
  loss <- nn_mse_loss()
  input <- torch_randn(3, 5, requires_grad = TRUE)
  target <- torch_randn(3, 5)
  output <- loss(input, target)
  output$backward()
}
```

Description

Allows the model to jointly attend to information from different representation subspaces. See reference: Attention Is All You Need

Usage

```r
nn_multihead_attention(
  embed_dim,
  num_heads,
  dropout = 0,
  bias = TRUE,
  add_bias_kv = FALSE,
  add_zero_attn = FALSE,
  kdim = NULL,
  vdim = NULL,
  batch_first = FALSE
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>embed_dim</td>
<td>total dimension of the model.</td>
</tr>
<tr>
<td>num_heads</td>
<td>parallel attention heads. Note that <code>embed_dim</code> will be split across <code>num_heads</code> (i.e. each head will have dimension <code>embed_dim %/% num_heads</code>).</td>
</tr>
<tr>
<td>dropout</td>
<td>a Dropout layer on <code>attn_output_weights</code>. Default: 0.0.</td>
</tr>
<tr>
<td>bias</td>
<td>add bias as module parameter. Default: True.</td>
</tr>
<tr>
<td>add_bias_kv</td>
<td>add bias to the key and value sequences at dim=0.</td>
</tr>
<tr>
<td>add_zero_attn</td>
<td>add a new batch of zeros to the key and value sequences at dim=1.</td>
</tr>
<tr>
<td>kdim</td>
<td>total number of features in key. Default: NULL</td>
</tr>
<tr>
<td>vdim</td>
<td>total number of features in value. Default: NULL. Note: if <code>kdim</code> and <code>vdim</code> are <code>NULL</code>, they will be set to <code>embed_dim</code> such that query, key, and value have the same number of features.</td>
</tr>
</tbody>
</table>
**batch_first** if TRUE then the input and output tensors are \((N, S, E)\) instead of \((S, N, E)\), where \(N\) is the batch size, \(S\) is the sequence length, and \(E\) is the embedding dimension.

**Details**

\[
\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \text{where} \text{head}_i = \text{Attention}(QW^Q, KW^K, VW^V)
\]

**Shape**

**Inputs:**

- query: \((L, N, E)\) where \(L\) is the target sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. (but see the batch_first argument)
- key: \((S, N, E)\), where \(S\) is the source sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. (but see the batch_first argument)
- value: \((S, N, E)\) where \(S\) is the source sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. (but see the batch_first argument)
- key_padding_mask: \((N, S)\) where \(N\) is the batch size, \(S\) is the source sequence length. If a ByteTensor is provided, the non-zero positions will be ignored while the position with the zero positions will be unchanged. If a BoolTensor is provided, the positions with the value of True will be ignored while the position with the value of False will be unchanged.
- attn_mask: 2D mask \((L, S)\) where \(L\) is the target sequence length, \(S\) is the source sequence length. 3D mask \((N \times \text{numheads}, L, S)\) where \(N\) is the batch size, \(L\) is the target sequence length, \(S\) is the source sequence length. attn_mask ensure that position \(i\) is allowed to attend the unmasked positions. If a ByteTensor is provided, the non-zero positions are not allowed to attend while the zero positions will be unchanged. If a BoolTensor is provided, positions with True are not allowed to attend while False values will be unchanged. If a FloatTensor is provided, it will be added to the attention weight.

**Outputs:**

- attn_output: \((L, N, E)\) where \(L\) is the target sequence length, \(N\) is the batch size, \(E\) is the embedding dimension. (but see the batch_first argument)
- attn_output_weights:
  - if avg_weights is TRUE (the default), the output attention weights are averaged over the attention heads, giving a tensor of shape \((N, L, S)\) where \(N\) is the batch size, \(L\) is the target sequence length, \(S\) is the source sequence length.
  - if avg_weights is FALSE, the attention weight tensor is output as-is, with shape \((N, H, L, S)\), where \(H\) is the number of attention heads.

**Examples**

```r
if (torch_is_installd()) {
  # Not run:
  multihead_attn <- nn_multihead_attention(embed_dim, num_heads)
  out <- multihead_attn(query, key, value)
```
## nn_multilabel_margin_loss

**Multilabel margin loss**

### Description

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input \( x \) (a 2D mini-batch Tensor) and output \( y \) (which is a 2D Tensor of target class indices). For each sample in the mini-batch:

### Usage

```r
nn_multilabel_margin_loss(reduction = "mean")
```

### Arguments

- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

### Details

\[
loss(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{x.size(0)}
\]

where \( x \in \{0, \cdots, x.size(0) - 1\}, \ y \in \{0, \cdots, y.size(0) - 1\}, \ 0 \leq y[j] \leq x.size(0) - 1, \ \text{and} \ i \neq y[j] \ \text{for all} \ i \ \text{and} \ j. \ y \ \text{and} \ x \ \text{must have the same size.}

The criterion only considers a contiguous block of non-negative targets that starts at the front. This allows for different samples to have variable amounts of target classes.

### Shape

- Input: \((C)\) or \((N, C)\) where \(N\) is the batch size and \(C\) is the number of classes.
- Target: \((C)\) or \((N, C)\), label targets padded by -1 ensuring same shape as the input.
- Output: scalar. If reduction is 'none', then \((N)\).
Examples

```r
if (torch_is_installed()) {
  loss <- nn_multilabel_margin_loss()
  x <- torch_tensor(c(0.1, 0.2, 0.4, 0.8))$view(c(1, 4))
  # for target y, only consider labels 4 and 1, not after label -1
  y <- torch_tensor(c(4, 1, -1, 2), dtype = torch_long())$view(c(1, 4))
  loss(x, y)
}
```

### nn_multilabel_soft_margin_loss

*Multi label soft margin loss*

**Description**

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input \( x \) and target \( y \) of size \((N, C)\).

**Usage**

```r
nn_multilabel_soft_margin_loss(weight = NULL, reduction = "mean")
```

**Arguments**

- `weight` (Tensor, optional): a manual rescaling weight given to each class. If given, it has to be a Tensor of size \( C \). Otherwise, it is treated as if having all ones.
- `reduction` (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

**Details**

For each sample in the minibatch:

\[
loss(x, y) = -\frac{1}{C} \sum_i y[i] \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) \log \left( \frac{\exp(-x[i])}{(1 + \exp(-x[i]))} \right)
\]

where \( i \in \{0, \cdots, x.nElement() - 1\} \), \( y[i] \in \{0, 1\} \).

**Shape**

- Input: \((N, C)\) where \( N \) is the batch size and \( C \) is the number of classes.
- Target: \((N, C)\), label targets padded by -1 ensuring same shape as the input.
- Output: scalar. If `reduction` is 'none', then \((N)\).
**nn_multi_margin_loss**  
*Multi margin loss*

**Description**

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input \( x \) (a 2D mini-batch Tensor) and output \( y \) (which is a 1D tensor of target class indices, \( 0 \leq y \leq x\text{.size}(1) - 1 \)):

**Usage**

\[
\text{nn_multi_margin_loss}(p = 1, \text{margin} = 1, \text{weight} = \text{NULL}, \text{reduction} = \text{"mean"})
\]

**Arguments**

- \( p \) (int, optional): Has a default value of 1. 1 and 2 are the only supported values.
- \( \text{margin} \) (float, optional): Has a default value of 1.
- \( \text{weight} \) (Tensor, optional): a manual rescaling weight given to each class. If given, it has to be a Tensor of size \( C \). Otherwise, it is treated as if having all ones.
- \( \text{reduction} \) (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

**Details**

For each mini-batch sample, the loss in terms of the 1D input \( x \) and scalar output \( y \) is:

\[
\text{loss}(x, y) = \frac{\sum_i \max(0, \text{margin} - x[y] + x[i])^p}{x\text{.size}(0)}
\]

where \( x \in \{0, \ldots, x\text{.size}(0) - 1\} \) and \( i \neq y \).

Optionally, you can give non-equal weighting on the classes by passing a 1D weight tensor into the constructor. The loss function then becomes:

\[
\text{loss}(x, y) = \frac{\sum_i \max(0, w[y] * (\text{margin} - x[y] + x[i])^p)}{x\text{.size}(0)}
\]
**nn_nll_loss**

Description

The negative log likelihood loss. It is useful to train a classification problem with C classes.

Usage

```python
nn_nll_loss(weight = NULL, ignore_index = -100, reduction = "mean")
```

Arguments

- **weight** (Tensor, optional): a manual rescaling weight given to each class. If given, it has to be a Tensor of size C. Otherwise, it is treated as if having all ones.
- **ignore_index** (int, optional): Specifies a target value that is ignored and does not contribute to the input gradient.
- **reduction** (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the weighted mean of the output is taken, 'sum': the output will be summed.

Details

If provided, the optional argument `weight` should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The input given through a forward call is expected to contain log-probabilities of each class. Input has to be a Tensor of size either `(minibatch, C)` or `(minibatch, C, d1, d2, ..., dK)` with $K \geq 1$ for the K-dimensional case (described later).

Obtaining log-probabilities in a neural network is easily achieved by adding a LogSoftmax layer in the last layer of your network.

You may use CrossEntropyLoss instead, if you prefer not to add an extra layer.

The target that this loss expects should be a class index in the range $[0, C-1]$ where $C = \text{number of classes}$; if `ignore_index` is specified, this loss also accepts this class index (this index may not necessarily be in the class range).

The unreduced (i.e. with `reduction` set to 'none') loss can be described as:

$$
\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \quad l_n = -w_{yn}x_n, w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore_index}\},
$$

where $x$ is the input, $y$ is the target, $w$ is the weight, and $N$ is the batch size. If `reduction` is not 'none' (default 'mean'), then

$$
\ell(x, y) = \begin{cases} 
\sum_{n=1}^{N} \frac{l_n}{\sum_{n=1}^{N} w_{yn}} & \text{if } \text{reduction} = 'mean'; \\
\sum_{n=1}^{N} l_n & \text{if } \text{reduction} = 'sum'.
\end{cases}
$$
Can also be used for higher dimension inputs, such as 2D images, by providing an input of size \((\text{minibatch}, C, d_1, d_2, ..., d_K)\) with \(K \geq 1\), where \(K\) is the number of dimensions, and a target of appropriate shape (see below). In the case of images, it computes NLL loss per-pixel.

**Shape**

- Input: \((N, C)\) where \(C\) = number of classes, or \((N, d_1, d_2, ..., d_K)\) with \(K \geq 1\) in the case of \(K\)-dimensional loss.

- Target: \((N)\) where each value is \(0 \leq \text{targets}[i] \leq C - 1\), or \((N, d_1, d_2, ..., d_K)\) with \(K \geq 1\) in the case of \(K\)-dimensional loss.

- Output: scalar.

If reduction is 'none', then the same size as the target: \((N)\), or \((N, d_1, d_2, ..., d_K)\) with \(K \geq 1\) in the case of \(K\)-dimensional loss.

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_log_softmax(dim = 2)
  loss <- nn_nll_loss()
  # input is of size N x C = 3 x 5
  input <- torch_randn(3, 5, requires_grad = TRUE)
  # each element in target has to have 0 <= value < C
  target <- torch_tensor(c(2, 1, 5), dtype = torch_long())
  output <- loss(m(input), target)
  output$backward()

  # 2D loss example (used, for example, with image inputs)
  N <- 5
  C <- 4
  loss <- nn_nll_loss()
  # input is of size N x C x height x width
  data <- torch_randn(N, 16, 10, 10)
  conv <- nn_conv2d(16, C, c(3, 3))
  m <- nn_log_softmax(dim = 1)
  # each element in target has to have 0 <= value < C
  target <- torch_empty(N, 8, 8, dtype = torch_long())$random_(1, C)
  output <- loss(m(conv(data)), target)
  output$backward()
}
```

---

**Description**

Computes the batchwise pairwise distance between vectors \(v_1, v_2\) using the p-norm:
**Usage**

\[ \text{nn_pairwise_distance}(p = 2, \text{eps} = 1e-06, \text{keepdim} = \text{FALSE}) \]

**Arguments**

- **p** (real): the norm degree. Default: 2
- **eps** (float, optional): Small value to avoid division by zero. Default: 1e-6
- **keepdim** (bool, optional): Determines whether or not to keep the vector dimension. Default: FALSE

**Details**

\[ \|x\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p} \]

**Shape**

- Input1: \((N, D)\) where \(D = \text{vector dimension}\)
- Input2: \((N, D)\), same shape as the Input1
- Output: \((N)\). If \(\text{keepdim}\) is TRUE, then \((N, 1)\).

**Examples**

```r
if (torch_is_installed()) {
  pdist <- nn_pairwise_distance(p = 2)
  input1 <- torch_randn(100, 128)
  input2 <- torch_randn(100, 128)
  output <- pdist(input1, input2)
}
```

---

**nn_parameter**

*Creates an nn_parameter*

**Description**

Indicates to \text{nn_module} that \(x\) is a parameter

**Usage**

\[ \text{nn_parameter}(x, \text{requires_grad} = \text{TRUE}) \]

**Arguments**

- **x** the tensor that you want to indicate as parameter
- **requires_grad** whether this parameter should have \text{requires_grad} = TRUE
nn_poisson_nll_loss  Poisson NLL loss

Description

Negative log likelihood loss with Poisson distribution of target. The loss can be described as:

Usage

```
n_poisson_nll_loss(
    log_input = TRUE,
    full = FALSE,
    eps = 1e-08,
    reduction = "mean"
)
```

Arguments

- **log_input**: (bool, optional): if TRUE the loss is computed as \(\exp(\text{input}) - \text{target} \times \text{input}\), if FALSE the loss is \(\text{input} - \text{target} \times \log(\text{input} + \text{eps})\).
- **full**: (bool, optional): whether to compute full loss, i.e. to add the Stirling approximation term \(\text{target} \times \log(\text{target}) - \text{target} + 0.5 \times \log(2\pi\text{target})\).
- **eps**: (float, optional): Small value to avoid evaluation of \(\log(0)\) when log_input = FALSE. Default: 1e-8
- **reduction**: (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

\[
\text{target} \sim \text{Poisson}(\text{input}) \Rightarrow \text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} \times \log(\text{input}) + \log(\text{target}!)
\]

The last term can be omitted or approximated with Stirling formula. The approximation is used for target values more than 1. For targets less or equal to 1 zeros are added to the loss.

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Target: \((N, *)\), same shape as the input
- Output: scalar by default. If reduction is 'none', then \((N, *)\), the same shape as the input
Examples

```r
if (torch_is_installed()) {
    loss <- nn_poisson_nll_loss()
    log_input <- torch_randn(5, 2, requires_grad = TRUE)
    target <- torch_randn(5, 2)
    output <- loss(log_input, target)
    output$backward()
}
```

---

**nn_prelu**  
*PReLU module*

**Description**

Applies the element-wise function:

$$\text{PReLU}(x) = \max(0, x) + a \times \min(0, x)$$

or

$$\begin{cases} 
    x, & \text{if } x \geq 0 \\
    ax, & \text{otherwise}
\end{cases}$$

**Usage**

```r
nn_prelu(num_parameters = 1, init = 0.25)
```

**Arguments**

- **num_parameters** (int): number of $a$ to learn. Although it takes an int as input, there is only two values are legitimate: 1, or the number of channels at input. Default: 1
- **init** (float): the initial value of $a$. Default: 0.25

**Details**

Here $a$ is a learnable parameter. When called without arguments, `nn.prelu()` uses a single parameter $a$ across all input channels. If called with `nn_prelu(nChannels)`, a separate $a$ is used for each input channel.

**Shape**

- Input: $(N, \ast)$ where $\ast$ means, any number of additional dimensions
- Output: $(N, \ast)$, same shape as the input

**Attributes**

- `weight` (Tensor): the learnable weights of shape (num_parameters).
Note

weight decay should not be used when learning $a$ for good performance.

Channel dim is the 2nd dim of input. When input has dims < 2, then there is no channel dim and
the number of channels = 1.

Examples

```r
if (torch_is_installed()) {
  m <- nn_prelu()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

**nn_prune_head**

*Prune top layer(s) of a network*

**Description**

Prune head_size last layers of a nn_module in order to replace them by your own head, or in order
to use the pruned module as a sequential embedding module.

**Usage**

```
nn_prune_head(x, head_size)
```

**Arguments**

- `x` : nn_network to prune
- `head_size` : number of nn_layers to prune

**Value**

a nn_sequential network with the top nn_layer removed

**Examples**

```r
if (torch_is_installed()) {
  x <- nn_sequential(
    nn_relu(),
    nn_tanh(),
    nn_relu6(),
    nn_relu(),
    nn_linear(2,10),
    nn_batch_norm1d(10),
    nn_tanh(),
    nn_linear(10,3)
  )
}
```
nn_relu

```r
prune <- nn_prune_head(x, 3)
prune
}
}
```

---

### nn_relu

**ReLU module**

**Description**

Applies the rectified linear unit function element-wise

\[
\text{ReLU}(x) = (x)^+ = \max(0, x)
\]

**Usage**

```r
nn_relu(inplace = FALSE)
```

**Arguments**

- `inplace` can optionally do the operation in-place. Default: FALSE

**Shape**

- Input: \((N, \ast)\) where \(\ast\) means, any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_relu()
  input <- torch_randn(2)
  m(input)
}
```

---

### nn_relu6

**ReLU6 module**

**Description**

Applies the element-wise function:

**Usage**

```r
nn_relu6(inplace = FALSE)
```
Arguments

inplace can optionally do the operation in-place. Default: FALSE

Details

ReLU6(x) = min(max(0, x), 6)

Shape

• Input: (N, *) where * means, any number of additional dimensions
• Output: (N, *), same shape as the input

Examples

if (torch_is_installed()) {
  m <- nn_relu6()
  input <- torch_randn(2)
  output <- m(input)
}

nn_rnn

RNN module

Description

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

Usage

nn_rnn(
  input_size,
  hidden_size,
  num_layers = 1,
  nonlinearity = NULL,
  bias = TRUE,
  batch_first = FALSE,
  dropout = 0,
  bidirectional = FALSE,
  ...
)
**Arguments**

- **input_size**: The number of expected features in the input \( x \)
- **hidden_size**: The number of features in the hidden state \( h \)
- **num_layers**: Number of recurrent layers. E.g., setting `num_layers`=2 would mean stacking two RNNs together to form a stacked RNN, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity**: The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
- **bias**: If `FALSE`, then the layer does not use bias weights \( b_{ih} \) and \( b_{hh} \). Default: `TRUE`
- **batch_first**: If `TRUE`, then the input and output tensors are provided as `(batch, seq, feature)`. Default: `FALSE`
- **dropout**: If non-zero, introduces a Dropout layer on the outputs of each RNN layer except the last layer, with dropout probability equal to dropout. Default: 0
- **bidirectional**: If `TRUE`, becomes a bidirectional RNN. Default: `FALSE`
- ... other arguments that can be passed to the super class.

**Details**

For each element in the input sequence, each layer computes the following function:

\[
h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})
\]

where \( h_t \) is the hidden state at time \( t \), \( x_t \) is the input at time \( t \), and \( h_{(t-1)} \) is the hidden state of the previous layer at time \( t-1 \) or the initial hidden state at time 0. If `nonlinearity` is 'relu', then ReLU is used instead of tanh.

**Inputs**

- **input** of shape `(seq_len, batch, input_size)`: tensor containing the features of the input sequence. The input can also be a packed variable length sequence.
- **h_0** of shape `(num_layers * num_directions, batch, hidden_size)`: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, `num_directions` should be 2, else it should be 1.

**Outputs**

- **output** of shape `(seq_len, batch, num_directions * hidden_size)`: tensor containing the output features \( (h_t) \) from the last layer of the RNN, for each \( t \). If a :class:`nn_packed_sequence` has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using `output$view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction 0 and 1 respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape `(num_layers * num_directions, batch, hidden_size)`: tensor containing the hidden state for \( t = seq_len \). Like `output`, the layers can be separated using `h_n$view(num_layers, num_directions, batch, hidden_size)`. 

*nn_rnn* 271
Shape

- Input1: \((L, N, H_{in})\) tensor containing input features where \(H_{in} = \text{input\_size}\) and \(L\) represents a sequence length.
- Input2: \((S, N, H_{out})\) tensor containing the initial hidden state for each element in the batch. \(H_{out} = \text{hidden\_size}\) Defaults to zero if not provided. where \(S = \text{num\_layers} \times \text{num\_directions}\) If the RNN is bidirectional, num_directions should be 2, else it should be 1.
- Output1: \((L, N, H_{all})\) where \(H_{all} = \text{num\_directions} \times \text{hidden\_size}\)
- Output2: \((S, N, H_{out})\) tensor containing the next hidden state for each element in the batch

Attributes

- \text{weight\_ih\_l[k]}: the learnable input-hidden weights of the \(k\)-th layer, of shape \((\text{hidden\_size}, \text{input\_size})\) for \(k = 0\). Otherwise, the shape is \((\text{hidden\_size}, \text{num\_directions} \times \text{hidden\_size})\)
- \text{weight\_hh\_l[k]}: the learnable hidden-hidden weights of the \(k\)-th layer, of shape \((\text{hidden\_size}, \text{hidden\_size})\)
- \text{bias\_ih\_l[k]}: the learnable input-hidden bias of the \(k\)-th layer, of shape \((\text{hidden\_size})\)
- \text{bias\_hh\_l[k]}: the learnable hidden-hidden bias of the \(k\)-th layer, of shape \((\text{hidden\_size})\)

Note

All the weights and biases are initialized from \(\mathcal{U}(-\sqrt{k}, \sqrt{k})\) where \(k = \frac{1}{\text{hidden\_size}}\)

Examples

```r
if (torch_is_installed()) {
  rnn <- nn_rnn(10, 20, 2)
  input <- torch_randn(5, 3, 10)
  h0 <- torch_randn(2, 3, 20)
  rnn(input, h0)
}
```

---

### nn_rrelu

**RReLU module**

**Description**
Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:

**Usage**

```r
nn_rrelu(lower = 1/8, upper = 1/3, inplace = FALSE)
```

**Arguments**

- **lower**: lower bound of the uniform distribution. Default: \(\frac{1}{8}\)
- **upper**: upper bound of the uniform distribution. Default: \(\frac{1}{3}\)
- **inplace**: can optionally do the operation in-place. Default: \(\text{FALSE}\)
**Details**
The function is defined as:

\[
RReLU(x) = \begin{cases} 
  x & \text{if } x \geq 0 \\
  ax & \text{otherwise}
\end{cases}
\]

where \( a \) is randomly sampled from uniform distribution \( U(\text{lower}, \text{upper}) \). See: https://arxiv.org/pdf/1505.00853.pdf

**Shape**
- Input: \((N, *)\) where \(*\) means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

**Examples**
```r
if (torch_is_installed()) {
  m <- nn_rrelu(0.1, 0.3)
  input <- torch_randn(2)
  m(input)
}
```

---

**nn_selu**  
**SELU module**  

**Description**
Applied element-wise, as:

**Usage**

```r
nn_selu(inplace = FALSE)
```

**Arguments**

- `inplace` (bool, optional): can optionally do the operation in-place. Default: FALSE

**Details**

\[
SELU(x) = \text{scale} \times (\max(0, x) + \min(0, \alpha \times (\exp(x) - 1)))
\]

with \( \alpha = 1.6732632423543772848170429916717 \) and \( \text{scale} = 1.0507009873554804934193349852946 \).

More details can be found in the paper **Self-Normalizing Neural Networks**.
**Shape**

- Input: \((N, *)\) where \(*\) means, any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

**Examples**

```r
if (torch_is_installed()) {
    m <- nn_selu()
    input <- torch_randn(2)
    output <- m(input)
}
```

---

**nn_sequential**  
*A sequential container*

**Description**

A sequential container. Modules will be added to it in the order they are passed in the constructor. See examples.

**Usage**

`nn_sequential(...)`

**Arguments**

...  
sequence of modules to be added

**Examples**

```r
if (torch_is_installed()) {
    model <- nn_sequential(
        nn_conv2d(1, 20, 5),
        nn_relu(),
        nn_conv2d(20, 64, 5),
        nn_relu()
    )
    input <- torch_randn(32, 1, 28, 28)
    output <- model(input)
}
```
**nn_sigmoid**

*Description*

Applies the element-wise function:

\[
\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}
\]

**Usage**

`nn_sigmoid()`

**Details**

\[
\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}
\]

**Shape**

- Input: \((N, *)\) where \(*\) means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_sigmoid()
  input <- torch_randn(2)
  output <- m(input)
}
```

**nn_silu**

*Applies the Sigmoid Linear Unit (SiLU) function, element-wise. The SiLU function is also known as the swish function.*

*Description*

Applies the Sigmoid Linear Unit (SiLU) function, element-wise. The SiLU function is also known as the swish function.

**Usage**

`nn_silu(inplace = FALSE)`

**Arguments**

- `inplace` can optionally do the operation in-place. Default: FALSE
Details

See Gaussian Error Linear Units (GELUs) where the SiLU (Sigmoid Linear Unit) was originally coined, and see Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning and Swish: a Self-Gated Activation Function where the SiLU was experimented with later.

nn_smooth_l1_loss Smooth L1 loss

Description

Creates a criterion that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise. It is less sensitive to outliers than the MSELoss and in some cases prevents exploding gradients (e.g. see Fast R-CNN paper by Ross Girshick). Also known as the Huber loss:

Usage

nn_smooth_l1_loss(reduction = "mean")

Arguments

reduction (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

\[
loss(x, y) = \frac{1}{n} \sum_{i} z_i
\]

where \( z_i \) is given by:

\[
z_i = \begin{cases} 
0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\
|x_i - y_i| - 0.5, & \text{otherwise}
\end{cases}
\]

\( x \) and \( y \) arbitrary shapes with a total of \( n \) elements each the sum operation still operates over all the elements, and divides by \( n \). The division by \( n \) can be avoided if sets reduction = 'sum'.

Shape

- Input: \((N, *)\) where \( * \) means, any number of additional dimensions
- Target: \((N, *)\), same shape as the input
- Output: scalar. If reduction is 'none', then \((N, *)\), same shape as the input
nn_softmax

Softmax module

Description

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range $[0, 1]$ and sum to 1. Softmax is defined as:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

When the input Tensor is a sparse tensor then the unspecified values are treated as $-\infty$.

Value

: a Tensor of the same dimension and shape as the input with values in the range $[0, 1]$

Shape

- Input: (*) where * means, any number of additional dimensions
- Output: (*), same shape as the input

Note

This module doesn’t work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use LogSoftmax instead (it’s faster and has better numerical properties).

Examples

```python
if (torch_is_installed()) {
  m <- nn_softmax(1)
  input <- torch_randn(2, 3)
  output <- m(input)
}
```
**nn_softmax2d**  
*Softmax2d module*

**Description**
Applies SoftMax over features to each spatial location. When given an image of Channels x Height x Width, it will apply Softmax to each location \((Channels, h_i, w_j)\).

**Usage**
```
nn_softmax2d()
```

**Value**
a Tensor of the same dimension and shape as the input with values in the range \([0, 1]\)

**Shape**
- Input: \((N, C, H, W)\)
- Output: \((N, C, H, W)\) (same shape as input)

**Examples**
```
if (torch_is_installed()) {
  m <- nn_softmax2d()
  input <- torch_randn(2, 3, 12, 13)
  output <- m(input)
}
```

---

**nn_softmin**  
*Softmin*

**Description**
Applies the Softmin function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range \([0, 1]\) and sum to 1. Softmin is defined as:

**Usage**
```
nn_softmin(dim)
```

**Arguments**
- `dim` (int): A dimension along which Softmin will be computed (so every slice along `dim` will sum to 1).
Details

\[
\text{Softmin}(x_i) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}
\]

Value

A Tensor of the same dimension and shape as the input, with values in the range \([0, 1]\).

Shape

- Input: (*) where * means, any number of additional dimensions
- Output: (*), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_softmin(dim = 1)
  input <- torch_randn(2, 2)
  output <- m(input)
}
```

---

### nn_softplus

**Softplus module**

**Description**

Applies the element-wise function:

\[
\text{Softplus}(x) = \frac{1}{\beta} \log(1 + \exp(\beta \times x))
\]

**Usage**

`nn_softplus(beta = 1, threshold = 20)`

**Arguments**

- `beta`: the \(\beta\) value for the Softplus formulation. Default: 1
- `threshold`: values above this revert to a linear function. Default: 20

**Details**

SoftPlus is a smooth approximation to the ReLU function and can be used to constrain the output of a machine to always be positive. For numerical stability the implementation reverts to the linear function when \(\text{input} \times \beta > \text{threshold}\).
Shape

- Input: \((N, \ast)\) where \(*\) means, any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_softplus()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

**nn_softshrink**

**Softshrink module**

Description

Applies the soft shrinkage function elementwise:

Usage

```r
nn_softshrink(lambda = 0.5)
```

Arguments

- `lambda` the \(\lambda\) (must be no less than zero) value for the Softshrink formulation. Default: 0.5

Details

\[
\text{SoftShrinkage}(x) = \begin{cases} 
  x - \lambda, & \text{if } x > \lambda \\
  x + \lambda, & \text{if } x < -\lambda \\
  0, & \text{otherwise}
\end{cases}
\]

Shape

- Input: \((N, \ast)\) where \(*\) means, any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_softshrink()
  input <- torch_randn(2)
  output <- m(input)
}
```
nn_softsign  

**Softsign module**

**Description**
Applies the element-wise function:

\[
\text{SoftSign}(x) = \frac{x}{1 + |x|}
\]

**Usage**

`nn_softsign()`

**Shape**
- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

**Examples**

```r
if (torch_is_installed()) {
  m <- nn_softsign()
  input <- torch_randn(2)
  output <- m(input)
}
```

**nn_soft_margin_loss**  

**Soft margin loss**

**Description**
Creates a criterion that optimizes a two-class classification logistic loss between input tensor \(x\) and target tensor \(y\) (containing 1 or -1).

**Usage**

`nn_soft_margin_loss(reduction = "mean")`

**Arguments**
- `reduction` (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.
Details

\[ \text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] \cdot x[i]))}{x\text{-nelement}} \]

Shape

- Input: (*) where * means, any number of additional dimensions
- Target: (*), same shape as the input
- Output: scalar. If reduction is 'none', then same shape as the input

---

nn_tanh  

Tanh module

Description

Applies the element-wise function:

Usage

\[ \text{nn_tanh}() \]

Details

\[ \text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \]

Shape

- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_tanh()
  input <- torch_randn(2)
  output <- m(input)
}
```
nn_tanhshrink

Description
Applies the element-wise function:

Usage
nn_tanhshrink()

Details
\[ \text{Tanhshrink}(x) = x - \tanh(x) \]

Shape
- Input: \((N, *)\) where * means, any number of additional dimensions
- Output: \((N, *)\), same shape as the input

Examples
```r
if (torch_is_installed()) {
  m <- nn_tanhshrink()
  input <- torch_randn(2)
  output <- m(input)
}
```

nn_threshold

Threshold module

Description
Thresholds each element of the input Tensor.

Usage
nn_threshold(threshold, value, inplace = FALSE)

Arguments
- **threshold** The value to threshold at
- **value** The value to replace with
- **inplace** can optionally do the operation in-place. Default: FALSE
Details

Threshold is defined as:

\[ y = \begin{cases} 
  x, & \text{if } x > \text{threshold value}, \\
  \text{value}, & \text{otherwise} 
\end{cases} \]

Shape

- Input: \((N, \ast)\) where \(\ast\) means, any number of additional dimensions
- Output: \((N, \ast)\), same shape as the input

Examples

```r
if (torch_is_installed()) {
  m <- nn_threshold(0.1, 20)
  input <- torch_randn(2)
  output <- m(input)
}
```

nn_triplet_margin_loss

Triplet margin loss

Description

Creates a criterion that measures the triplet loss given an input tensors \(x_1, x_2, x_3\) and a margin with a value greater than 0. This is used for measuring a relative similarity between samples. A triplet is composed by a, p and n (i.e., anchor, positive examples and negative examples respectively). The shapes of all input tensors should be \((N, D)\).

Usage

```r
nn_triplet_margin_loss(
  margin = 1,
  p = 2,
  eps = 1e-06,
  swap = FALSE,
  reduction = "mean"
)
```

Arguments

- **margin** (float, optional): Default: 1.
- **p** (int, optional): The norm degree for pairwise distance. Default: 2.
- **eps** constant to avoid NaN’s
- **swap** (bool, optional): The distance swap is described in detail in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al. Default: FALSE.
nn_triplet_margin_with_distance_loss

reduction (string, optional): Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed.

Details

The distance swap is described in detail in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al.

The loss function for each sample in the mini-batch is:

\[ L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\} \]

where

\[ d(x_i, y_i) = |x_i - y_i|_p \]

See also nn_triplet_margin_with_distance_loss(), which computes the triplet margin loss for input tensors using a custom distance function.

Shape

- Input: \((N, D)\) where \(D\) is the vector dimension.
- Output: A Tensor of shape \((N)\) if reduction is 'none', or a scalar otherwise.

Examples

```r
if (torch_is_installed()) {
  triplet_loss <- nn_triplet_margin_loss(margin = 1, p = 2)
  anchor <- torch_randn(100, 128, requires_grad = TRUE)
  positive <- torch_randn(100, 128, requires_grad = TRUE)
  negative <- torch_randn(100, 128, requires_grad = TRUE)
  output <- triplet_loss(anchor, positive, negative)
  output$backward()
}
```

Description

Creates a criterion that measures the triplet loss given input tensors \(a, p,\) and \(n\) (representing anchor, positive, and negative examples, respectively), and a nonnegative, real-valued function ("distance function") used to compute the relationship between the anchor and positive example ("positive distance") and the anchor and negative example ("negative distance").
Usage

```
nn_triplet_margin_with_distance_loss(
    distance_function = NULL,
    margin = 1,
    swap = FALSE,
    reduction = "mean"
)
```

Arguments

distance_function (callable, optional): A nonnegative, real-valued function that quantifies the closeness of two tensors. If not specified, `nn_pairwise_distance()` will be used. Default: None

margin (float, optional): A non-negative margin representing the minimum difference between the positive and negative distances required for the loss to be 0. Larger margins penalize cases where the negative examples are not distant enough from the anchors, relative to the positives. Default: 1.

swap (bool, optional): Whether to use the distance swap described in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al. If TRUE, and if the positive example is closer to the negative example than the anchor is, swaps the positive example and the anchor in the loss computation. Default: FALSE.

reduction (string, optional): Specifies the (optional) reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Default: 'mean'

Details

The unreduced loss (i.e., with reduction set to 'none') can be described as:

\[
\ell(a, p, n) = \mathbb{L} = \{l_1, \ldots, l_N\}^\top, \quad l_i = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}
\]

where \(N\) is the batch size; \(d\) is a nonnegative, real-valued function quantifying the closeness of two tensors, referred to as the distance_function; and \text{margin} is a non-negative margin representing the minimum difference between the positive and negative distances that is required for the loss to be 0. The input tensors have \(N\) elements each and can be of any shape that the distance function can handle. If reduction is not 'none' (default 'mean'), then:

\[
\ell(x, y) = \begin{cases} 
\text{mean}(\mathbb{L}), & \text{if reduction = 'mean'}; \\
\text{sum}(\mathbb{L}), & \text{if reduction = 'sum'}. 
\end{cases}
\]

See also `nn_triplet_margin_loss()`, which computes the triplet loss for input tensors using the \(l_p\) distance as the distance function.
Unflattens a tensor dim expanding it to a desired shape. For use with [nn_sequential].
nn_upsample

Usage

```
nn_unflatten(dim, unflattened_size)
```

Arguments

- `dim`: Dimension to be unflattened
- `unflattened_size`: New shape of the unflattened dimension

Examples

```r
if (torch_is_installed()) {
  input <- torch_randn(2, 50)

  m <- nn_sequential(
    nn_linear(50, 50),
    nn_unflatten(2, c(2, 5, 5))
  )
  output <- m(input)
  output$size()
}
```

nn_upsample

Upsample module

Description

Upsamples a given multi-channel 1D (temporal), 2D (spatial) or 3D (volumetric) data. The input data is assumed to be of the form minibatch x channels x optional depth x optional height x width. Hence, for spatial inputs, we expect a 4D Tensor and for volumetric inputs, we expect a 5D Tensor.

Usage

```
nn_upsample(
  size = NULL,
  scale_factor = NULL,
  mode = "nearest",
  align_corners = NULL
)
```

Arguments

- `size`: (int or Tuple[int] or Tuple[int, int] or Tuple[int, int, int], optional): output spatial sizes
- `scale_factor`: (float or Tuple[float] or Tuple[float, float] or Tuple[float, float, float], optional): multiplier for spatial size. Has to match input size if it is a tuple.
- `mode`: (str, optional): the upsampling algorithm: one of 'nearest', 'linear', 'bilinear', 'bicubic' and 'trilinear'. Default: 'nearest'
align_corners (bool, optional): if True, the corner pixels of the input and output tensors are aligned, and thus preserving the values at those pixels. This only has effect when mode is 'linear', 'bilinear', or 'trilinear'. Default: FALSE

Details

The algorithms available for upsampling are nearest neighbor and linear, bilinear, bicubic and trilinear for 3D, 4D and 5D input Tensor, respectively.

One can either give a scale_factor or the target output size to calculate the output size. (You cannot give both, as it is ambiguous)

Examples

if (torch_is_installed()) {
  input <- torch_arange(start = 1, end = 4, dtype = torch_float())$view(c(1, 1, 2, 2))
  nn_upsample(scale_factor = c(2), mode = "nearest")(input)
  nn_upsample(scale_factor = c(2, 2), mode = "nearest")(input)
}

Description

The norm is computed over all gradients together, as if they were concatenated into a single vector. Gradients are modified in-place.

Usage

nn_utils_clip_grad_norm_(parameters, max_norm, norm_type = 2)

Arguments

parameters (IterableTensor or Tensor): an iterable of Tensors or a single Tensor that will have gradients normalized
max_norm (float or int): max norm of the gradients
norm_type (float or int): type of the used p-norm. Can beInf for infinity norm.

Value

Total norm of the parameters (viewed as a single vector).
nn_utils_clip_grad_value_

Clips gradient of an iterable of parameters at specified value.

Description

Gradients are modified in-place.

Usage

nn_utils_clip_grad_value_(parameters, clip_value)

Arguments

parameters (Iterable(Tensor) or Tensor): an iterable of Tensors or a single Tensor that will have gradients normalized
clip_value (float or int): maximum allowed value of the gradients.

Details

The gradients are clipped in the range [-clip_value, clip_value]

nn_utils_rnn_pack_padded_sequence

Packs a Tensor containing padded sequences of variable length.

Description

input can be of size T x B x * where T is the length of the longest sequence (equal to lengths[1]), B is the batch size, and * is any number of dimensions (including 0). If batch_first is TRUE, B x T x * input is expected.

Usage

nn_utils_rnn_pack_padded_sequence(
    input,
    lengths,
    batch_first = FALSE,
    enforce_sorted = TRUE
)
**Arguments**

- **input** (Tensor): padded batch of variable length sequences.
- **lengths** (Tensor): list of sequences lengths of each batch element.
- **batch_first** (bool, optional): if TRUE, the input is expected in B x T x * format.
- **enforce_sorted** (bool, optional): if TRUE, the input is expected to contain sequences sorted by length in a decreasing order. If FALSE, the input will get sorted unconditionally. Default: TRUE.

**Details**

For unsorted sequences, use enforce_sorted = FALSE. If enforce_sorted is TRUE, the sequences should be sorted by length in a decreasing order, i.e. input[1,1] should be the longest sequence, and input[B,1] the shortest one. enforce_sorted = TRUE is only necessary for ONNX export.

**Value**

a PackedSequence object

**Note**

This function accepts any input that has at least two dimensions. You can apply it to pack the labels, and use the output of the RNN with them to compute the loss directly. A Tensor can be retrieved from a PackedSequence object by accessing its .data attribute.
Details

For unsorted sequences, use enforce_sorted = FALSE. If enforce_sorted is TRUE, the sequences should be sorted in the order of decreasing length. enforce_sorted = TRUE is only necessary for ONNX export.

Value

a PackedSequence object

Examples

```r
if (torch_is_installed()) {
  x <- torch_tensor(c(1, 2, 3), dtype = torch_long())
  y <- torch_tensor(c(4, 5), dtype = torch_long())
  z <- torch_tensor(c(6), dtype = torch_long())
  p <- nn_utils_rnn_pack_sequence(list(x, y, z))
}
```

---

**nn_utils_rnn_pad_packed_sequence**

*Pads a packed batch of variable length sequences.*

Description

It is an inverse operation to `nn_utils_rnn_pack_padded_sequence()`.

Usage

```r
nn_utils_rnn_pad_packed_sequence(
  sequence,
  batch_first = FALSE,
  padding_value = 0,
  total_length = NULL
)
```

Arguments

- `sequence` (PackedSequence): batch to pad
- `batch_first` (bool, optional): if True, the output will be in “B x T x *” format.
- `padding_value` (float, optional): values for padded elements.
- `total_length` (int, optional): if not NULL, the output will be padded to have length total_length. This method will throw ValueError if total_length is less than the max sequence length in sequence.
Details

The returned Tensor’s data will be of size $T \times B \times *$, where $T$ is the length of the longest sequence and $B$ is the batch size. If batch_first is TRUE, the data will be transposed into $B \times T \times *$ format.

Value

Tuple of Tensor containing the padded sequence, and a Tensor containing the list of lengths of each sequence in the batch. Batch elements will be re-ordered as they were ordered originally when the batch was passed to nn_utils_rnn_pack_padded_sequence() or nn_utils_rnn_pack_sequence().

Note

total_length is useful to implement the pack sequence -> recurrent network -> unpack sequence pattern in a nn_module wrapped in ~torch.nn.DataParallel.

Examples

if (torch_is_installed()) {
  seq <- torch_tensor(rbind(c(1, 2, 0), c(3, 0, 0), c(4, 5, 6)))
  lens <- c(2, 1, 3)
  packed <- nn_utils_rnn_pack_padded_sequence(seq, lens,
    batch_first = TRUE,
    enforce_sorted = FALSE
  )
  packed
  nn_utils_rnn_pad_packed_sequence(packed, batch_first = TRUE)
}

nn_utils_rnn_pad_sequence

Pad a list of variable length Tensors with padding_value

Description

pad_sequence stacks a list of Tensors along a new dimension, and pads them to equal length. For example, if the input is list of sequences with size $L \times *$ and if batch_first is False, and $T \times B \times *$ otherwise.

Usage

nn_utils_rnn_pad_sequence(sequences, batch_first = FALSE, padding_value = 0)

Arguments

sequences (list[Tensor]): list of variable length sequences.
batch_first (bool, optional): output will be in $B \times T \times *$ if TRUE, or in $T \times B \times *$ otherwise
padding_value (float, optional): value for padded elements. Default: 0.
Details

\( B \) is batch size. It is equal to the number of elements in sequences. \( T \) is length of the longest sequence. \( L \) is length of the sequence. \( * \) is any number of trailing dimensions, including none.

Value

Tensor of size \( T \times B \times * \) if batch_first is FALSE. Tensor of size \( B \times T \times * \) otherwise.

Note

This function returns a Tensor of size \( T \times B \times * \) or \( B \times T \times * \) where \( T \) is the length of the longest sequence. This function assumes trailing dimensions and type of all the Tensors in sequences are same.

Examples

```r
if (torch_is_installed()) {
  a <- torch_ones(25, 300)
  b <- torch_ones(22, 300)
  c <- torch_ones(15, 300)
  nn_utils_rnn_pad_sequence(list(a, b, c))$size()
}
```

nn_utils_weight_norm

Description

Applies weight normalization to a parameter in the given module.

Details

\[
\mathbf{w} = g \frac{\mathbf{v}}{\|\mathbf{v}\|}
\]

Weight normalization is a reparameterization that decouples the magnitude of a weight tensor from its direction. This replaces the parameter specified by name (e.g. 'weight') with two parameters: one specifying the magnitude (e.g. 'weight_g') and one specifying the direction (e.g. 'weight_v').

Value

The original module with the weight_v and weight_g parameters.
Methods

Public methods:

• `nn_utils_weight_norm$new()`
• `nn_utils_weight_norm$compute_weight()`
• `nn_utils_weight_norm$apply()`
• `nn_utils_weight_norm$call()`
• `nn_utils_weight_norm$recompute()`
• `nn_utils_weight_norm$remove()`
• `nn_utils_weight_norm$clone()`

Method `new()`:

Usage:
`nn_utils_weight_norm$new(name, dim)`

Arguments:
- `name` (str, optional): name of weight parameter
- `dim` (int, optional): dimension over which to compute the norm

Method `compute_weight()`:

Usage:
`nn_utils_weight_norm$compute_weight(module, name = NULL, dim = NULL)`

Arguments:
- `module` (Module): containing module
- `name` (str, optional): name of weight parameter
- `dim` (int, optional): dimension over which to compute the norm

Method `apply()`:

Usage:
`nn_utils_weight_norm$apply(module, name = NULL, dim = NULL)`

Arguments:
- `module` (Module): containing module
- `name` (str, optional): name of weight parameter
- `dim` (int, optional): dimension over which to compute the norm

Method `call()`:

Usage:
`nn_utils_weight_norm$call(module)`

Arguments:
- `module` (Module): containing module

Method `recompute()`:

Usage:
`nn_utils_weight_norm$recompute(module)`
**Arguments:**

module (Module): containing module

**Method remove():**

**Usage:**

```r
nn_utils_weight_norm$remove(module, name = NULL)
```

**Arguments:**

module (Module): containing module

name (str, optional): name of weight parameter

**Method clone():** The objects of this class are cloneable with this method.

**Usage:**

```r
nn_utils_weight_norm$clone(deep = FALSE)
```

**Arguments:**

deep Whether to make a deep clone.

**Note**

The pytorch Weight normalization is implemented via a hook that recomputes the weight tensor from the magnitude and direction before every `forward()` call. Since torch for R still do not support hooks, the weight recomputation need to be done explicitly inside the `forward()` definition trough a call of the `recompute()` method. See examples.

By default, with `dim = 0`, the norm is computed independently per output channel/plane. To compute a norm over the entire weight tensor, use `dim = NULL`.

@references [https://arxiv.org/abs/1602.07868](https://arxiv.org/abs/1602.07868)

**Examples**

```r
if (torch_is_installed()) {
  x = nn_linear(in_features = 20, out_features = 40)
  weight_norm = nn_utils_weight_norm$new(name = 'weight', dim = 2)
  weight_norm$apply(x)
  x$weight_g$size()
  x$weight_v$size()
  x$weight

  # the recompute() method recomputes the weight using g and v. It must be called
  # explicitly inside `forward()`.
  weight_norm$recompute(x)
}
```
optimizer  

Creates a custom optimizer

Description

When implementing custom optimizers you will usually need to implement the `initialize` and `step` methods. See the example section below for a full example.

Usage

```r
optimizer(
  name = NULL,
  inherit = Optimizer,
  ..., 
  private = NULL, 
  active = NULL, 
  parent_env = parent.frame()
)
```

Arguments

- `name` (optional) name of the optimizer
- `inherit` (optional) you can inherit from other optimizers to re-use some methods.
- `...` Pass any number of fields or methods. You should at least define the `initialize` and `step` methods. See the examples section.
- `private` (optional) a list of private methods for the optimizer.
- `active` (optional) a list of active methods for the optimizer.
- `parent_env` used to capture the right environment to define the class. The default is fine for most situations.

Warning

If you need to move a model to GPU via `$cuda()`, please do so before constructing optimizers for it. Parameters of a model after `$cuda()` will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Examples

```r
if (torch_is_installed()) {

  # In this example we will create a custom optimizer
  # that's just a simplified version of the 'optim_sgd' function.

  optim_sgd2 <- optimizer(
    initialize = function(params, learning_rate) {
        ...
    }
  )
}
```
optim_adadelta <- function() {
  defaults <- list(
    learning_rate = learning_rate
  )
  super$initialize(params, defaults)
  step = function() {
    with_no_grad({
      for (g in seq_along(self$param_groups)) {
        group <- self$param_groups[[g]]
        for (p in seq_along(group$params)) {
          param <- group$params[[p]]

          if (is.null(param$grad) || is_undefined_tensor(param$grad)) {
            next
          }

          param$add_(param$grad, alpha = -group$learning_rate)
        }
      }
    })
  }
}

x <- torch_randn(1, requires_grad = TRUE)
opt <- optim_sgd2(x, learning_rate = 0.1)
for (i in 1:100) {
  opt$zero_grad()
  y <- x^2
  y$backward()
  opt$step()
} all.equal(x$item(), 0, tolerance = 1e-9)

---

**optim_adadelta**  
*Adadelta optimizer*

**Description**
It has been proposed in ADADELTA: An Adaptive Learning Rate Method

**Usage**

```
optim_adadelta(params, lr = 1, rho = 0.9, eps = 1e-06, weight_decay = 0)
```

**Arguments**

- **params** (iterable): list of parameters to optimize or list defining parameter groups
- **lr** (float, optional): learning rate (default: 1e-3)
rho (float, optional): coefficient used for computing a running average of squared gradients (default: 0.9)

eps (float, optional): term added to the denominator to improve numerical stability (default: 1e-6)

weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

Warning

If you need to move a model to GPU via $\texttt{cuda()}$, please do so before constructing optimizers for it. Parameters of a model after $\texttt{cuda()}$ will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Note

According to the original paper, decaying average of the squared gradients is computed as follows:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho)g_t^2$$

RMS of previous squared gradients up to time $t$:

$$RMS[g_t] = \sqrt{E[g^2]_t + \epsilon}$$

Adadelta update rule:

$$\Delta \theta_t = -\frac{RMS[\Delta \theta]_{t-1}}{RMS[g_t]} \theta_{t+1} = \theta_t + \Delta \theta_t$$

Examples

```{r}
if (torch_is_installed()) {
    ## Not run:
    optimizer <- optim_adadelta(model$parameters, lr = 0.1)
    optimizer$zero_grad()
    loss_fn(model(input), target)$backward()
    optimizer$step()

    ## End(Not run)
}
```

Description

Proposed in *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*
Usage

```python
optim_adagrad(
    params,
    lr = 0.01,
    lr_decay = 0,
    weight_decay = 0,
    initial_accumulator_value = 0,
    eps = 1e-10
)
```

Arguments

- **params** (iterable): list of parameters to optimize or list parameter groups
- **lr** (float, optional): learning rate (default: 1e-2)
- **lr_decay** (float, optional): learning rate decay (default: 0)
- **weight_decay** (float, optional): weight decay (L2 penalty) (default: 0)
- **initial_accumulator_value**
  the initial value for the accumulator. (default: 0)

Adagrad is an especially good optimizer for sparse data. It individually modifies learning rate for every single parameter, dividing the original learning rate value by sum of the squares of the gradients. It causes that the rarely occurring features get greater learning rates. The main downside of this method is the fact that learning rate may be getting small too fast, so that at some point a model cannot learn anymore.

- **eps** (float, optional): term added to the denominator to improve numerical stability (default: 1e-10)

Warning

If you need to move a model to GPU via `$cuda()` , please do so before constructing optimizers for it. Parameters of a model after `$cuda()` will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Note

Update rule:

$$
θ_{t+1} = θ_t - \frac{η}{\sqrt{G_t} + \epsilon} \odot g_t
$$

The equation above and some remarks quoted after *An overview of gradient descent optimization algorithms* by Sebastian Ruder.
Implements Adam algorithm.

Description

It has been proposed in Adam: A Method for Stochastic Optimization.

Usage

```r
optim_adam(
  params,
  lr = 0.001,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0,
  amsgrad = FALSE
)
```

Arguments

- `params` (iterable): iterable of parameters to optimize or dicts defining parameter groups
- `lr` (float, optional): learning rate (default: 1e-3)
- `betas` (tuple of float, float, optional): coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- `eps` (float, optional): term added to the denominator to improve numerical stability (default: 1e-8)
- `weight_decay` (float, optional): weight decay (L2 penalty) (default: 0)
- `amsgrad` (boolean, optional): whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: FALSE)

Warning

If you need to move a model to GPU via `cuda()`, please do so before constructing optimizers for it. Parameters of a model after `cuda()` will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Examples

```r
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_adam(model$parameters(), lr = 0.1)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()
}
optim_adamw

*Implements AdamW algorithm*

**Description**

For further details regarding the algorithm we refer to *Decoupled Weight Decay Regularization*.

**Usage**

```r
optim_adamw(
  params,
  lr = 0.001,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0.01,
  amsgrad = FALSE
)
```

**Arguments**

- `params` *(iterable)*: iterable of parameters to optimize or dicts defining parameter groups
- `lr` *(float, optional)*: learning rate (default: 1e-3)
- `betas` *(Tuple[float, float], optional)*: coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- `eps` *(float, optional)*: term added to the denominator to improve numerical stability (default: 1e-8)
- `weight_decay` *(float, optional)*: weight decay (L2 penalty) (default: 0)
- `amsgrad` *(boolean, optional)*: whether to use the AMSGrad variant of this algorithm from the paper *On the Convergence of Adam and Beyond* (default: FALSE)

optim_asgd

*Averaged Stochastic Gradient Descent optimizer*

**Description**

Proposed in *Acceleration of stochastic approximation by averaging*.
optim_lbfgs

Usage

    optim_asgd(
        params,
        lr = 0.01,
        lambda = 1e-04,
        alpha = 0.75,
        t0 = 1e+06,
        weight_decay = 0
    )

Arguments

    params (iterable): iterable of parameters to optimize or lists defining parameter groups
    lr (float): learning rate
    lambda (float, optional): decay term (default: 1e-4)
    alpha (float, optional): power for eta update (default: 0.75)
    t0 (float, optional): point at which to start averaging (default: 1e6)
    weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

Warning

    If you need to move a model to GPU via $cuda()$, please do so before constructing optimizers for it. Parameters of a model after $cuda()$ will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Examples

    if (torch_is_installed()) {
        ## Not run:
        optimizer <- optim_asgd(model$parameters(), lr = 0.1)
        optimizer$zero_grad()
        loss_fn(model(input), target)$backward()
        optimizer$step()
        
        ## End(Not run)
    }

optim_lbfgs  

LBFGS optimizer

Description

    Implements L-BFGS algorithm, heavily inspired by minFunc
Usage

```r
optim_lbfgs(
  params,
  lr = 1,
  max_iter = 20,
  max_eval = NULL,
  tolerance_grad = 1e-07,
  tolerance_change = 1e-09,
  history_size = 100,
  line_search_fn = NULL
)
```

Arguments

- `params` (iterable): iterable of parameters to optimize or dicts defining parameter groups
- `lr` (float): learning rate (default: 1)
- `max_iter` (int): maximal number of iterations per optimization step (default: 20)
- `max_eval` (int): maximal number of function evaluations per optimization step (default: `max_iter * 1.25`).
- `tolerance_grad` (float): termination tolerance on first order optimality (default: 1e-5).
- `tolerance_change` (float): termination tolerance on function value/parameter changes (default: 1e-9).
- `history_size` (int): update history size (default: 100).
- `line_search_fn` (str): either 'strong_wolfe' or None (default: None).

Details

This optimizer is different from the others in that in `optimizer$step()`, it needs to be passed a closure that (1) calculates the loss, (2) calls `backward()` on it, and (3) returns it. See example below.

Warning

This optimizer doesn’t support per-parameter options and parameter groups (there can be only one).

Right now all parameters have to be on a single device. This will be improved in the future.

If you need to move a model to GPU via `$cuda()`, please do so before constructing optimizers for it. Parameters of a model after `$cuda()` will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Note

This is a very memory intensive optimizer (it requires additional `param_bytes * (history_size + 1)` bytes). If it doesn’t fit in memory try reducing the history size, or use a different algorithm.
Examples

```r
if (torch_is_installed()) {
  a <- 1
  b <- 5
  rosenbrock <- function(x) {
    x1 <- x[1]
    x2 <- x[2]
    (a - x1)^2 + b * (x2 - x1^2)^2
  }
  x <- torch_tensor(c(-1, 1), requires_grad = TRUE)
  optimizer <- optim_lbfgs(x)
  calc_loss <- function() {
    optimizer$zero_grad()
    value <- rosenbrock(x)
    value$backward()
    value
  }
  num_iterations <- 2
  for (i in 1:num_iterations) {
    optimizer$step(calc_loss)
  }
  rosenbrock(x)
}
```

---

`optim_required`  
*Dummy value indicating a required value.*

**Description**

export

**Usage**

`optim_required()`

---

`optim_rmsprop`  
*RMSprop optimizer*

**Description**

Proposed by G. Hinton in his course.
Usage

```python
optim_rmsprop(
    params,
    lr = 0.01,
    alpha = 0.99,
    eps = 1e-08,
    weight_decay = 0,
    momentum = 0,
    centered = FALSE
)
```

Arguments

- **params** (iterable): iterable of parameters to optimize or list defining parameter groups
- **lr** (float, optional): learning rate (default: 1e-2)
- **alpha** (float, optional): smoothing constant (default: 0.99)
- **eps** (float, optional): term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** optional weight decay penalty. (default: 0)
- **momentum** (float, optional): momentum factor (default: 0)
- **centered** (bool, optional): if TRUE, compute the centered RMSProp, the gradient is normalized by an estimation of its variance weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

Warning

If you need to move a model to GPU via $\texttt{cuda()}$, please do so before constructing optimizers for it. Parameters of a model after $\texttt{cuda()}$ will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Note

The centered version first appears in *Generating Sequences With Recurrent Neural Networks*. The implementation here takes the square root of the gradient average before adding epsilon (note that TensorFlow interchanges these two operations). The effective learning rate is thus $\alpha / (\sqrt{v} + \epsilon)$ where $\alpha$ is the scheduled learning rate and $v$ is the weighted moving average of the squared gradient.

Update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t} + \epsilon} \cdot g_t$$
Implements the resilient backpropagation algorithm.

Description

Proposed first in RPROP - A Fast Adaptive Learning Algorithm

Usage

```r
optim_rprop(params, lr = 0.01, etas = c(0.5, 1.2), step_sizes = c(1e-06, 50))
```

Arguments

- `params` (iterable): iterable of parameters to optimize or lists defining parameter groups
- `lr` (float, optional): learning rate (default: 1e-2)
- `etas` (Tuple(float, float), optional): pair of (etaminus, etaplus), that are multiplicative increase and decrease factors (default: (0.5, 1.2))
- `step_sizes` (vector(float, float), optional): a pair of minimal and maximal allowed step sizes (default: (1e-6, 50))

Warning

If you need to move a model to GPU via `$cuda()`, please do so before constructing optimizers for it. Parameters of a model after `$cuda()` will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Examples

```r
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_rprop(model$parameters(), lr = 0.1)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()

  ## End(Not run)
}
```
optim_sgd

SGD optimizer

Description

Implements stochastic gradient descent (optionally with momentum). Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Usage

```r
optim_sgd(
  params,
  lr = optim_required(),
  momentum = 0,
  dampening = 0,
  weight_decay = 0,
  nesterov = FALSE
)
```

Arguments

- **params** (iterable): iterable of parameters to optimize or dicts defining parameter groups
- **lr** (float): learning rate
- **momentum** (float, optional): momentum factor (default: 0)
- **dampening** (float, optional): dampening for momentum (default: 0)
- **weight_decay** (float, optional): weight decay (L2 penalty) (default: 0)
- **nesterov** (bool, optional): enables Nesterov momentum (default: FALSE)

Note

The implementation of SGD with Momentum-Nesterov subtly differs from Sutskever et. al. and implementations in some other frameworks.

Considering the specific case of Momentum, the update can be written as

\[
\begin{align*}
    v_{t+1} &= \mu \cdot v_t + g_{t+1}, \\
    p_{t+1} &= p_t - lr \cdot v_{t+1},
\end{align*}
\]

where \( p, g, v \) and \( \mu \) denote the parameters, gradient, velocity, and momentum respectively.

This is in contrast to Sutskever et. al. and other frameworks which employ an update of the form

\[
\begin{align*}
    v_{t+1} &= \mu \cdot v_t + lr \cdot g_{t+1}, \\
    p_{t+1} &= p_t - v_{t+1}.
\end{align*}
\]

The Nesterov version is analogously modified.
Warning

If you need to move a model to GPU via $\texttt{cuda()}$, please do so before constructing optimizers for it. Parameters of a model after $\texttt{cuda()}$ will be different objects from those before the call. In general, you should make sure that the objects pointed to by model parameters subject to optimization remain the same over the whole lifecycle of optimizer creation and usage.

Examples

```r
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_sgd(model$parameters(), lr = 0.1, momentum = 0.9)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()

  ## End(Not run)
}
```

sampler

Creates a new Sampler

Description

Samplers can be used with `\texttt{dataloader()}` when creating batches from a torch `\texttt{dataset()}`.

Usage

```r
sampler(
  name = NULL,
  inherit = Sampler,
  ...,
  private = NULL,
  active = NULL,
  parent_env = parent.frame()
)
```

Arguments

- **name**: (optional) name of the sampler
- **inherit**: (optional) you can inherit from other samplers to re-use some methods.
- **...**: Pass any number of fields or methods. You should at least define the `initialize` and `step` methods. See the examples section.
- **private**: (optional) a list of private methods for the sampler
- **active**: (optional) a list of active methods for the sampler.
- **parent_env**: used to capture the right environment to define the class. The default is fine for most situations.
Details
A sampler must implement the `iter` and `length()` methods.

- `initialize` takes in a `data_source`. In general this is a `dataset()`.
- `iter` returns a function that returns a dataset index everytime it’s called.
- `length` returns the maximum number of samples that can be retrieved from that sampler.

### tensor_dataset
**Dataset wrapping tensors.**

Description
Each sample will be retrieved by indexing tensors along the first dimension.

Usage
tensor_dataset(...)  

Arguments

...  tensors that have the same size of the first dimension.

### threads
**Number of threads**

Description
Get and set the numbers used by torch computations.

Usage
torch_set_num_threads(num_threads)  
torch_set_num_interop_threads(num_threads)  
torch_get_num_interop_threads()  
torch_get_num_threads()

Arguments

num_threads  number of threads to set.

Details
For details see the CPU threading article in the PyTorch documentation.
torch_abs

Note
torch_set_threads do not work on macOS system as it must be 1.

torch_abs  Abs

Description
Abs

Usage
torch_abs(self)

Arguments
self (Tensor) the input tensor.

abs(input) -> Tensor
Computes the element-wise absolute value of the given input tensor.

\[
\text{out}_i = |\text{input}_i|
\]

Examples
if (torch_is_installed()) {
    torch_abs(torch_tensor(c(-1, -2, 3)))
}

torch_absolute  Absolute

Description
Absolute

Usage
torch_absolute(self)

Arguments
self (Tensor) the input tensor.

absolute(input, *, out=None) -> Tensor
Alias for torch_abs()
torch_acos

**Description**

Acos

**Usage**

torch_acos(self)

**Arguments**

self (Tensor) the input tensor.

acos(input) -> Tensor

Returns a new tensor with the arccosine of the elements of input.

\[
\text{out}_i = \cos^{-1}(\text{input}_i)
\]

**Examples**

```python
if (torch_is_installed()) {
    a = torch.randn(c(4))
    a
    torch.acos(a)
}
```

---

torch_acosh

**Description**

Acosh

**Usage**

torch_acosh(self)

**Arguments**

self (Tensor) the input tensor.
acosh(input, *, out=None) -> Tensor

Returns a new tensor with the inverse hyperbolic cosine of the elements of input.

Note

The domain of the inverse hyperbolic cosine is [1, inf) and values outside this range will be mapped to NaN, except for + INF for which the output is mapped to + INF.

\[ \text{out}_i = \cosh^{-1}(\text{input}_i) \]

Examples

```python
if (torch_is_installed()) {
    a <- torch_randn(c(4))$uniform_(1, 2)
    a
    torch_acosh(a)
}
```

torch_adaptive_avg_pool1d

Adaptive_avg_pool1d

Description

Adaptive_avg_pool1d

Usage

torch_adaptive_avg_pool1d(self, output_size)

Arguments

- **self** the input tensor
- **output_size** the target output size (single integer)

Adaptive_avg_pool1d(input, output_size) -> Tensor

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

See `nn_adaptive_avg_pool1d()` for details and output shape.
torch_add

Add

Description
Add

Usage
torch_add(self, other, alpha = 1L)

Arguments
self (Tensor) the input tensor.
other (Tensor/Number) the second input tensor/number.
alpha (Number) the scalar multiplier for other

add(input, other, out=NULL)
Adds the scalar other to each element of the input input and returns a new resulting tensor.

out = input + other

If input is of type FloatTensor or DoubleTensor, other must be a real number, otherwise it should be an integer.

add(input, other, *, alpha=1, out=NULL)
Each element of the tensor other is multiplied by the scalar alpha and added to each element of the tensor input. The resulting tensor is returned.
The shapes of input and other must be broadcastable.

out = input + alpha × other

If other is of type FloatTensor or DoubleTensor, alpha must be a real number, otherwise it should be an integer.

Examples
if (torch_is_installed()) {

a = torch_randn(c(4))
a
torch_add(a, 20)

a = torch_randn(c(4))
a
torch_addbmm

b = torch.randn(c(4, 1))
b = torch_add(a, b)
}

torch_addbmm

Addbmm

Description
Addbmm

Usage
torch_addbmm(self, batch1, batch2, beta = 1L, alpha = 1L)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>(Tensor) matrix to be added</td>
</tr>
<tr>
<td>batch1</td>
<td>(Tensor) the first batch of matrices to be multiplied</td>
</tr>
<tr>
<td>batch2</td>
<td>(Tensor) the second batch of matrices to be multiplied</td>
</tr>
<tr>
<td>beta</td>
<td>(Number, optional) multiplier for input (β)</td>
</tr>
<tr>
<td>alpha</td>
<td>(Number, optional) multiplier for batch1 @ batch2 (α)</td>
</tr>
</tbody>
</table>

addbmm(input, batch1, batch2, *, beta=1, alpha=1, out=NULL) -> Tensor

Performs a batch matrix-matrix product of matrices stored in batch1 and batch2, with a reduced add step (all matrix multiplications get accumulated along the first dimension). input is added to the final result.

batch1 and batch2 must be 3-D tensors each containing the same number of matrices.

If batch1 is a \((b \times n \times m)\) tensor, batch2 is a \((b \times m \times p)\) tensor, input must be broadcastable with a \((n \times p)\) tensor and out will be a \((n \times p)\) tensor.

\[
out = \beta \text{ input} + \alpha \left( \sum_{i=0}^{b-1} \text{batch1}_i \odot \text{batch2}_i \right)
\]

For inputs of type FloatTensor or DoubleTensor, arguments beta and alpha must be real numbers, otherwise they should be integers.

Examples

if (torch_is_installed()) {
M = torch.randn(c(3, 5))
batch1 = torch.randn(c(10, 3, 4))
batch2 = torch.randn(c(10, 4, 5))
torch_addbmm(M, batch1, batch2)
}
torch_addcdiv

Description

Addcdiv

Usage

torch_addcdiv(self, tensor1, tensor2, value = 1L)

Arguments

self (Tensor) the tensor to be added
tensor1 (Tensor) the numerator tensor
tensor2 (Tensor) the denominator tensor
value (Number, optional) multiplier for tensor1/tensor2

addcdiv(input, tensor1, tensor2, *, value=1, out=NULL) -> Tensor

Performs the element-wise division of tensor1 by tensor2, multiply the result by the scalar value and add it to input.

Warning

Integer division with addcdiv is deprecated, and in a future release addcdiv will perform a true division of tensor1 and tensor2. The current addcdiv behavior can be replicated using torch_floor_divide() for integral inputs (input + value × tensor1 // tensor2) and torch_div() for float inputs (input + value × tensor1 / tensor2). The new addcdiv behavior can be implemented with torch_true_divide() (input + value × torch.true_divide(tensor1, tensor2)).

\[
\text{out}_i = \text{input}_i + \text{value} \times \frac{\text{tensor1}_i}{\text{tensor2}_i}
\]

The shapes of input, tensor1, and tensor2 must be broadcastable.

For inputs of type FloatTensor or DoubleTensor, value must be a real number, otherwise an integer.

Examples

if (torch_is_installed()) {

    t = torch_randn(c(1, 3))
t1 = torch_randn(c(3, 1))
t2 = torch_randn(c(1, 3))
torch_addcdiv(t, t1, t2, 0.1)
}
**torch_addcmul**  

**Addcmul**

**Description**  
Addcmul

**Usage**  
torch_addcmul(self, tensor1, tensor2, value = 1L)

**Arguments**

- **self** (Tensor) the tensor to be added
- **tensor1** (Tensor) the tensor to be multiplied
- **tensor2** (Tensor) the tensor to be multiplied
- **value** (Number, optional) multiplier for tensor1 $\times$ tensor2

**addcmul(input, tensor1, tensor2, *, value=1, out=NULL) -> Tensor**

Performs the element-wise multiplication of tensor1 by tensor2, multiply the result by the scalar value and add it to input.

$$out_i = \text{input}_i + \text{value} \times \text{tensor}_1 \times \text{tensor}_2$$

The shapes of tensor, tensor1, and tensor2 must be broadcastable.

For inputs of type FloatTensor or DoubleTensor, value must be a real number, otherwise an integer.

**Examples**

```python
if (torch_is_installed()) {
    t = torch.randn(c(1, 3))
    t1 = torch.randn(c(3, 1))
    t2 = torch.randn(c(1, 3))
    torch_addcmul(t, t1, t2, 0.1)
}
```
torch_addmm

Description
Addmm

Usage
torch_addmm(self, mat1, mat2, beta = 1L, alpha = 1L)

Arguments
- **self** (Tensor) matrix to be added
- **mat1** (Tensor) the first matrix to be multiplied
- **mat2** (Tensor) the second matrix to be multiplied
- **beta** (Number, optional) multiplier for input ($\beta$)
- **alpha** (Number, optional) multiplier for $\text{mat1 @ mat2}$ ($\alpha$)

addmm(input, mat1, mat2, *, beta=1, alpha=1, out=NULL) -> Tensor

Performs a matrix multiplication of the matrices mat1 and mat2. The matrix input is added to the final result.

If mat1 is a $(n \times m)$ tensor, mat2 is a $(m \times p)$ tensor, then input must be broadcastable with a $(n \times p)$ tensor and out will be a $(n \times p)$ tensor.

alpha and beta are scaling factors on matrix-vector product between mat1 and mat2 and the added matrix input respectively.

$$\text{out} = \beta \text{input} + \alpha (\text{mat1} @ \text{mat2})$$

For inputs of type FloatTensor or DoubleTensor, arguments beta and alpha must be real numbers, otherwise they should be integers.

Examples
if (torch_is_installed()) {

M = torch_randn(c(2, 3))
mat1 = torch_randn(c(2, 3))
mat2 = torch_randn(c(3, 3))
torch_addmm(M, mat1, mat2)
}
torch_addmv

Description
Addmv

Usage
torch_addmv(self, mat, vec, beta = 1L, alpha = 1L)

Arguments
- **self** (Tensor) vector to be added
- **mat** (Tensor) matrix to be multiplied
- **vec** (Tensor) vector to be multiplied
- **beta** (Number, optional) multiplier for input ($\beta$)
- **alpha** (Number, optional) multiplier for $\text{mat} \odot \text{vec}$ ($\alpha$)

addmv(input, mat, vec, *, beta=1, alpha=1, out=NULL) -> Tensor

Performs a matrix-vector product of the matrix mat and the vector vec. The vector input is added to the final result.

If mat is a $(n \times m)$ tensor, vec is a 1-D tensor of size $m$, then input must be broadcastable with a 1-D tensor of size $n$ and out will be 1-D tensor of size $n$.

alpha and beta are scaling factors on matrix-vector product between mat and vec and the added tensor input respectively.

$$\text{out} = \beta \text{ input} + \alpha (\text{mat} \odot \text{vec})$$

For inputs of type FloatTensor or DoubleTensor, arguments beta and alpha must be real numbers, otherwise they should be integers.

Examples
if (torch_is_installed()) {

  M = torch_randn(c(2))
  mat = torch_randn(c(2, 3))
  vec = torch_randn(c(3))
  torch_addmv(M, mat, vec)
}

torch_addr

Description
Addr

Usage
torch_addr(self, vec1, vec2, beta = 1L, alpha = 1L)

Arguments

self (Tensor) matrix to be added
vec1 (Tensor) the first vector of the outer product
vec2 (Tensor) the second vector of the outer product
beta (Number, optional) multiplier for input ($\beta$)
alpha (Number, optional) multiplier for vec1 $\otimes$ vec2 ($\alpha$)

addr(input, vec1, vec2, *, beta=1, alpha=1, out=NULL) -> Tensor

Performs the outer-product of vectors vec1 and vec2 and adds it to the matrix input.
Optional values beta and alpha are scaling factors on the outer product between vec1 and vec2 and the added matrix input respectively.

$$\text{out} = \beta \text{input} + \alpha (\text{vec1} \otimes \text{vec2})$$

If vec1 is a vector of size n and vec2 is a vector of size m, then input must be broadcastable with a matrix of size ($n \times m$) and out will be a matrix of size ($n \times m$).

For inputs of type FloatTensor or DoubleTensor, arguments beta and alpha must be real numbers, otherwise they should be integers

Examples

```python
if (torch_is_installed()) {

vec1 = torch.arange(1, 3)
vec2 = torch.arange(1, 2)
M = torch.zeros(c(3, 2))
torch_addr(M, vec1, vec2)
}
```
torch_allclose

Description

Allclose

Usage

torch_allclose(self, other, rtol = 1e-05, atol = 1e-08, equal_nan = FALSE)

Arguments

- **self**: (Tensor) first tensor to compare
- **other**: (Tensor) second tensor to compare
- **rtol**: (float, optional) relative tolerance. Default: 1e-05
- **atol**: (float, optional) absolute tolerance. Default: 1e-08
- **equal_nan**: (bool, optional) if TRUE, then two NaNs will be compared as equal. Default: FALSE

torch_allclose(torch_tensor(c(10000., 1e-07)), torch_tensor(c(10000.1, 1e-08)))
torch_allclose(torch_tensor(c(10000., 1e-08)), torch_tensor(c(10000.1, 1e-09)))
torch_allclose(torch_tensor(c(1.0, NaN)), torch_tensor(c(1.0, NaN)))
torch_allclose(torch_tensor(c(1.0, NaN)), torch_tensor(c(1.0, NaN)), equal_nan=True)

Examples

if (torch_is_installed()) {
  torch_allclose(torch_tensor(c(10000., 1e-07)), torch_tensor(c(10000.1, 1e-08)))
  torch_allclose(torch_tensor(c(10000., 1e-08)), torch_tensor(c(10000.1, 1e-09)))
  torch_allclose(torch_tensor(c(1.0, NaN)), torch_tensor(c(1.0, NaN)))
  torch_allclose(torch_tensor(c(1.0, NaN)), torch_tensor(c(1.0, NaN)), equal_nan=TRUE)
}
Description

Amax

Usage

torch_amax(self, dim = list(), keepdim = FALSE)

Arguments

self (Tensor) the input tensor.
dim (int or tuple of ints) the dimension or dimensions to reduce.
keepdim (bool) whether the output tensor has dim retained or not.

amax(input, dim, keepdim=FALSE, *, out=None) -> Tensor

Returns the maximum value of each slice of the input tensor in the given dimension(s) dim.

Note

The difference between max/min and amax/amin is:

- amax/amin supports reducing on multiple dimensions,
- amax/amin does not return indices,
- amax/amin evenly distributes gradient between equal values, while max(dim)/min(dim) propagates gradient only to a single index in the source tensor.

If keepdim is TRUE, the output tensors are of the same size as input except in the dimension(s) dim where they are of size 1. Otherwise, dim is squeezed (see torch_squeeze()), resulting in the output tensors having fewer dimensions than in input'.

Examples

if (torch_is_installed()) {
    a <- torch_randn(c(4, 4))
    a
    torch_amax(a, 1)
}
**torch_amin**

<table>
<thead>
<tr>
<th>torch_amin</th>
<th>Amin</th>
</tr>
</thead>
</table>

**Description**

Amin

**Usage**

```r
torch_amin(self, dim = list(), keepdim = FALSE)
```

**Arguments**

- `self` (Tensor) the input tensor.
- `dim` (int or tuple of ints) the dimension or dimensions to reduce.
- `keepdim` (bool) whether the output tensor has `dim` retained or not.

**amin(input, dim, keepdim=FALSE, *, out=None) -> Tensor**

Returns the minimum value of each slice of the `input` tensor in the given dimension(s) `dim`.

**Note**

The difference between `max/min` and `amax/amin` is:

- `amax/amin` supports reducing on multiple dimensions,
- `amax/amin` does not return indices,
- `amax/amin` evenly distributes gradient between equal values, while `max(dim)/min(dim)` propagates gradient only to a single index in the source tensor.

If `keepdim` is `TRUE`, the output tensors are of the same size as `input` except in the dimension(s) `dim` where they are of size 1. Otherwise, dims are squeezed (see `torch_squeeze()`), resulting in the output tensors having fewer dimensions than `input`.

**Examples**

```r
if (torch_is_installed()) {

  a <- torch_randn(c(4, 4))
  a
  torch_amin(a, 1)
}
```
torch_angle  

**Description**

Angle

**Usage**

```
torch_angle(self)
```

**Arguments**

- `self` (Tensor) the input tensor.

**angle(input) -> Tensor**

Computes the element-wise angle (in radians) of the given input tensor.

\[
\text{out}_i = \text{angle}(\text{input}_i)
\]

**Examples**

```python
if (torch_is_installed()) {
    ## Not run:
    torch_angle(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))*180/3.14159
    ## End(Not run)
}
```

torch_arange  

**Description**

Arange

**Examples**

```python
if (torch_is_installed()) {
    ## Not run:
    torch_arange(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))*180/3.14159
    ## End(Not run)
}
```
torch_arange

Usage

    torch_arange(
        start,
        end,
        step = 1,
        dtype = NULL,
        layout = NULL,
        device = NULL,
        requires_grad = FALSE
    )

Arguments

    start  (Number) the starting value for the set of points. Default: 0.
    end    (Number) the ending value for the set of points
    step   (Number) the gap between each pair of adjacent points. Default: 1.
    dtype  (torch.dtype, optional) the desired data type of returned tensor. Default: if
           NULL, uses a global default (see torch_set_default_tensor_type). If dtype
           is not given, infer the data type from the other input arguments. If any of
           start, end, or stop are floating-point, the dtype is inferred to be the default
dtype, see ~torch.get_default_ddtype. Otherwise, the dtype is inferred to be
           torch.int64.
    layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
    device (torch.device, optional) the desired device of returned tensor. Default: if
           NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type).
           device will be the CPU for CPU tensor types and the current CUDA device for
           CUDA tensor types.
    requires_grad (bool, optional) If autograd should record operations on the returned tensor. De-
dfault: FALSE.

    arange(start=0, end, step=1, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, re-
           quires_grad=False) -> Tensor

Returns a 1-D tensor of size \( \left\lceil \frac{end - start}{step} \right\rceil \) with values from the interval \([start, end)\) taken with
common difference step beginning from start.

Note that non-integer step is subject to floating point rounding errors when comparing against end;
to avoid inconsistency, we advise adding a small epsilon to end in such cases.

    out_{i+1} = out_i + step

Examples

    if (torch_is_installed()) {
        torch_arange(start = 0, end = 5)
        torch_arange(1, 4)
torch_arccosh

torch_arange(1, 2.5, 0.5)
}

torch_arccos  Arccos

Description
Arccos

Usage
torch_arccos(self)

Arguments
self (Tensor) the input tensor.

arccos(input, *, out=None) -> Tensor
Alias for torch_acos().

torch_arccosh  Arccosh

Description
Arccosh

Usage
torch_arccosh(self)

Arguments
self (Tensor) the input tensor.

arccosh(input, *, out=None) -> Tensor
Alias for torch_acosh().
torch_arcsin

<table>
<thead>
<tr>
<th>Description</th>
<th>Arcsin</th>
</tr>
</thead>
</table>

Usage

torch_arcsin(self)

Arguments

self (Tensor) the input tensor.

arcsin(input, *, out=None) -> Tensor

Alias for torch_asin().

torch_arcsinh

<table>
<thead>
<tr>
<th>Description</th>
<th>Arcsinh</th>
</tr>
</thead>
</table>

Usage

torch_arcsinh(self)

Arguments

self (Tensor) the input tensor.

arcsinh(input, *, out=None) -> Tensor

Alias for torch_asinh().
torch_arctan

**Description**

Arctan

**Usage**

torch_arctan(self)

**Arguments**

self  
(Tensor) the input tensor.

arctan(input, *, out=None) -> Tensor

Alias for torch_atan().

---

torch_arctanh

**Description**

Arctanh

**Usage**

torch_arctanh(self)

**Arguments**

self  
(Tensor) the input tensor.

arctanh(input, *, out=None) -> Tensor

Alias for torch_atanh().
torch_argmax

Argmax

Description
Argmax

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>(Tensor) the input tensor.</td>
</tr>
<tr>
<td>dim</td>
<td>(int) the dimension to reduce. If NULL, the argmax of the flattened input is returned.</td>
</tr>
<tr>
<td>keepdim</td>
<td>(bool) whether the output tensor has dim retained or not. Ignored if dim=NULL.</td>
</tr>
</tbody>
</table>

argmax(input) -> LongTensor

Returns the indices of the maximum value of all elements in the input tensor.
This is the second value returned by torch_max. See its documentation for the exact semantics of this method.

argmax(input, dim, keepdim=False) -> LongTensor

Returns the indices of the maximum values of a tensor across a dimension.
This is the second value returned by torch_max. See its documentation for the exact semantics of this method.

Examples

```r
if (torch_is_installed()) {

  ## Not run:
  a = torch_randn(c(4, 4))
  a
  torch_argmax(a)

  ## End(Not run)

  a = torch_randn(c(4, 4))
  a
  torch_argmax(a, dim=1)
}
```
torch_argmin

Description

Argmin

Arguments

- **self**: (Tensor) the input tensor.
- **dim**: (int) the dimension to reduce. If NULL, the argmin of the flattened input is returned.
- **keepdim**: (bool) whether the output tensor has dim retained or not. Ignored if dim=NULL.

argmin(input) -> LongTensor

Returns the indices of the minimum value of all elements in the input tensor.

This is the second value returned by torch_min. See its documentation for the exact semantics of this method.

argmin(input, dim, keepdim=False, out=NULL) -> LongTensor

Returns the indices of the minimum values of a tensor across a dimension.

This is the second value returned by torch_min. See its documentation for the exact semantics of this method.

Examples

```python
if (torch_is_installed()) {

    a = torch_randn(c(4, 4))
    a
    torch_argmin(a)

    a = torch_randn(c(4, 4))
    a
    torch_argmin(a, dim=1)
}
```
### torch_argsort

**Description**

Argsort

**Usage**

```python
torch_argsort(self, dim = -1L, descending = FALSE)
```

**Arguments**

- **self** *(Tensor)* the input tensor.
- **dim** *(int, optional)* the dimension to sort along
- **descending** *(bool, optional)* controls the sorting order (ascending or descending)

**argsort(input, dim=-1, descending=False) -> LongTensor**

Returns the indices that sort a tensor along a given dimension in ascending order by value.

This is the second value returned by `torch_sort`. See its documentation for the exact semantics of this method.

**Examples**

```python
if (torch_is_installed()) {
    a = torch_randn(c(4, 4))
    a
    torch_argsort(a, dim=1)
}
```

### torch_asin

**Description**

Asin

**Usage**

```python
torch_asin(self)
```

**Arguments**

- **self** *(Tensor)* the input tensor.
asinh(input, out=NULL) -> Tensor

Returns a new tensor with the inverse hyperbolic sine of the elements of input.

\[
\text{out}_i = \sinh^{-1}(\text{input}_i)
\]

Examples

```r
if (torch_is_installed()) {
  a <- torch_randn(c(4))
a  torch_asinh(a)
}
```

Description

Asinh

Usage

torch_asinh(self)

Arguments

self (Tensor) the input tensor.

asinh(input, *, out=None) -> Tensor

Returns a new tensor with the inverse hyperbolic sine of the elements of input.

\[
\text{out}_i = \sinh^{-1}(\text{input}_i)
\]

Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(4))
a  torch_asin(a)
}
```
torch_as_strided

Description

As_strided

Usage

torch_as_strided(self, size, stride, storage_offset = NULL)

Arguments

self  (Tensor) the input tensor.
size  (tuple or ints) the shape of the output tensor
stride (tuple or ints) the stride of the output tensor
storage_offset  (int, optional) the offset in the underlying storage of the output tensor

as_strided(input, size, stride, storage_offset=0) -> Tensor

Create a view of an existing torch_Tensor input with specified size, stride and storage_offset.

Warning

More than one element of a created tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensors, please clone them first.

Many PyTorch functions, which return a view of a tensor, are internally implemented with this function. Those functions, like `torch_Tensor.expand`, are easier to read and are therefore more advisable to use.

Examples

```python
if (torch_is_installed()) {
    x = torch_randn(c(3, 3))
    x
    t = torch_as_strided(x, list(2, 2), list(1, 2))
    t
    t = torch_as_strided(x, list(2, 2), list(1, 2), 1)
    t
}
```
torch_atan

Description
Atan

Usage
torch_atan(self)

Arguments
self (Tensor) the input tensor.

atan(input, out=NULL) -> Tensor
Returns a new tensor with the arctangent of the elements of input.

\[ \text{out}_i = \tan^{-1}(\text{input}_i) \]

Examples
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
torch_atan(a)
}

torch_atan2

Description
Atan2

Usage
torch_atan2(self, other)

Arguments
self (Tensor) the first input tensor
other (Tensor) the second input tensor
atan2(input, other, out=NULL) -> Tensor

Element-wise arctangent of \( \frac{i}{\text{other}} \), with consideration of the quadrant. Returns a new tensor with the signed angles in radians between vector \((\text{other}_i, \text{input}_i)\) and vector \((1, 0)\). (Note that \(\text{other}_i\), the second parameter, is the x-coordinate, while \(\text{input}_i\), the first parameter, is the y-coordinate.)

The shapes of \(\text{input}\) and \(\text{other}\) must be broadcastable.

**Examples**

```python
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_atan2(a, torch_randn(c(4)))
}
```

---

torch_atanh

**Atanh**

**Description**

Atanh

**Usage**

`torch_atanh(self)`

**Arguments**

- `self` (Tensor) the input tensor.

**atanh(input, *, out=None) -> Tensor**

Returns a new tensor with the inverse hyperbolic tangent of the elements of \(\text{input}\).

**Note**

The domain of the inverse hyperbolic tangent is \((-1, 1)\) and values outside this range will be mapped to NaN, except for the values 1 and -1 for which the output is mapped to +/-INF respectively.

\[ \text{out}_i = \tanh^{-1}(\text{input}_i) \]
torch_atleast_1d

Examples

```r
if (torch_is_installed()) {
    x <- torch_randn(c(2))
    x
    torch_atleast_1d(x)
    x <- torch_tensor(1.)
    x
    torch_atleast_1d(x)
    x <- torch_tensor(0.5)
    y <- torch_tensor(1.)
    torch_atleast_1d(list(x, y))
}
```

torch_atleast_1d  Atleast_1d

Description

Returns a 1-dimensional view of each input tensor with zero dimensions. Input tensors with one or more dimensions are returned as-is.

Usage

```r
torch_atleast_1d(self)
```

Arguments

- **self**: (Tensor or list of Tensors)

Examples

```r
if (torch_is_installed()) {
    a = torch_randn(c(4))$uniform_(-1, 1)
    a
    torch_atanh(a)
}
```
**torch_atleast_2d**

**Atleast_2d**

**Description**

Returns a 2-dimensional view of each each input tensor with zero dimensions. Input tensors with two or more dimensions are returned as-is.

**Usage**

```python
torch_atleast_2d(self)
```

**Arguments**

- `self` (Tensor or list of Tensors)

**Examples**

```python
if (torch_is_installed()) {
    x <- torch_tensor(1.)
    x
    torch_atleast_2d(x)
    x <- torch_randn(c(2,2))
    x
    torch_atleast_2d(x)
    x <- torch_tensor(0.5)
    y <- torch_tensor(1.)
    torch_atleast_2d(list(x,y))
}
```

**torch_atleast_3d**

**Atleast_3d**

**Description**

Returns a 3-dimensional view of each each input tensor with zero dimensions. Input tensors with three or more dimensions are returned as-is.

**Usage**

```python
torch_atleast_3d(self)
```

**Arguments**

- `self` (Tensor or list of Tensors)
torch_avg_pool1d

* **avg_pool1d**

### Description

Avg_pool1d

### Usage

```python
torch_avg_pool1d(
    self,
    kernel_size,
    stride = list(),
    padding = 0L,
    ceil_mode = FALSE,
    count_include_pad = TRUE
)
```

### Arguments

- **self**: input tensor of shape (minibatch, in_channels, iW)
- **kernel_size**: the size of the window. Can be a single number or a tuple (kW,)
- **stride**: the stride of the window. Can be a single number or a tuple (sW,). Default: kernel_size
- **padding**: implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0
- **ceil_mode**: when TRUE, will use ceil instead of floor to compute the output shape. Default: FALSE
- **count_include_pad**: when TRUE, will include the zero-padding in the averaging calculation. Default: TRUE

### avg_pool1d

```python
avg_pool1d(input, kernel_size, stride=NULL, padding=0, ceil_mode=FALSE, count_include_pad=TRUE) -> Tensor
```

Applies a 1D average pooling over an input signal composed of several input planes.

See *nn_avg_pool1d()* for details and output shape.
torch_baddbmm

Description

Baddbmm

Usage

torch_baddbmm(self, batch1, batch2, beta = 1L, alpha = 1L)

Arguments

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>(Tensor) the tensor to be added</td>
</tr>
<tr>
<td>batch1</td>
<td>(Tensor) the first batch of matrices to be multiplied</td>
</tr>
<tr>
<td>batch2</td>
<td>(Tensor) the second batch of matrices to be multiplied</td>
</tr>
<tr>
<td>beta</td>
<td>(Number, optional) multiplier for input (β)</td>
</tr>
<tr>
<td>alpha</td>
<td>(Number, optional) multiplier for batch1 @ batch2 (α)</td>
</tr>
</tbody>
</table>

baddbmm(input, batch1, batch2, *, beta=1, alpha=1, out=NULL) -> Tensor

Performs a batch matrix-matrix product of matrices in batch1 and batch2. input is added to the final result.

batch1 and batch2 must be 3-D tensors each containing the same number of matrices.
If batch1 is a $(b \times n \times m)$ tensor, batch2 is a $(b \times m \times p)$ tensor, then input must be broadcastable with a $(b \times n \times p)$ tensor and out will be a $(b \times n \times p)$ tensor. Both alpha and beta mean the same as the scaling factors used in torch_addbmm.

\[
\text{out}_i = \beta \text{input}_i + \alpha (\text{batch1}_i \odot \text{batch2}_i)
\]

For inputs of type FloatTensor or DoubleTensor, arguments beta and alpha must be real numbers, otherwise they should be integers.

Examples

```python
if (torch_is_installed()) {

  M = torch.randn(c(10, 3, 5))
  batch1 = torch.randn(c(10, 3, 4))
  batch2 = torch.randn(c(10, 4, 5))
  torch_baddbmm(M, batch1, batch2)
}
```
Bartlett_window

Description

Bartlett_window

Usage

torch_bartlett_window(
    window_length,
    periodic = TRUE,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)

Arguments

window_length  (int) the size of returned window
periodic       (bool, optional) If TRUE, returns a window to be used as periodic function. If False, return a symmetric window.
dtype          (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type). Only floating point types are supported.
layout         (torch.layout, optional) the desired layout of returned window tensor. Only torch_strided (dense layout) is supported.
device         (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad  (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

bartlett_window(window_length, periodic=TRUE, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Bartlett window function.

\[ w[n] = 1 - \left| \frac{2n}{N-1} - 1 \right| = \begin{cases} 
\frac{2n}{N-1} & \text{if } 0 \leq n \leq \frac{N-1}{2} \\
\frac{2n}{N-1} - \frac{2n}{N-1} & \text{if } \frac{N-1}{2} < n < N 
\end{cases} \]

where \( N \) is the full window size.

The input window_length is a positive integer controlling the returned window size. periodic flag determines whether the returned window trims off the last duplicate value from the symmetric
window and is ready to be used as a periodic window with functions like torch_stft. Therefore, if periodic is true, the \( N \) in above formula is in fact window_length + 1. Also, we always have torch_bartlett_window(L, periodic=TRUE) equal to torch_bartlett_window(L + 1, periodic=False)[:-1]).

**Note**

If `window_length` \( \equiv 1 \), the returned window contains a single value 1.

---

**torch_bernoulli**

**Bernoulli**

**Description**

Bernoulli

**Usage**

torch_bernoulli(self, p, generator = NULL)

**Arguments**

- **self** (Tensor) the input tensor of probability values for the Bernoulli distribution
- **p** (Number) a probability value. If \( p \) is passed then it's used instead of the values in self tensor.
- **generator** (torch.Generator, optional) a pseudorandom number generator for sampling

**bernoulli(input, *, generator=NULL, out=NULL) -> Tensor**

Draws binary random numbers (0 or 1) from a Bernoulli distribution.

The input tensor should be a tensor containing probabilities to be used for drawing the binary random number. Hence, all values in input have to be in the range: \( 0 \leq \text{input}_i \leq 1 \).

The \( i^{th} \) element of the output tensor will draw a value 1 according to the \( i^{th} \) probability value given in input.

\[
\text{out}_i \sim \text{Bernoulli}(p = \text{input}_i)
\]

The returned out tensor only has values 0 or 1 and is of the same shape as input.

out can have integral dtype, but input must have floating point dtype.
torch_bincount

**Description**

Bincount

**Arguments**

- **self**: (Tensor) 1-d int tensor
- **weights**: (Tensor) optional, weight for each value in the input tensor. Should be of same size as input tensor.
- **minlength**: (int) optional, minimum number of bins. Should be non-negative.

**bincount(input, weights=NULL, minlength=0) -> Tensor**

Count the frequency of each value in an array of non-negative ints.

The number of bins (size 1) is one larger than the largest value in input unless input is empty, in which case the result is a tensor of size 0. If minlength is specified, the number of bins is at least minlength and if input is empty, then the result is tensor of size minlength filled with zeros. If n is the value at position i, out[n] += weights[i] if weights is specified else out[n] += 1.

.. include:: cuda_deterministic.rst

**Examples**

```r
if (torch_is_installed()) {

    input = torch_randint(1, 8, list(5), dtype=torch_int64())
    weights = torch_linspace(0, 1, steps=5)
    input
    weights
    torch_bincount(input, weights)
    input$bincount(weights)
}
```
torch_bitwise_and

Description
Bitwise_and

Usage
torch_bitwise_and(self, other)

Arguments
- **self**: NA the first input tensor
- **other**: NA the second input tensor

**bitwise_and(input, other, out=NULL) -> Tensor**

Computes the bitwise AND of `input` and `other`. The input tensor must be of integral or Boolean types. For bool tensors, it computes the logical AND.

torch_bitwise_not

Description
Bitwise_not

Usage
torch_bitwise_not(self)

Arguments
- **self**: (Tensor) the input tensor.

**bitwise_not(input, out=NULL) -> Tensor**

Computes the bitwise NOT of the given input tensor. The input tensor must be of integral or Boolean types. For bool tensors, it computes the logical NOT.
### torch_bitwise_or

**Description**

Bitwise_or

**Usage**

torch_bitwise_or(self, other)

**Arguments**

- **self**: NA, the first input tensor
- **other**: NA, the second input tensor

**bitwise_or(input, other, out=NULL) -> Tensor**

Computes the bitwise OR of input and other. The input tensor must be of integral or Boolean types. For bool tensors, it computes the logical OR.

---

### torch_bitwise_xor

**Description**

Bitwise_xor

**Usage**

torch_bitwise_xor(self, other)

**Arguments**

- **self**: NA, the first input tensor
- **other**: NA, the second input tensor

**bitwise_xor(input, other, out=NULL) -> Tensor**

Computes the bitwise XOR of input and other. The input tensor must be of integral or Boolean types. For bool tensors, it computes the logical XOR.
Description

Blackman_window

Usage

```
torch_blackman_window(
    window_length,
    periodic = TRUE,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)
```

Arguments

- `window_length` (int) the size of returned window
- `periodic` (bool, optional) If TRUE, returns a window to be used as periodic function. If False, return a symmetric window.
- `dtype` (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see `torch_set_default_tensor_type`). Only floating point types are supported.
- `layout` (torch.layout, optional) the desired layout of returned window tensor. Only `torch.strided` (dense layout) is supported.
- `device` (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see `torch_set_default_tensor_type`). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- `requires_grad` (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

```
blackman_window(window_length, periodic=TRUE, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor
```

Blackman window function.

\[
w[n] = 0.42 - 0.5 \cos \left( \frac{2\pi n}{N-1} \right) + 0.08 \cos \left( \frac{4\pi n}{N-1} \right)
\]

where \( N \) is the full window size.

The input `window_length` is a positive integer controlling the returned window size. periodic flag determines whether the returned window trims off the last duplicate value from the symmetric
torch_bmm

window and is ready to be used as a periodic window with functions like torch_stft. Therefore, if periodic is true, the \( N \) in above formula is in fact \( \text{window\_length} + 1 \). Also, we always have \( \text{torch\_blackman\_window}(L, \text{periodic=TRUE}) = \text{torch\_blackman\_window}(L + 1, \text{periodic=False})[\text{:-1}] \).

Note

If \(`\text{window\_length}`\) \( \equiv 1 \), the returned window contains a single value 1.

---

torch_block_diag

**Description**

Create a block diagonal matrix from provided tensors.

**Usage**

```
torch_block_diag(tensors)
```

**Arguments**

- `tensors` (list of tensors) One or more tensors with 0, 1, or 2 dimensions.

**Examples**

```
if (torch_is_installed()) {

  A <- torch_tensor(rbind(c(0, 1), c(1, 0)))
  B <- torch_tensor(rbind(c(3, 4, 5), c(6, 7, 8)))
  C <- torch_tensor(7)
  D <- torch_tensor(rbind(4, 5, 6))
  E <- torch_tensor(rbind(1, 2, 3))
  torch_block_diag(list(A, B, C, D, E))
}
```

---

torch_bmm

**Description**

Bmm

**Usage**

```
torch_bmm(self, mat2)
```
torch_broadcast_tensors

Arguments

- **self** (Tensor) the first batch of matrices to be multiplied
- **mat2** (Tensor) the second batch of matrices to be multiplied

bmm(input, mat2, out=NULL) -> Tensor

Performs a batch matrix-matrix product of matrices stored in `input` and `mat2`. `input` and `mat2` must be 3-D tensors each containing the same number of matrices. If `input` is a \((b \times n \times m)\) tensor, `mat2` is a \((b \times m \times p)\) tensor, `out` will be a \((b \times n \times p)\) tensor.

\[ \text{out}_i = \text{input}_i @ \text{mat2}_i \]

Note

This function does not broadcast. For broadcasting matrix products, see `torch_matmul`.

Examples

```r
if (torch_is_installed()) {

  input = torch_randn(c(10, 3, 4))
  mat2 = torch_randn(c(10, 4, 5))
  res = torch_bmm(input, mat2)
  res
}
```

---

torch_broadcast_tensors

Broadcast_tensors

Description

Broadcast_tensors

Usage

torch_broadcast_tensors(tensors)

Arguments

- **tensors** a list containing any number of tensors of the same type

broadcast_tensors(tensors) -> List of Tensors

Broadcasts the given tensors according to broadcasting-semantics.
torch_bucketize

Examples
if (torch_is_installed()) {
    x = torch_arange(0, 3)$view(c(1, 4))
    y = torch_arange(0, 2)$view(c(3, 1))
    out = torch_broadcast_tensors(list(x, y))
    out[[1]]
}

torch_bucketize

Bucketize

Description
Bucketize

Usage
torch_bucketize(self, boundaries, out_int32 = FALSE, right = FALSE)

Arguments
self (Tensor or Scalar) N-D tensor or a Scalar containing the search value(s).
boundaries (Tensor) 1-D tensor, must contain a monotonically increasing sequence.
out_int32 (bool, optional) – indicate the output data type. torch_int32() if True, torch_int64() otherwise. Default value is FALSE, i.e. default output data type is torch_int64().
right (bool, optional) – if False, return the first suitable location that is found. If True, return the last such index. If no suitable index found, return 0 for non-numerical value (eg. nan, inf) or the size of boundaries (one pass the last index). In other words, if False, gets the lower bound index for each value in input from boundaries. If True, gets the upper bound index instead. Default value is False.

bucketize(input, boundaries, *, out_int32=FALSE, right=FALSE, out=None) -> Tensor

Returns the indices of the buckets to which each value in the input belongs, where the boundaries of the buckets are set by boundaries. Return a new tensor with the same size as input. If right is FALSE (default), then the left boundary is closed.

Examples
if (torch_is_installed()) {
    boundaries <- torch_tensor(c(1, 3, 5, 7, 9))
    v <- torch_tensor(rbind(c(3, 6, 9), c(3, 6, 9)))
    v
    torch_bucketize(v, boundaries)
    torch_bucketize(v, boundaries, right=TRUE)
}
torch_can_cast

Description

Can_cast

Usage

torch_can_cast(from, to)

Arguments

from (dtype) The original torch_dtype.
to (dtype) The target torch_dtype.

can_cast(from, to) -> bool

Determines if a type conversion is allowed under PyTorch casting rules described in the type promotion documentation.

Examples

if (torch_is_installed()) {
    torch_can_cast(torch_double(), torch_float())
    torch_can_cast(torch_float(), torch_int())
    }

torch_cartesian_prod

Description

Do cartesian product of the given sequence of tensors.

Usage

torch_cartesian_prod(tensors)

Arguments

tensors a list containing any number of 1 dimensional tensors.
torch_cat

Examples

```r
if (torch_is_installed()) {
    a = c(1, 2, 3)
    b = c(4, 5)
    tensor_a = torch_tensor(a)
    tensor_b = torch_tensor(b)
    torch_cartesian_prod(list(tensor_a, tensor_b))
}
```

---

torch_cat

Cat

Description

Cat

Usage

```r
torch_cat(tensors, dim = 1L)
```

Arguments

- **tensors** (sequence of Tensors) any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.
- **dim** (int, optional) the dimension over which the tensors are concatenated

cat(tensors, dim=0, out=NULL) -> Tensor

Concatenates the given sequence of seq tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

**torch_cat** can be seen as an inverse operation for **torch_split()** and **torch_chunk**.

**torch_cat** can be best understood via examples.

Examples

```r
if (torch_is_installed()) {
    x = torch_randn(c(2, 3))
    x
    torch_cat(list(x, x, x), 1)
    torch_cat(list(x, x, x), 2)
}
```
torch_cdist

Description
Cdist

Usage
torch_cdist(x1, x2, p = 2L, compute_mode = NULL)

Arguments
x1 (Tensor) input tensor of shape $B \times P \times M$.
x2 (Tensor) input tensor of shape $B \times R \times M$.
p NA $p$ value for the $p$-norm distance to calculate between each vector pair $\in [0, \infty]$.
compute_mode NA 'use_mm_for_euclid_dist_if_necessary' - will use matrix multiplication approach to calculate euclidean distance ($p = 2$) if $P > 25$ or $R > 25$ 'use_mm_for_euclid_dist' - will always use matrix multiplication approach to calculate euclidean distance ($p = 2$) 'donot_use_mm_for_euclid_dist' - will never use matrix multiplication approach to calculate euclidean distance ($p = 2$) Default: use_mm_for_euclid_dist_if_necessary.

TEST
Computes batched the $p$-norm distance between each pair of the two collections of row vectors.

torch.ceil

Description
Ceil

Usage
torch.ceil(self)

Arguments
self (Tensor) the input tensor.
ceil(input, out=NULL) -> Tensor

Returns a new tensor with the ceil of the elements of input, the smallest integer greater than or equal to each element.

\[ \text{out}_i = \lceil \text{input}_i \rceil = \lfloor \text{input}_i \rfloor + 1 \]

Examples

```c
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a, torch.ceil(a)
}
```

torch_celu

Celu

Description

Celu

Usage

torch_celu(self, alpha = 1L)

Arguments

- **self**: the input tensor
- **alpha**: the alpha value for the CELU formulation. Default: 1.0

celu(input, alpha=1.) -> Tensor

See nnf_celu() for more info.
torch_celu

Description

Celu

Usage

torch_celu(self, alpha = 1L)

Arguments

self the input tensor
alpha the alpha value for the CELU formulation. Default: 1.0

celu_(input, alpha=1.) -> Tensor

In-place version of torch_celu().

torch_chain_matmul

Description

Chain_matmul

Usage

torch_chain_matmul(matrices)

Arguments

matrices (Tensors...) a sequence of 2 or more 2-D tensors whose product is to be determined.

TEST

Returns the matrix product of the N 2-D tensors. This product is efficiently computed using the matrix chain order algorithm which selects the order in which incurs the lowest cost in terms of arithmetic operations ([CLRS]_). Note that since this is a function to compute the product, N needs to be greater than or equal to 2; if equal to 2 then a trivial matrix-matrix product is returned. If N is 1, then this is a no-op - the original matrix is returned as is.
torch_channel_shuffle

Channel_shuffle

Description

Channel_shuffle

Usage

torch_channel_shuffle(self, groups)

Arguments

self (Tensor) the input tensor

groups (int) number of groups to divide channels in and rearrange.

Divide the channels in a tensor of shape

math:(*, C, H, W):

Divide the channels in a tensor of shape (*, C, H, W) into g groups and rearrange them as (*, C/g, g, H, W), while keeping the original tensor shape.

Examples

if (torch_is_installed()) {

    input <- torch_randn(c(1, 4, 2, 2))
    print(input)
    output <- torch_channel_shuffle(input, 2)
    print(output)
}
torch_cholesky

Description

Cholesky

Usage

torch_cholesky(self, upper = FALSE)

Arguments

- **self** (Tensor) the input tensor of size \((*, n, n)\) where \(*\) is zero or more batch dimensions consisting of symmetric positive-definite matrices.
- **upper** (bool, optional) flag that indicates whether to return a upper or lower triangular matrix. Default: FALSE

cholesky(input, upper=False, out=NULL) -> Tensor

Computes the Cholesky decomposition of a symmetric positive-definite matrix \(A\) or for batches of symmetric positive-definite matrices.

If upper is TRUE, the returned matrix \(U\) is upper-triangular, and the decomposition has the form:

\[
A = U^T U
\]

If upper is FALSE, the returned matrix \(L\) is lower-triangular, and the decomposition has the form:

\[
A = LL^T
\]

If upper is TRUE, and \(A\) is a batch of symmetric positive-definite matrices, then the returned tensor will be composed of upper-triangular Cholesky factors of each of the individual matrices. Similarly, when upper is FALSE, the returned tensor will be composed of lower-triangular Cholesky factors of each of the individual matrices.

Examples

```r
if (torch_is_installed()) {

  a = torch_randn(c(3, 3))
  a = torch_mm(a, a$t()) # make symmetric positive-definite
  l = torch_cholesky(a)
  a
  l
  torch_mm(l, l$t())
  a = torch_randn(c(3, 2, 2))
  # Not run:
  a = torch_matmul(a, a$transpose(-1, -2)) + 1e-03 # make symmetric positive-definite
```
torch_cholesky_inverse

Cholesky_inverse

Description
Cholesky_inverse

Usage
torch_cholesky_inverse(self, upper = FALSE)

Arguments
self (Tensor) the input 2-D tensor $u$, a upper or lower triangular Cholesky factor
upper (bool, optional) whether to return a lower (default) or upper triangular matrix

cholesky_inverse(input, upper=FALSE, out=NULL) -> Tensor

Computes the inverse of a symmetric positive-definite matrix $A$ using its Cholesky factor $u$: returns matrix $inv$. The inverse is computed using LAPACK routines dpotri and spotri (and the corresponding MAGMA routines).

If upper is FALSE, $u$ is lower triangular such that the returned tensor is

$$inv = (uu^T)^{-1}$$

If upper is TRUE or not provided, $u$ is upper triangular such that the returned tensor is

$$inv = (u^Tu)^{-1}$$

Examples
if (torch_is_installed()) {

## Not run:
a = torch_randn(c(3, 3))
a = torch_mm(a, a$t()) + 1e-05 * torch_eye(3) # make symmetric positive definite
u = torch_cholesky(a)
a
torch_cholesky_inverse(u)
a$inverse()
torch_cholesky_solve

Description

Cholesky_solve

Usage

```r
torch_cholesky_solve(self, input2, upper = FALSE)
```

Arguments

- `self` (Tensor) input matrix `b` of size `(*, m, k)`, where `*` is zero or more batch dimensions
- `input2` (Tensor) input matrix `u` of size `(*, m, m)`, where `*` is zero of more batch dimensions composed of upper or lower triangular Cholesky factor
- `upper` (bool, optional) whether to consider the Cholesky factor as a lower or upper triangular matrix. Default: FALSE.

`cholesky_solve(input, input2, upper=FALSE, out=NULL) -> Tensor`

Solves a linear system of equations with a positive semidefinite matrix to be inverted given its Cholesky factor matrix `u`.

If `upper` is FALSE, `u` is and lower triangular and `c` is returned such that:

```
c = (uu^T)^{-1}b
```

If `upper` is TRUE or not provided, `u` is upper triangular and `c` is returned such that:

```
c = (u^T u)^{-1}b
```

torch_cholesky_solve(b, u) can take in 2D inputs `b`, `u` or inputs that are batches of 2D matrices. If the inputs are batches, then returns batched outputs `c`

Examples

```r
if (torch_is_installed()) {
    a = torch_rannd(c(3, 3))
    a = torch_mm(a, a$t()) # make symmetric positive definite
    u = torch_cholesky(a)
    a
    b = torch_rannd(c(3, 2))
    ```
torch_chol_decomp

b
torch_cholesky_solve(b, u)
torch_mm(a$inverse(), b)
}

torch_chunk

<table>
<thead>
<tr>
<th>Description</th>
<th>Chunk</th>
</tr>
</thead>
</table>

Usage

torch_chunk(self, chunks, dim = 1L)

Arguments

- **self**: (Tensor) the tensor to split
- **chunks**: (int) number of chunks to return
- **dim**: (int) dimension along which to split the tensor

chunk(input, chunks, dim=0) -> List of Tensors

Splits a tensor into a specific number of chunks. Each chunk is a view of the input tensor.

Last chunk will be smaller if the tensor size along the given dimension `dim` is not divisible by `chunks`.

torch_clamp

<table>
<thead>
<tr>
<th>Description</th>
<th>Clamp</th>
</tr>
</thead>
</table>

Usage

torch_clamp(self, min = NULL, max = NULL)

Arguments

- **self**: (Tensor) the input tensor.
- **min**: (Number) lower-bound of the range to be clamped to
- **max**: (Number) upper-bound of the range to be clamped to
clamp(input, min, max, out=NULL) -> Tensor

Clamp all elements in input into the range \([\text{min}, \text{max}]\) and return a resulting tensor:

\[
y_i = \begin{cases} 
\text{min} & \text{if } x_i < \text{min} \\
 x_i & \text{if } \text{min} \leq x_i \leq \text{max} \\
\text{max} & \text{if } x_i > \text{max}
\end{cases}
\]

If input is of type FloatTensor or DoubleTensor, args min and max must be real numbers, otherwise they should be integers.

clamp(input, *, min, out=NULL) -> Tensor

Clamps all elements in input to be larger or equal min.

If input is of type FloatTensor or DoubleTensor, value should be a real number, otherwise it should be an integer.

clamp(input, *, max, out=NULL) -> Tensor

Clamps all elements in input to be smaller or equal max.

If input is of type FloatTensor or DoubleTensor, value should be a real number, otherwise it should be an integer.

Examples

```python
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_clamp(a, min=-0.5, max=0.5)

    a = torch_randn(c(4))
    a
    torch_clamp(a, min=0.5)

    a = torch_randn(c(4))
    a
    torch_clamp(a, max=0.5)
}
```

---

**torch_clip**  

**Clip**

**Description**

Clip
### Usage

```plaintext
torch_clip(self, min = NULL, max = NULL)
```

### Arguments

- **self** (Tensor) the input tensor.
- **min** (Number) lower-bound of the range to be clamped to
- **max** (Number) upper-bound of the range to be clamped to

```plaintext
clip(input, min, max, *, out=None) -> Tensor
```

* Alias for `torch_clamp()`.

### Description

Clone

### Usage

```plaintext
torch_clone(self, memory_format = NULL)
```

### Arguments

- **self** (Tensor) the input tensor.
- **memory_format** a torch memory format. see `torch_preserve_format()`.

```plaintext
clone(input, *, memory_format=torch.preserve_format) -> Tensor
```

* Returns a copy of input.

### Note

This function is differentiable, so gradients will flow back from the result of this operation to `input`. To create a tensor without an autograd relationship to `input` see `Tensor$detach`.  

---

**torch_clone**  
Clone

---

**torch_clip**  
Usage

```plaintext
torch_clip(self, min = NULL, max = NULL)
```

### Arguments

- **self** (Tensor) the input tensor.
- **min** (Number) lower-bound of the range to be clamped to
- **max** (Number) upper-bound of the range to be clamped to

```plaintext
clip(input, min, max, *, out=None) -> Tensor
```

* Alias for `torch_clamp()`.
torch_combinations  Combinations

Description
Combinations

Usage
torch_combinations(self, r = 2L, with_replacement = FALSE)

Arguments
self          (Tensor) 1D vector.
r            (int, optional) number of elements to combine
with_replacement
            (boolean, optional) whether to allow duplication in combination

combinations(input, r=2, with_replacement=False) -> seq

Compute combinations of length r of the given tensor. The behavior is similar to python's itertools.combinations when with_replacement is set to False, and itertools.combinations_with_replacement when with_replacement is set to TRUE.

Examples
if (torch_is_installed()) {
    a = c(1, 2, 3)
tensor_a = torch_tensor(a)
torch_combinations(tensor_a)
torch_combinations(tensor_a, r=3)
torch_combinations(tensor_a, with_replacement=TRUE)
}

torch_complex  Complex

Description
Complex

Usage
torch_complex(real, imag)
**Arguments**

- `real` *(Tensor)* The real part of the complex tensor. Must be float or double.
- `imag` *(Tensor)* The imaginary part of the complex tensor. Must be same dtype as `real`.

**complex(real, imag, *, out=None) -> Tensor**

Constructs a complex tensor with its real part equal to `real` and its imaginary part equal to `imag`.

**Examples**

```r
if (torch_is_installed()) {
    real <- torch_tensor(c(1, 2), dtype=torch_float32())
    imag <- torch_tensor(c(3, 4), dtype=torch_float32())
    z <- torch_complex(real, imag)
    z
    z$dtype
}
```

---

**torch_conj**

**Conj**

**Description**

Conj

**Usage**

`torch_conj(self)`

**Arguments**

- `self` *(Tensor)* the input tensor.

**conj(input) -> Tensor**

Computes the element-wise conjugate of the given `input` tensor.

\[
\text{out}_i = \text{conj}(\text{input}_i)
\]

**Examples**

```r
if (torch_is_installed()) {
    ## Not run:
    torch_conj(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))
    ## End(Not run)
}
```
Description

Conv1d

Usage

torch_conv1d(input, weight, bias = list(), stride = 1L, padding = 0L, dilation = 1L, groups = 1L)

Arguments

- **input**: input tensor of shape \((\text{minibatch}, \text{in_channels}, iW)\)
- **weight**: filters of shape \((\text{out_channels}, \frac{\text{in_channels}}{\text{groups}}, kW)\)
- **bias**: optional bias of shape \((\text{out_channels})\). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a one-element tuple \((sW,)\). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a one-element tuple \((padW,)\). Default: 0
- **dilation**: the spacing between kernel elements. Can be a single number or a one-element tuple \((dW,)\). Default: 1
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1

conv1d(input, weight, bias=NULL, stride=1, padding=0, dilation=1, groups=1) -> Tensor

Applies a 1D convolution over an input signal composed of several input planes.

See `nn_conv1d()` for details and output shape.

Examples

```r
if (torch_is_installed()) {

  filters = torch_randn(c(33, 16, 3))
  inputs = torch_randn(c(20, 16, 50))
  nnf_conv1d(inputs, filters)
}
```
**torch_conv2d**

**Conv2d**

**Description**

Conv2d

**Usage**

```r
torch_conv2d(  
    input,  
    weight,  
    bias = list(),  
    stride = 1L,  
    padding = 0L,  
    dilation = 1L,  
    groups = 1L  
)
```

**Arguments**

- **input**: input tensor of shape (minibatch, in_channels, iH, iW)
- **weight**: filters of shape (out_channels, in_channels/groups, kH, kW)
- **bias**: optional bias tensor of shape (out_channels). Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a tuple (sH, sW). Default: 1
- **padding**: implicit paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0
- **dilation**: the spacing between kernel elements. Can be a single number or a tuple (dH, dW). Default: 1
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1

**Examples**

```r
if (torch_is_installed()) {
  # With square kernels and equal stride
  filters = torch_randn(c(8,4,3,3))
  inputs = torch_randn(c(1,4,5,5))
  nnf_conv2d(inputs, filters, padding=1)
}
```
**torch.conv3d**  

**Conv3d**

**Description**

Conv3d

**Usage**

```r
torch.conv3d(
    input,
    weight,
    bias = list(),
    stride = 1L,
    padding = 0L,
    dilation = 1L,
    groups = 1L
)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>input tensor of shape (minibatch, in_channels, iT, iH, iW)</td>
</tr>
<tr>
<td>weight</td>
<td>filters of shape (out_channels, in_channels / groups, kT, kH, kW)</td>
</tr>
<tr>
<td>bias</td>
<td>optional bias tensor of shape (out_channels). Default: NULL</td>
</tr>
<tr>
<td>stride</td>
<td>the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sW). Default: 1</td>
</tr>
<tr>
<td>padding</td>
<td>implicit paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0</td>
</tr>
<tr>
<td>dilation</td>
<td>the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1</td>
</tr>
<tr>
<td>groups</td>
<td>split input into groups, in_channels should be divisible by the number of groups. Default: 1</td>
</tr>
</tbody>
</table>

**conv3d(input, weight, bias=NULL, stride=1, padding=0, dilation=1, groups=1) -> Tensor**

Applies a 3D convolution over an input image composed of several input planes.  
See `nn.conv3d()` for details and output shape.

**Examples**

```r
if (torch_is_installed()) {

  # filters = torch.randn(c(33, 16, 3, 3, 3))
  # inputs = torch.randn(c(20, 16, 50, 10, 20))
  # nnf.conv3d(inputs, filters)
}
```
torch_conv_tbc

**Description**

Conv_tbc

**Usage**

```python
torch_conv_tbc(self, weight, bias, pad = 0L)
```

**Arguments**

- `self`: NA input tensor of shape \((\text{sequence length} \times \text{batch} \times \text{in_channels})\)
- `weight`: NA filter of shape \((\text{kernel width} \times \text{in_channels} \times \text{out_channels})\)
- `bias`: NA bias of shape \((\text{out_channels})\)
- `pad`: NA number of timesteps to pad. Default: 0

**TEST**

Applies a 1-dimensional sequence convolution over an input sequence. Input and output dimensions are \((\text{Time, Batch, Channels})\) - hence TBC.

---

torch_conv_transpose1d

**Description**

Conv_transpose1d

**Usage**

```python
torch_conv_transpose1d(
    input,  
    weight,  
    bias = list(),  
    stride = 1L,  
    padding = 0L,  
    output_padding = 0L,  
    groups = 1L,  
    dilation = 1L  
)
```
**torch.conv_transpose2d**

**Description**

Conv_transpose2d

**Arguments**

- **input**
  - input tensor of shape \((\text{minibatch}, \text{in_channels}, iW)\)

- **weight**
  - filters of shape \((\text{in_channels}, \frac{\text{out_channels}}{\text{groups}}, kW)\)

- **bias**
  - optional bias of shape \((\text{out_channels})\). Default: NULL

- **stride**
  - the stride of the convolving kernel. Can be a single number or a tuple \((sW,)\). Default: 1

- **padding**
  - \(\text{dilation} \times (\text{kernel}\_\text{size} - 1) - \text{padding}\) zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple \((padW,)\). Default: 0

- **output_padding**
  - additional size added to one side of each dimension in the output shape. Can be a single number or a tuple \((out\_padW,)\). Default: 0

- **groups**
  - split input into groups, \(\text{in_channels}\) should be divisible by the number of groups. Default: 1

- **dilation**
  - the spacing between kernel elements. Can be a single number or a tuple \((dW,)\). Default: 1

**conv_transpose1d(input, weight, bias=NULL, stride=1, padding=0, output_padding=0, groups=1, dilation=1) -> Tensor**

Applies a 1D transposed convolution operator over an input signal composed of several input planes, sometimes also called “deconvolution”.

See **nn.conv_transpose1d()** for details and output shape.

**Examples**

```python
if (torch_is_installed()) {

  inputs = torch.randn(c(20, 16, 50))
  weights = torch.randn(c(16, 33, 5))
  nnf.conv_transpose1d(inputs, weights)
}
```
Usage

```python
torch_conv_transpose2d(
    input,
    weight,
    bias = list(),
    stride = 1L,
    padding = 0L,
    output_padding = 0L,
    groups = 1L,
    dilation = 1L
)
```

Arguments

- **input**: input tensor of shape `(minibatch, in_channels, iH, iW)`
- **weight**: filters of shape `(in_channels, out_channels // groups, kH, kW)`
- **bias**: optional bias of shape `(out_channels)`. Default: NULL
- **stride**: the stride of the convolving kernel. Can be a single number or a tuple `(sH, sW)`. Default: 1
- **padding**: dilation * (kernel_size - 1) - padding zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple `(padH, padW)`. Default: 0
- **output_padding**: additional size added to one side of each dimension in the output shape. Can be a single number or a tuple `(out_padH, out_padW)`. Default: 0
- **groups**: split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation**: the spacing between kernel elements. Can be a single number or a tuple `(dH, dW)`. Default: 1

```python
conv_transpose2d(input, weight, bias=NULL, stride=1, padding=0, output_padding=0, groups=1, dilation=1) -> Tensor
```

Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution".

See `nn.conv_transpose2d()` for details and output shape.

Examples

```python
if (torch_is_installed()) {

    # With square kernels and equal stride
    inputs = torch_randn(c(1, 4, 5, 5))
    weights = torch_randn(c(4, 8, 3, 3))
    nnf_conv_transpose2d(inputs, weights, padding=1)
}
```
torch_conv_transpose3d

Conv_transpose3d

Description

Conv_transpose3d

Usage

torch_conv_transpose3d(
    input,
    weight,
    bias = list(),
    stride = 1L,
    padding = 0L,
    output_padding = 0L,
    groups = 1L,
    dilation = 1L
)

Arguments

input input tensor of shape (minibatch, in_channels, iT, iH, iW)
weight filters of shape (in_channels, out_channels / groups, kT, kH, kW)
bias optional bias of shape (out_channels). Default: NULL
stride the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sw). Default: 1
padding dilation * (kernel_size - 1) - padding zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple (padT, padH, padW). Default: 0
output_padding additional size added to one side of each dimension in the output shape. Can be a single number or a tuple (out_padT, out_padH, out_padW). Default: 0
groups split input into groups, in_channels should be divisible by the number of groups. Default: 1
dilation the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1

conv_transpose3d(input, weight, bias=NULL, stride=1, padding=0, output_padding=0, groups=1, dilation=1) -> Tensor

Applies a 3D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution"

See nn_conv_transpose3d() for details and output shape.
Examples

```r
if (torch_is_installed()) {
  ## Not run:
  inputs = torch_randn(c(20, 16, 50, 10, 20))
  weights = torch_randn(c(16, 33, 3, 3, 3))
  nnf_conv_transpose3d(inputs, weights)

  ## End(Not run)
}
```

---

### torch_cos Cos

**Description**

Cos

**Usage**

```r
torch_cos(self)
```

**Arguments**

- `self` (Tensor) the input tensor.

**cos(input, out=NULL) -> Tensor**

Returns a new tensor with the cosine of the elements of `input`.

```r
out_i = cos(input_i)
```

**Examples**

```r
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_cos(a)
}
```
torch_cosh

Description
Cosh

Usage
torch_cosh(self)

Arguments
self (Tensor) the input tensor.

cosh(input, out=NULL) -> Tensor
Returns a new tensor with the hyperbolic cosine of the elements of input.

out_i = cosh(input_i)

Examples
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_cosh(a)
}

torch_cosine_similarity

Description
Cosine_similarity

Usage
torch_cosine_similarity(x1, x2, dim = 2L, eps = 1e-08)
Arguments

- **x1** (Tensor) First input.
- **x2** (Tensor) Second input (of size matching x1).
- **dim** (int, optional) Dimension of vectors. Default: 1
- **eps** (float, optional) Small value to avoid division by zero. Default: 1e-8

**cosine_similarity(x1, x2, dim=1, eps=1e-8) -> Tensor**

Returns cosine similarity between x1 and x2, computed along dim.

\[
similarity = \frac{x_1 \cdot x_2}{\max(||x_1||_2 \cdot ||x_2||_2, \epsilon)}
\]

**Examples**

```python
if (torch_is_installed()) {
    input1 = torch_randn(c(100, 128))
    input2 = torch_randn(c(100, 128))
    output = torch_cosine_similarity(input1, input2)
    output
}
```

---

**torch_count_nonzero**

Count_nonzero

**Description**

Count_nonzero

**Usage**

torch_count_nonzero(self, dim = NULL)

**Arguments**

- **self** (Tensor) the input tensor.
- **dim** (int or tuple of ints, optional) Dim or tuple of dims along which to count non-zeros.

**count_nonzero(input, dim=None) -> Tensor**

Counts the number of non-zero values in the tensor input along the given dim. If no dim is specified then all non-zeros in the tensor are counted.
Examples

```r
if (torch_is_installed()) {

  x <- torch_zeros(3,3)
  x[torch_randn(3,3) > 0.5] = 1
  x
  torch_count_nonzero(x)
  torch_count_nonzero(x, dim=1)
}
```

### torch_cross

<table>
<thead>
<tr>
<th>Description</th>
<th>Cross</th>
</tr>
</thead>
</table>

**Usage**

```r
torch_cross(self, other, dim = NULL)
```

**Arguments**

- `self` (Tensor): the input tensor.
- `other` (Tensor): the second input tensor
- `dim` (int, optional): the dimension to take the cross-product in.

**cross(input, other, dim=-1, out=NULL) -> Tensor**

Returns the cross product of vectors in dimension `dim` of `input` and `other`.

- `input` and `other` must have the same size, and the size of their `dim` dimension should be 3.
- If `dim` is not given, it defaults to the first dimension found with the size 3.

**Examples**

```r
if (torch_is_installed()) {

  a = torch_randn(c(4, 3))
  a
  b = torch_randn(c(4, 3))
  b
  torch_cross(a, b, dim=2)
  torch_cross(a, b)
}
```
torch_cummax  

Description
Cummax

Usage
torch_cummax(self, dim)

Arguments

self (Tensor) the input tensor.
dim (int) the dimension to do the operation over

cummax(input, dim) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the cumulative maximum of elements of input in the dimension dim. And indices is the index location of each maximum value found in the dimension dim.

\[ y_i = \max(x_1, x_2, x_3, \ldots, x_i) \]

Examples
if (torch_is_installed()) {

    a = torch_randn(c(10))
a  torch_cummax(a, dim=1)
}

torch_cummin  

Description
Cummin

Usage
torch_cummin(self, dim)
torch_cumprod

Arguments

self (Tensor) the input tensor.
dim (int) the dimension to do the operation over

cummin(input, dim) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the cumulative minimum of elements of input in the dimension dim. And indices is the index location of each maximum value found in the dimension dim.

\[ y_i = \min(x_1, x_2, x_3, \ldots, x_i) \]

Examples

if (torch_is_installed()) {
    a = torch_randn(c(10))
a
torch_cummin(a, dim=1)
}

torch_cumprod (Cumprod)

Description

Cumprod

Usage

torch_cumprod(self, dim, dtype = NULL)

Arguments

self (Tensor) the input tensor.
dim (int) the dimension to do the operation over
dtype (torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

cumprod(input, dim, out=NULL, dtype=NULL) -> Tensor

Returns the cumulative product of elements of input in the dimension dim.

For example, if input is a vector of size N, the result will also be a vector of size N, with elements.

\[ y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i \]
Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(10))
  a
  torch_cumprod(a, dim=1)
}
```

torch_cumsum

### Description

Cumsum

### Usage

```r
torch_cumsum(self, dim, dtype = NULL)
```

### Arguments

- **self** (Tensor): the input tensor.
- **dim** (int): the dimension to do the operation over
- **dtype** (torch.dtype, optional): the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

### cumsum(input, dim, out=NULL, dtype=NULL) -> Tensor

Returns the cumulative sum of elements of input in the dimension dim.

For example, if input is a vector of size N, the result will also be a vector of size N, with elements.

\[ y_i = x_1 + x_2 + x_3 + \cdots + x_i \]

### Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(10))
  a
  torch_cumsum(a, dim=1)
}
```
torch_deg2rad

### Description

Deg2rad

### Usage

`torch_deg2rad(self)`

### Arguments

- `self` (Tensor) the input tensor.

### Example

```python
if (torch_is_installed()) {
    a <- torch_tensor(rbind(c(180.0, -180.0), c(360.0, -360.0), c(90.0, -90.0)))
    torch_deg2rad(a)
}
```

torch_dequantize

### Description

Dequantize

### Usage

`torch_dequantize(tensor)`

### Arguments

- `tensor` (Tensor) A quantized Tensor or a list of quantized tensors

### Example

```python
dequantize(tensor) -> Tensor

Returns a new tensor with each of the elements of input converted from angles in degrees to radians.
```

### Example

```python
if (torch_is_installed()) {
    a <- torch_tensor(rbind(c(180.0, -180.0), c(360.0, -360.0), c(90.0, -90.0)))
    torch_deg2rad(a)
}
```
dequantize(tensors) -> sequence of Tensors

Given a list of quantized Tensors, dequantize them and return a list of fp32 Tensors

```
torch_det  Det

Description
  Det

Usage
  torch_det(self)

Arguments
  self  (Tensor) the input tensor of size (*, n, n) where * is zero or more batch dimensions.

det(input) -> Tensor

Calculates determinant of a square matrix or batches of square matrices.

Note
  Backward through 'det' internally uses SVD results when 'input' is not invertible. In this case, double backward through 'det' will be unstable in when 'input' doesn't have distinct singular values. See '~torch.svd' for details.

Examples
  if (torch_is_installed()) {
    A = torch_randn(c(3, 3))
    torch_det(A)
    A = torch_randn(c(3, 2, 2))
    A
    A$det()
  }
```
torch_device

Description

A torch_device is an object representing the device on which a torch_tensor is or will be allocated.

Usage

torch_device(type, index = NULL)

Arguments

- **type** (character) a device type “cuda” or “cpu”
- **index** (integer) optional device ordinal for the device type. If the device ordinal is not present, this object will always represent the current device for the device type, even after torch_cuda_set_device() is called; e.g., a torch_tensor constructed with device 'cuda' is equivalent to 'cuda:X' where X is the result of torch_cuda_current_device(). A torch_device can be constructed via a string or via a string and device ordinal

Examples

```r
if (torch_is_installed()) {

    # Via string
    torch_device("cuda:1")
    torch_device("cpu")
    torch_device("cuda") # current cuda device

    # Via string and device ordinal
    torch_device("cuda", 0)
    torch_device("cpu", 0)
}
```

torch_diag

Description

Diag

Usage

torch_diag(self, diagonal = 0L)
torch_diagflat

**Arguments**

- **self** *(Tensor)* the input tensor.
- **diagonal** *(int, optional)* the diagonal to consider

**diag(input, diagonal=0, out=NULL) -> Tensor**

- If `input` is a vector (1-D tensor), then returns a 2-D square tensor with the elements of `input` as the diagonal.
- If `input` is a matrix (2-D tensor), then returns a 1-D tensor with the diagonal elements of `input`.

The argument `diagonal` controls which diagonal to consider:

- If `diagonal = 0`, it is the main diagonal.
- If `diagonal > 0`, it is above the main diagonal.
- If `diagonal < 0`, it is below the main diagonal.

---

**torch_diagflat**  
Diagflat

**Description**

Diagflat

**Usage**

torch_diagflat(self, offset = 0L)

**Arguments**

- **self** *(Tensor)* the input tensor.
- **offset** *(int, optional)* the diagonal to consider. Default: 0 (main diagonal).

**diagflat(input, offset=0) -> Tensor**

- If `input` is a vector (1-D tensor), then returns a 2-D square tensor with the elements of `input` as the diagonal.
- If `input` is a tensor with more than one dimension, then returns a 2-D tensor with diagonal elements equal to a flattened `input`.

The argument `offset` controls which diagonal to consider:

- If `offset = 0`, it is the main diagonal.
- If `offset > 0`, it is above the main diagonal.
- If `offset < 0`, it is below the main diagonal.
torch_diagonal

Examples
if (torch_is_installed()) {
    a = torch_randn(c(3))
    a
    torch.diagflat(a)
    torch.diagflat(a, 1)
    a = torch_randn(c(2, 2))
    a
    torch.diagflat(a)
}

torch_diagonal  Diagonal

Description
Diagonal

Usage
torch_diagonal(self, outdim, dim1 = 1L, dim2 = 2L, offset = 0L)

Arguments

self (Tensor) the input tensor. Must be at least 2-dimensional.
outdim dimension name if self is a named tensor.
dim1 (int, optional) first dimension with respect to which to take diagonal. Default: 0.
dim2 (int, optional) second dimension with respect to which to take diagonal. Default: 1.
offset (int, optional) which diagonal to consider. Default: 0 (main diagonal).

diagonal(input, offset=0, dim1=0, dim2=1) -> Tensor

Returns a partial view of input with the its diagonal elements with respect to dim1 and dim2 appended as a dimension at the end of the shape.
The argument offset controls which diagonal to consider:

- If offset = 0, it is the main diagonal.
- If offset > 0, it is above the main diagonal.
- If offset < 0, it is below the main diagonal.

Applying torch_diag_embed to the output of this function with the same arguments yields a diagonal matrix with the diagonal entries of the input. However, torch_diag_embed has different default dimensions, so those need to be explicitly specified.
Examples

```r
if (torch_is_installed()) {

  a = torch_randn(c(3, 3))
a  
torch_diagonal(a, offset = 0)
torch_diagonal(a, offset = 1)
  x = torch_randn(c(2, 5, 4, 2))
torch_diagonal(x, offset=-1, dim1=1, dim2=2)
}
```

**Description**

**Diag_embed**

**Usage**

`torch_diag_embed(self, offset = 0L, dim1 = -2L, dim2 = -1L)`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>self</code></td>
<td>Tensor</td>
<td></td>
</tr>
<tr>
<td><code>offset</code></td>
<td>(int, optional) which diagonal to consider. Default: 0 (main diagonal).</td>
<td>0</td>
</tr>
<tr>
<td><code>dim1</code></td>
<td>(int, optional) first dimension with respect to which to take diagonal. Default: -2.</td>
<td>-2</td>
</tr>
<tr>
<td><code>dim2</code></td>
<td>(int, optional) second dimension with respect to which to take diagonal. Default: -1.</td>
<td>-1</td>
</tr>
</tbody>
</table>

`diag_embed(input, offset=0, dim1=-2, dim2=-1) -> Tensor`

Creates a tensor whose diagonals of certain 2D planes (specified by `dim1` and `dim2`) are filled by `input`. To facilitate creating batched diagonal matrices, the 2D planes formed by the last two dimensions of the returned tensor are chosen by default.

The argument `offset` controls which diagonal to consider:

- If `offset = 0`, it is the main diagonal.
- If `offset > 0`, it is above the main diagonal.
- If `offset < 0`, it is below the main diagonal.

The size of the new matrix will be calculated to make the specified diagonal of the size of the last input dimension. Note that for `offset` other than 0, the order of `dim1` and `dim2` matters. Exchanging them is equivalent to changing the sign of `offset`.

Applying `torch_diagonal` to the output of this function with the same arguments yields a matrix identical to `input`. However, `torch_diagonal` has different default dimensions, so those need to be explicitly specified.
torch_diff

Examples

if (torch_is_installed()) {
    a = torch_randn(c(2, 3))
    torch_diag_embed(a)
    torch_diag_embed(a, offset=1, dim1=1, dim2=3)
}

torch_diff

Computes the n-th forward difference along the given dimension.

Description

The first-order differences are given by \( \text{out}[i] = \text{input}[i + 1] - \text{input}[i] \). Higher-order differences are calculated by using torch_diff() recursively.

Usage

torch_diff(self, n = 1L, dim = -1L, prepend = list(), append = list())

Arguments

self the tensor to compute the differences on
n the number of times to recursively compute the difference
dim the dimension to compute the difference along. Default is the last dimension.
prepend values to prepend to input along dim before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input’s shape except on dim.
append values to append to input along dim before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input’s shape except on dim.

Note

Only \( n = 1 \) is currently supported

Examples

if (torch_is_installed()) {
    a <- torch_tensor(c(1, 2, 3))
    torch_diff(a)

    b <- torch_tensor(c(4, 5))
    torch_diff(a, append = b)

    c <- torch_tensor(rbind(c(1, 2, 3), c(3, 4, 5)))
    torch_diff(c, dim = 1)
torch_diff(c, dim = 2)

}{
torch_digamma

Description

Digamma

Usage

torch_digamma(self)

Arguments

self (Tensor) the tensor to compute the digamma function on

digamma(input, out=NULL) -> Tensor

Computes the logarithmic derivative of the gamma function on input.

\[
\psi(x) = \frac{d}{dx} \ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)}
\]

Examples

if (torch_is_installed()) {

    a = torch_tensor(c(1, 0.5))
    torch_digamma(a)
}

torch_dist

Description

Dist

Usage

torch_dist(self, other, p = 2L)
torch_div

Arguments

- **self** (Tensor) the input tensor.
- **other** (Tensor) the Right-hand-side input tensor
- **p** (float, optional) the norm to be computed

**dist(input, other, p=2) -> Tensor**

Returns the p-norm of (input - other)

The shapes of input and other must be broadcastable.

Examples

```python
if (torch_is_installed()) {
    x = torch.randn(c(4))
    x
    y = torch.randn(c(4))
    y
    torch_dist(x, y, 3.5)
    torch_dist(x, y, 3)
    torch_dist(x, y, 0)
    torch_dist(x, y, 1)
}
```

---

torch_div **Div**

Description

**Div**

Usage

```python
torch_div(self, other, rounding_mode)
```

Arguments

- **self** (Tensor) the input tensor.
- **other** (Number) the number to be divided to each element of input
- **rounding_mode** (str, optional) – Type of rounding applied to the result:
  - NULL - default behavior. Performs no rounding and, if both input and other are integer types, promotes the inputs to the default scalar type. Equivalent to true division in Python (the / operator) and NumPy’s np.true_divide.
  - "trunc" - rounds the results of the division towards zero. Equivalent to C-style integer division.
  - "floor" - rounds the results of the division down. Equivalent to floor division in Python (the // operator) and NumPy's np.floor_divide.


div(input, other, out=NULL) -> Tensor

Divides each element of the input input with the scalar other and returns a new resulting tensor. Each element of the tensor input is divided by each element of the tensor other. The resulting tensor is returned.

\[
\text{out}_i = \frac{\text{input}_i}{\text{other}_i}
\]

The shapes of input and other must be broadcastable. If the torch_dtype of input and other differ, the torch_dtype of the result tensor is determined following rules described in the type promotion documentation. If out is specified, the result must be castable to the torch_dtype of the specified output tensor. Integral division by zero leads to undefined behavior.

Warning

Integer division using div is deprecated, and in a future release div will perform true division like torch_true_divide(). Use torch_floor_divide() to perform integer division, instead.

\[
\text{out}_i = \frac{\text{input}_i}{\text{other}}
\]

If the torch_dtype of input and other differ, the torch_dtype of the result tensor is determined following rules described in the type promotion documentation. If out is specified, the result must be castable to the torch_dtype of the specified output tensor. Integral division by zero leads to undefined behavior.

Examples

if (torch_is_installed()) {
    a = torch_randn(c(5))
atorch_div(a, 0.5)

    a = torch_randn(c(4, 4))
a b = torch_randn(c(4))btorch_div(a, b)
}

---

torch_divide 

Divide

Description

Divide
torch_dot

Usage

torch_dot(self, tensor)

Arguments

self (Tensor) the input tensor.
tensor (Tensor) the other input tensor

dot(input, tensor) -> Tensor

Computes the dot product (inner product) of two tensors.

Note

This function does not broadcast.
Examples

```r
if (torch_is_installed()) {
  torch_dot(torch_tensor(c(2, 3)), torch_tensor(c(2, 1)))
}
```

### torch_dstack

**Dstack**

**Description**

Dstack

**Usage**

```r
torch_dstack(tensors)
```

**Arguments**

- `tensors` (sequence of Tensors) sequence of tensors to concatenate

`dstack(tensors, *, out=None) -> Tensor`

Stack tensors in sequence depthwise (along third axis).

This is equivalent to concatenation along the third axis after 1-D and 2-D tensors have been reshaped by `torch_atleast_3d()`.

**Examples**

```r
if (torch_is_installed()) {
    a <- torch_tensor(c(1, 2, 3))
    b <- torch_tensor(c(4, 5, 6))
    torch_dstack(list(a, b))
    a <- torch_tensor(rbind(1, 2, 3))
    b <- torch_tensor(rbind(4, 5, 6))
    torch_dstack(list(a, b))
}
```
torch_dtype

<table>
<thead>
<tr>
<th>torch_dtype</th>
<th>Torch data types</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Returns the correspondent data type.

**Usage**

torch_float32()
torch_float()
torch_float64()
torch_double()
torch_cfloat32()
torch_chalf()
torch_cfloat()
torch_cfloat64()
torch_cdouble()
torch_cfloat128()
torch_float16()
torch_half()
torch_uint8()
torch_int8()
torch_int16()
torch_short()
torch_int32()
torch_int()
torch_int64()
torch_long()
torch_bool()
torch_quint8()
torch_qint8()
torch_qint32()

torch_eig

Description
Eig

Arguments

self (Tensor) the square matrix of shape \((n \times n)\) for which the eigenvalues and eigenvectors will be computed

eigenvectors (bool) TRUE to compute both eigenvalues and eigenvectors; otherwise, only eigenvalues will be computed

eig(input, eigenvectors=False, out=NULL) -> (Tensor, Tensor)
   Computes the eigenvalues and eigenvectors of a real square matrix.

Einsum

Description
Einsum

Usage
torch_einsum(equation, tensors, path = NULL)
**torch_einsum**

### Arguments

- **equation** (string) The equation is given in terms of lower case letters (indices) to be associated with each dimension of the operands and result. The left hand side lists the operands dimensions, separated by commas. There should be one index letter per tensor dimension. The right hand side follows after `->` and gives the indices for the output. If the `->` and right hand side are omitted, it implicitly defined as the alphabetically sorted list of all indices appearing exactly once in the left hand side. The indices not appearing in the output are summed over after multiplying the operands entries. If an index appears several times for the same operand, a diagonal is taken. Ellipses ... represent a fixed number of dimensions. If the right hand side is inferred, the ellipsis dimensions are at the beginning of the output.

- **tensors** (Tensor) The operands to compute the Einstein sum of.

- **path** (int) This function uses opt_einsum to speed up computation or to consume less memory by optimizing contraction order. This optimization occurs when there are at least three inputs, since the order does not matter otherwise. Note that finding the optimal path is an NP-hard problem, thus, opt_einsum relies on different heuristics to achieve near-optimal results. If opt_einsum is not available, the default order is to contract from left to right. The path argument is used to changed that default, but it should only be set by advanced users.

### einsum(equation, *operands) -> Tensor

This function provides a way of computing multilinear expressions (i.e. sums of products) using the Einstein summation convention.

### Examples

```python
if (torch_is_installed()) {
    x = torch_randn(c(5))
    y = torch_randn(c(4))
    torch_einsum('i,j->ij', x, y) # outer product
    A = torch_randn(c(3,5,4))
    l = torch_randn(c(2,5))
    r = torch_randn(c(2,4))
    torch_einsum('bn,anm,bm->ba', l, A, r) # compare torch.nn.functional.bilinear
    As = torch_randn(c(3,2,5))
    Bs = torch_randn(c(3,5,4))
    torch_einsum('bij,bjk->bik', As, Bs) # batch matrix multiplication
    A = torch.rand(c(3, 3))
    torch_einsum('ii->i', A) # diagonal
    A = torch.rand(c(4, 3, 3))
    torch_einsum('...ii->...i', A) # batch diagonal
    A = torch.rand(c(2, 3, 4, 5))
    torch_einsum('...ij->...ji', A) # batch permute
}
```
torch_empty

Description
Empty

Usage
torch_empty(
  ..., 
  names = NULL, 
  dtype = NULL, 
  layout = NULL, 
  device = NULL, 
  requires_grad = FALSE
)

Arguments
...
  a sequence of integers defining the shape of the output tensor.

names
  optional character vector naming each dimension.

dtype
  (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).

layout
  (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.

device
  (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad
  (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

empty(*size, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False, pin_memory=False) -> Tensor

Returns a tensor filled with uninitialized data. The shape of the tensor is defined by the variable argument size.

Examples
if (torch_is_installed()) {
  torch_empty(c(2, 3))
}
torch_empty_like

Description
Empty_like

Usage
torch_empty_like(
    input,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)

Arguments
input (Tensor) the size of input will determine size of the output tensor.
dtype (torch.dtype, optional) the desired data type of returned Tensor. Default: if NULL, defaults to the dtype of input.
layout (torch.layout, optional) the desired layout of returned tensor. Default: if NULL, defaults to the layout of input.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, defaults to the device of input.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
memory_format (torch.memory_format, optional) the desired memory format of returned Tensor. Default: torch_preserve_format.

empty_like(input, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, memory_format=torch.preserve_format) -> Tensor

Returns an uninitialized tensor with the same size as input. torch_empty_like(input) is equivalent to torch_empty(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Examples
if (torch_is_installed()) {
    torch_empty(list(2,3), dtype = torch_int64())
}
torch_empty_strided

Empty_strided

Description

Empty_strided

Usage

torch_empty_strided(
    size,
    stride,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    pin_memory = FALSE
)

Arguments

size (tuple of ints) the shape of the output tensor
stride (tuple of ints) the strides of the output tensor
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
pin_memory (bool, optional) If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: FALSE.

empty_strided(size, stride, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, pin_memory=False) -> Tensor

Returns a tensor filled with uninitialized data. The shape and strides of the tensor is defined by the variable argument size and stride respectively. torch_empty_strided(size, stride) is equivalent to torch_empty(size).as_strided(size, stride).

Warning

More than one element of the created tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensors, please clone them first.
**torch_eq**

**Examples**

```python
if (torch_is_installed()) {
    a = torch_empty_strided(list(2, 3), list(1, 2))
    a
    a$stride(1)
    a$size(1)
}
```

---

**torch_eq**  
*Eq*

**Description**

Eq

**Usage**

```python
torch_eq(self, other)
```

**Arguments**

- **self**  
  (Tensor) the tensor to compare
- **other**  
  (Tensor or float) the tensor or value to compare Must be a ByteTensor

**eq**(*input, other, out=NULL*)  
-> Tensor

Computes element-wise equality

The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

**Examples**

```python
if (torch_is_installed()) {
    torch_eq(torch_tensor(c(1,2,3,4)), torch_tensor(c(1, 3, 2, 4)))
}
```
torch_equal

Description
Equal

Usage
torch_equal(self, other)

Arguments
self the input tensor
other the other input tensor

equal(input, other) -> bool
TRUE if two tensors have the same size and elements, FALSE otherwise.

Examples
if (torch_is_installed()) {
    torch_equal(torch_tensor(c(1, 2)), torch_tensor(c(1, 2)))
}

torch_erf

Description
Erf

Usage
torch_erf(self)

Arguments
self (Tensor) the input tensor.

erf(input, out=NULL) -> Tensor
Computes the error function of each element. The error function is defined as follows:

\[
erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, dt
\]
Examples

```r
if (torch_is_installed()) {
    torch_erfc(torch_tensor(c(0, -1., 10.)))
}
```

Description

Erfc

Usage

torch_erfc(self)

Arguments

self (Tensor) the input tensor.

erfc(input, out=NULL) -> Tensor

Computes the complementary error function of each element of input. The complementary error function is defined as follows:

\[
erfc(x) = 1 - \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt
\]

Examples

```r
if (torch_is_installed()) {
    torch_erfc(torch_tensor(c(0, -1., 10.)))
}
```
**torch_erfinv**  
*Erfinv*

**Description**
Erfinv

**Usage**
torch_erfinv(self)

**Arguments**

self  
(Tensor) the input tensor.

erfinv(input, out=NULL) -> Tensor

Computes the inverse error function of each element of input. The inverse error function is defined in the range $(-1, 1)$ as:

$$\text{erfinv}(\text{erf}(x)) = x$$

**Examples**

```r
if (torch_is_installed()) {
    torch_erfinv(torch_tensor(c(0, 0.5, -1.)))
}
```

**torch_exp**  
*Exp*

**Description**
Exp

**Usage**
torch_exp(self)

**Arguments**

self  
(Tensor) the input tensor.
exp(input, out=NULL) -> Tensor

Returns a new tensor with the exponential of the elements of the input tensor input.

\[ y_i = e^{x_i} \]

Examples

```r
if (torch_is_installed()) {
  torch_exp(torch_tensor(c(0, log(2))))
}
```

---

torch_exp2 

Exp2

Description

Exp2

Usage

torch_exp2(self)

Arguments

self (Tensor) the input tensor.

exp2(input, *, out=None) -> Tensor

Computes the base two exponential function of input.

\[ y_i = 2^{x_i} \]

Examples

```r
if (torch_is_installed()) {
  torch_exp2(torch_tensor(c(0, log2(2.), 3, 4)))
}
```
torch_expm1  \( \text{Expm1} \)

**Description**

\( \text{Expm1} \)

**Usage**

\( \text{torch_expm1}(\text{self}) \)

**Arguments**

self  (Tensor) the input tensor.

\( \text{expm1}(\text{input, out=NULL}) \rightarrow \text{Tensor} \)

Returns a new tensor with the exponential of the elements minus 1 of \( \text{input} \).

\[ y_i = e^{x_i} - 1 \]

**Examples**

```r
if (torch_is_installed()) {
    torch_expm1(torch_tensor(c(0, log(2))))
}
```

torch_eye  \( \text{Eye} \)

**Description**

\( \text{Eye} \)

**Usage**

\( \text{torch_eye}(\text{n, m=n, dtype=NULL, layout=NULL, device=NULL, requires_grad=FALSE}) \)

\( \text{torch_eye}(\text{n}) \)

Returns a new tensor with the same size as \( \text{input} \) with ones on the diagonal and zeros elsewhere.

\[ y_{ii} = 1 \]
torch_fft.fft

**Arguments**

- **n** (int) the number of rows
- **m** (int, optional) the number of columns with default being n
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
- **layout** (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

**eye(n, m=NULL, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False)**

-> Tensor

Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.

**Examples**

```cpp
if (torch_is_installed()) {

    torch_eye(3)
}
```

---

**torch_fft.fft**

**Fft**

**Description**

Computes the one dimensional discrete Fourier transform of input.

**Usage**

torch_fft.fft(self, n = NULL, dim = -1L, norm = NULL)

**Arguments**

- **self** (Tensor) the input tensor
- **n** (int) Signal length. If given, the input will either be zero-padded or trimmed to this length before computing the FFT.
- **dim** (int, optional) The dimension along which to take the one dimensional FFT.
- **norm** (str, optional) Normalization mode. For the forward transform, these correspond to:
  - "forward" - normalize by 1/n
• "backward" - no normalization
• "ortho" - normalize by 1/sqrt(n) (making the FFT orthonormal) Calling the backward transform (ifft()) with the same normalization mode will apply an overall normalization of 1/n between the two transforms. This is required to make IFFT the exact inverse. Default is "backward" (no normalization).

**Note**

The Fourier domain representation of any real signal satisfies the Hermitian property: \( X[i] = \text{conj}(X[-i]) \). This function always returns both the positive and negative frequency terms even though, for real inputs, the negative frequencies are redundant. rfft() returns the more compact one-sided representation where only the positive frequencies are returned.

**Examples**

```r
if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 3)
  t
  torch_fft_fft(t, norm = "backward")
}
```

---

**torch_fft_fftfreq**

**fftfreq**

**Description**

Computes the discrete Fourier Transform sample frequencies for a signal of size \( n \).

**Usage**

```r
torch_fft_fftfreq(
  n,
  d = 1,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE
)
```

**Arguments**

- **n** (integer) – the FFT length
- **d** (float, optional) – the sampling length scale. The spacing between individual samples of the FFT input. The default assumes unit spacing, dividing that result by the actual spacing gives the result in physical frequency units.
- **dtype** (default: `torch_get_default_dtype()`) the desired data type of returned tensor.
torch_fft_ifft

Description
Computes the one dimensional inverse discrete Fourier transform of input.

Usage
torch_fft_ifft(self, n = NULL, dim = -1L, norm = NULL)

Arguments
- self (Tensor) the input tensor
- n (int, optional) – Signal length. If given, the input will either be zero-padded or trimmed to this length before computing the IFFT.
- dim (int, optional) – The dimension along which to take the one dimensional IFFT.
- norm (str, optional) – Normalization mode. For the backward transform, these correspond to:
  - "forward" - no normalization
  - "backward" - normalize by 1/n
  - "ortho" - normalize by 1/sqrt(n) (making the IFFT orthonormal) Calling the forward transform with the same normalization mode will apply an overall normalization of 1/n between the two transforms. This is required to make ifft() the exact inverse. Default is "backward" (normalize by 1/n).
torch_fft_irfft

Examples

```r
if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 3)
  x <- torch_fft_fft(t, norm = "backward")
  torch_fft_ifft(x)
}
```

Description

Computes the inverse of `torch_fft_rfft()`. Input is interpreted as a one-sided Hermitian signal in the Fourier domain, as produced by `torch_fft_rfft()`. By the Hermitian property, the output will be real-valued.

Usage

```r
torch_fft_irfft(self, n = NULL, dim = -1L, norm = NULL)
```

Arguments

- **self**: (Tensor) the input tensor representing a half-Hermitian signal
- **n**: (int) Output signal length. This determines the length of the output signal. If given, the input will either be zero-padded or trimmed to this length before computing the real IFFT. Defaults to even output: \( n=2 \times (\text{input.size(dim)} - 1) \).
- **dim**: (int, optional) – The dimension along which to take the one dimensional real IFFT.
- **norm**: (str, optional) – Normalization mode. For the backward transform, these correspond to:
  - "forward" - no normalization
  - "backward" - normalize by \( 1/n \)
  - "ortho" - normalize by \( 1/\sqrt{n} \) (making the real IFFT orthonormal). Calling the forward transform (`torch_fft_rfft()`) with the same normalization mode will apply an overall normalization of \( 1/n \) between the two transforms. This is required to make irfft() the exact inverse. Default is "backward" (normalize by \( 1/n \)).
Note

Some input frequencies must be real-valued to satisfy the Hermitian property. In these cases the imaginary component will be ignored. For example, any imaginary component in the zero-frequency term cannot be represented in a real output and so will always be ignored.

The correct interpretation of the Hermitian input depends on the length of the original data, as given by n. This is because each input shape could correspond to either an odd or even length signal. By default, the signal is assumed to be even length and odd signals will not round-trip properly. So, it is recommended to always pass the signal length n.

Examples

```r
if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 4)
  x <- torch_fft_rfft(t)
  torch_fft_irfft(x)
  torch_fft_irfft(x, n = t$ numel())
}
```

Description

Computes the one dimensional Fourier transform of real-valued input.

Usage

```r
torch_fft_rfft(self, n = NULL, dim = -1L, norm = NULL)
```

Arguments

- `self` (Tensor) the real input tensor
- `n` (int) Signal length. If given, the input will either be zero-padded or trimmed to this length before computing the real FFT.
- `dim` (int, optional) – The dimension along which to take the one dimensional real FFT.
- `norm` norm (str, optional) – Normalization mode. For the forward transform, these correspond to:
  - "forward" - normalize by 1/n
  - "backward" - no normalization
  - "ortho" - normalize by 1/sqrt(n) (making the FFT orthonormal) Calling the backward transform (`torch_fft_irfft()`) with the same normalization mode will apply an overall normalization of 1/n between the two transforms. This is required to make irfft() the exact inverse. Default is "backward" (no normalization).
Details

The FFT of a real signal is Hermitian-symmetric, \(X[i] = \text{conj}(X[-i])\) so the output contains only the positive frequencies below the Nyquist frequency. To compute the full output, use `torch_fft_fft()`.

Examples

```r
if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 3)
  torch_fft_rfft(t)
}
```

---

`torch_finfo`  
Floating point type info

Description

A list that represents the numerical properties of a floating point torch.dtype

Usage

`torch_finfo(dtype)`

Arguments

dtype  
dtype to check information

---

`torch_fix`  
Fix

Description

Fix

Usage

`torch_fix(self)`

Arguments

self  
(Tensor) the input tensor.

`fix(input, *, out=None) -> Tensor`

Alias for `torch_trunc()`
torch_flatten

Description
Flatten

Usage
torch_flatten(self, dims, start_dim = 1L, end_dim = -1L, out_dim)

Arguments
- self (Tensor) the input tensor.
- dims if tensor is named you can pass the name of the dimensions to flatten
- start_dim (int) the first dim to flatten
- end_dim (int) the last dim to flatten
- out_dim the name of the resulting dimension if a named tensor.

flatten(input, start_dim=0, end_dim=-1) -> Tensor
Flattens a contiguous range of dims in a tensor.

Examples
if (torch_is_installed()) {
    t = torch_tensor(matrix(c(1, 2), ncol = 2))
    torch_flatten(t)
    torch_flatten(t, start_dim=2)
}

torch_flip

Description
Flip

Usage
torch_flip(self, dims)
Arguments

- **self** *(Tensor)* the input tensor.
- **dims** *(a list or tuple)* axis to flip on

**flip(input, dims) -> Tensor**

Reverse the order of a n-D tensor along given axis in dims.

**Examples**

```python
if (torch_is_installed()) {
    x <- torch_arange(1, 8)$view(c(2, 2, 2))
    x
    torch_flip(x, c(1, 2))
}
```

---

**torch_fliplr**

*Fliplr*

**Description**

Fliplr

**Usage**

`torch_fliplr(self)`

**Arguments**

- **self** *(Tensor)* Must be at least 2-dimensional.

**fliplr(input) -> Tensor**

Flip array in the left/right direction, returning a new tensor.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Note**

Equivalent to `input[:, -1]`. Requires the array to be at least 2-D.
torch_flipud

Examples
if (torch_is_installed()) {

  x <- torch_arange(start = 1, end = 4)$view(c(2, 2))
  x
  torch_fliplr(x)
}

Description
Flipud

Usage
torch_flipud(self)

Arguments
self  (Tensor) Must be at least 1-dimensional.

flipud(input) -> Tensor
Flip array in the up/down direction, returning a new tensor.
Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a
different order than before.

Note
Equivalent to input[-1,]. Requires the array to be at least 1-D.

Examples
if (torch_is_installed()) {

  x <- torch_arange(start = 1, end = 4)$view(c(2, 2))
  x
  torch_flipud(x)
}
torch_floor

Floor

Description

Floor

Usage

torch_floor(self)

Arguments

self (Tensor) the input tensor.

floor(input, out=NULL) -> Tensor

Returns a new tensor with the floor of the elements of input, the largest integer less than or equal to each element.

out_i = ⌊input_i⌋

Examples

if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_floor(a)
}

torch_floor_divide

Floor_divide

Description

Floor_divide

Usage

torch_floor_divide(self, other)

Arguments

self (Tensor) the numerator tensor
other (Tensor or Scalar) the denominator
floor_divide(input, other, out=NULL) -> Tensor

Return the division of the inputs rounded down to the nearest integer. See torch_div for type promotion and broadcasting rules.

\[ \text{out}_i = \left\lfloor \frac{\text{input}_i}{\text{other}_i} \right\rfloor \]

Examples

```python
if (torch_is_installed()) {
    a = torch_tensor(c(4.0, 3.0))
    b = torch_tensor(c(2.0, 2.0))
    torch_floor_divide(a, b)
    torch_floor_divide(a, 1.4)
}
```

torch_fmod

Description

Fmod

Usage

torch_fmod(self, other)

Arguments

- **self** (Tensor) the dividend
- **other** (Tensor or float) the divisor, which may be either a number or a tensor of the same shape as the dividend

fmod(input, other, out=NULL) -> Tensor

Computes the element-wise remainder of division.

The dividend and divisor may contain both for integer and floating point numbers. The remainder has the same sign as the dividend input.

When other is a tensor, the shapes of input and other must be broadcastable.

Examples

```python
if (torch_is_installed()) {
    torch_fmod(torch_tensor(c(-3., -2, -1, 1, 2, 3)), 2)
    torch_fmod(torch_tensor(c(1., 2, 3, 4)), 1.5)
}
```
torch_frac

Description
Frac

Usage
torch_frac(self)

Arguments
self the input tensor.

frac(input, out=NULL) -> Tensor
Computes the fractional portion of each element in input.
\[ \text{out}_i = \text{input}_i - \lfloor |\text{input}_i| \rfloor \times \text{sgn}(\text{input}_i) \]

Examples
if (torch_is_installed()) {
    torch_frac(torch_tensor(c(1, 2.5, -3.2)))
}

torch_full

Description
Full

Usage
torch_full(
    size,
    fill_value,
    names = NULL,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)
torch_full_like

Arguments

size (int...) a list, tuple, or torch_Size of integers defining the shape of the output tensor.

fill_value NA the number to fill the output tensor with.

names optional names of the dimensions

dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).

layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.

device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

full(size, fill_value, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Returns a tensor of size size filled with fill_value.

Warning

In PyTorch 1.5 a bool or integral fill_value will produce a warning if dtype or out are not set. In a future PyTorch release, when dtype and out are not set a bool fill_value will return a tensor of torch.bool dtype, and an integral fill_value will return a tensor of torch.long dtype.

Examples

if (torch_is_installed()) {

    torch_full(list(2, 3), 3.141592)
}

Description

Full_like
Usage

```
torch_full_like(
    input,
    fill_value,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)
```

Arguments

- `input`: (Tensor) the size of `input` will determine size of the output tensor.
- `fill_value`: the number to fill the output tensor with.
- `dtype`: (torch.dtype, optional) the desired data type of returned Tensor. Default: if `NULL`, defaults to the dtype of `input`.
- `layout`: (torch.layout, optional) the desired layout of returned tensor. Default: if `NULL`, defaults to the layout of `input`.
- `device`: (torch.device, optional) the desired device of returned tensor. Default: if `NULL`, defaults to the device of `input`.
- `requires_grad`: (bool, optional) If autograd should record operations on the returned tensor. Default: `FALSE`.
- `memory_format`: (torch.memory_format, optional) the desired memory format of returned Tensor. Default: `torch_preserve_format`.

```
full_like(input, fill_value, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False,
memory_format=torch.preserve_format) -> Tensor
```

Returns a tensor with the same size as `input` filled with `fill_value`. `torch_full_like(input, fill_value)` is equivalent to `torch_full(input.size(), fill_value, dtype=input.dtype, layout=input.layout, device=input.device)`.

---

**torch_gather**


**Description**

Gather

**Usage**

```
torch_gather(self, dim, index, sparse_grad = FALSE)
```
torch_gcd

**Arguments**

- `self` *(Tensor)* the source tensor
- `dim` *(int)* the axis along which to index
- `index` *(LongTensor)* the indices of elements to gather
- `sparse_grad` *(bool, optional)* If TRUE, gradient w.r.t. input will be a sparse tensor.

**torch_gather** *(input, dim, index, sparse_grad=FALSE) -> Tensor*

Gathers values along an axis specified by `dim`.
For a 3-D tensor the output is specified by::

\[
\begin{align*}
\text{out}[\text{id}][\text{jd}][\text{kd}] &= \text{input}[\text{index}[\text{id}][\text{jd}][\text{kd}]] \quad \text{if dim == 0} \\
\text{out}[\text{id}][\text{jd}][\text{kd}] &= \text{input}[\text{id}][\text{index}[\text{id}][\text{jd}][\text{kd}]] \quad \text{if dim == 1} \\
\text{out}[\text{id}][\text{jd}][\text{kd}] &= \text{input}[\text{id}][\text{jd}][\text{index}[\text{id}][\text{jd}][\text{kd}]] \quad \text{if dim == 2}
\end{align*}
\]

If input is an n-dimensional tensor with size \((x_0, x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_{n-1})\) and dim = i, then index must be an n-dimensional tensor with size \((x_0, x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_{n-1})\) where \(y \geq 1\) and out will have the same size as index.

**Examples**

```python
if (torch_is_installed()) {
    t = torch_tensor(matrix(c(1,2,3,4), ncol = 2, byrow = TRUE))
    torch_gather(t, 2, torch_tensor(matrix(c(1,1,2,1), ncol = 2, byrow=TRUE), dtype = torch_int64()))
}
```

**torch_gcd** *(input, other) -> Tensor*

**Gcd**

**Usage**

torch_gcd(self, other)

**Arguments**

- `self` *(Tensor)* the input tensor.
- `other` *(Tensor)* the second input tensor

**gcd** *(input, other, *, out=None) -> Tensor*

Computes the element-wise greatest common divisor (GCD) of input and other.
Both input and other must have integer types.
Note
This defines \( \gcd(0, 0) = 0 \).

Examples
if (torch_is_installed()) {
  if (torch::cuda_is_available()) {
    a <- torch_tensor(c(5, 10, 15), dtype = torch_long(), device = "cuda")
    b <- torch_tensor(c(3, 4, 5), dtype = torch_long(), device = "cuda")
    torch_gcd(a, b)
    c <- torch_tensor(c(3L), device = "cuda")
    torch_gcd(a, c)
  }
}

---

torch_ge  Ge

Description
Ge

Usage
torch_ge(self, other)

Arguments

self    (Tensor) the tensor to compare
other   (Tensor or float) the tensor or value to compare

ge(input, other, out=NULL) -> Tensor

Computes \( \text{input} \geq \text{other} \) element-wise.
The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

Examples
if (torch_is_installed()) {
  torch_ge(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
           torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
torch_generator

Create a Generator object

Description

A torch_generator is an object which manages the state of the algorithm that produces pseudo random numbers. Used as a keyword argument in many In-place random sampling functions.

Usage

torch_generator()

Examples

```r
if (torch_is_installed()) {
  # Via string
  generator <- torch_generator()
  generator$current_seed()
  generator$set_current_seed(1234567L)
  generator$current_seed()
}
```

torch_geqrf

Geqrf

Description

Geqrf

Usage

torch_geqrf(self)

Arguments

- self (Tensor) the input matrix
geqrf(input, out=NULL) -> (Tensor, Tensor)

This is a low-level function for calling LAPACK directly. This function returns a namedtuple (a, tau) as defined in LAPACK documentation for geqrf_.
You'll generally want to use torch_qr instead.

Computes a QR decomposition of input, but without constructing Q and R as explicit separate matrices.
Rather, this directly calls the underlying LAPACK function ?geqrf which produces a sequence of 'elementary reflectors'.
See LAPACK documentation for geqrf_ for further details.

---

torch_ger       Ger

Description
Ger

Usage
torch_ger(self, vec2)

Arguments

self          (Tensor) 1-D input vector
vec2         (Tensor) 1-D input vector

ger(input, vec2, out=NULL) -> Tensor

Outer product of input and vec2. If input is a vector of size n and vec2 is a vector of size m, then out must be a matrix of size (n × m).

Note
This function does not broadcast.

Examples

if (torch_is_installed()) {
  v1 = torch_arange(1., 5.)
  v2 = torch_arange(1., 4.)
  torch_ger(v1, v2)
}
torch_get_rng_state  RNG state management

Description
Low level functionality to set and change the RNG state. It’s recommended to use torch_manual_seed() for most cases.

Usage
torch_get_rng_state()

torch_set_rng_state(state)

cuda_get_rng_state(device = NULL)

cuda_set_rng_state(state, device = NULL)

Arguments
state  A tensor with the current state or a list containing the state for each device - (for CUDA).
device  The cuda device index to get or set the state. If NULL gets the state for all available devices.

Functions
• torch_set_rng_state(): Sets the RNG state for the CPU
• cuda_get_rng_state(): Gets the RNG state for CUDA.
• cuda_set_rng_state(): Sets the RNG state for CUDA.

torch_greater  Greater

Description
Greater

Usage
torch_greater(self, other)

Arguments
self  (Tensor) the tensor to compare
other  (Tensor or float) the tensor or value to compare
greater(input, other, *, out=None) -> Tensor

Alias for torch_gt().

greater_equal(input, other, *, out=None) -> Tensor

Alias for torch_ge().

gt(input, other, out=NULL) -> Tensor

Computes input > other element-wise.
The second argument can be a number or a tensor whose shape is broadcastable with the first argument.
torch_hamming_window

Examples

```r
if (torch_is_installed()) {
  torch_gt(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
           torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

torch_hamming_window  Hamming_window

Description

Hamming_window

Usage

```r
torch_hamming_window(
  window_length,
  periodic = TRUE,
  alpha = 0.54,
  beta = 0.46,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE
)
```

Arguments

- **window_length** (int) the size of returned window
- **periodic** (bool, optional) If TRUE, returns a window to be used as periodic function. If False, return a symmetric window.
- **alpha** (float, optional) The coefficient $\alpha$ in the equation above
- **beta** (float, optional) The coefficient $\beta$ in the equation above
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type). Only floating point types are supported.
- **layout** (torch.layout, optional) the desired layout of returned window tensor. Only torch_strided (dense layout) is supported.
- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
Hamming window function.

\[ w[n] = \alpha - \beta \cos \left( \frac{2\pi n}{N - 1} \right), \]

where \( N \) is the full window size.

The input window_length is a positive integer controlling the returned window size. periodic flag determines whether the returned window trims off the last duplicate value from the symmetric window and is ready to be used as a periodic window with functions like torch.stft. Therefore, if periodic is true, the \( N \) in above formula is in fact window_length + 1. Also, we always have torch.hamming_window(L, periodic=True) equal to torch.hamming_window(L + 1, periodic=False)[:-1].

Note

If \( \text{window_length} = 1 \), the returned window contains a single value 1.

This is a generalized version of 'torch.hann_window'.

```
torch_hann_window(  
    window_length,  
    periodic=TRUE,  
    alpha=0.54, beta=0.46,  
    dtype=NULL, layout=torch.strided,  
    device=NULL, requires_grad=False) -> Tensor
```

**Arguments**

- `window_length` (int): the size of returned window
- `periodic` (bool, optional): If TRUE, returns a window to be used as periodic function. If False, return a symmetric window.
- `dtype` (torch.dtype, optional): the desired data type of returned tensor. Default: if NULL, uses a global default (see torch.set_default_tensor_type). Only floating point types are supported.
torch_heaviside

<table>
<thead>
<tr>
<th>torch_heaviside</th>
<th>Heaviside</th>
</tr>
</thead>
</table>

**Description**

Heaviside

**Usage**

torch_heaviside(self, values)

**Arguments**

<table>
<thead>
<tr>
<th>self</th>
<th>(Tensor) the input tensor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>(Tensor) The values to use where input is zero.</td>
</tr>
</tbody>
</table>
heaviside(input, values, *, out=None) -> Tensor

Computes the Heaviside step function for each element in input. The Heaviside step function is defined as:

\[ \text{heaviside}(\text{input}, \text{values}) = \begin{cases} 0, & \text{if } \text{input} < 0 \\ \text{values}, & \text{if } \text{input} == 0 \\ 1, & \text{if } \text{input} > 0 \end{cases} \]

Examples

```r
if (torch_is_installed()) {
  input <- torch_tensor(c(-1.5, 0, 2.0))
  values <- torch_tensor(c(0.5))
  torch_heaviside(input, values)
  values <- torch_tensor(c(1.2, -2.0, 3.5))
  torch_heaviside(input, values)
}
```

torch_histc

Histc

Description

Histc

Usage

torch_histc(self, bins = 100L, min = 0L, max = 0L)

Arguments

- **self** (Tensor) the input tensor.
- **bins** (int) number of histogram bins
- **min** (int) lower end of the range (inclusive)
- **max** (int) upper end of the range (inclusive)

histc(input, bins=100, min=0, max=0, out=NULL) -> Tensor

Computes the histogram of a tensor.

The elements are sorted into equal width bins between min and max. If min and max are both zero, the minimum and maximum values of the data are used.

Examples

```r
if (torch_is_installed()) {
  torch_histc(torch_tensor(c(1., 2, 1)), bins=4, min=0, max=3)
}
```
torch_hstack

Description
Hstack

Usage
torch_hstack(tensors)

Arguments
tensors (sequence of Tensors) sequence of tensors to concatenate

hstack(tensors, *, out=None) -> Tensor
Stack tensors in sequence horizontally (column wise).
This is equivalent to concatenation along the first axis for 1-D tensors, and along the second axis
for all other tensors.

Examples
if (torch_is_installed()) {
    a <- torch_tensor(c(1, 2, 3))
    b <- torch_tensor(c(4, 5, 6))
    torch_hstack(list(a,b))
    a <- torch_tensor(rbind(1,2,3))
    b <- torch_tensor(rbind(4,5,6))
    torch_hstack(list(a,b))
}

torch_hypot

Description
Hypot

Usage
torch_hypot(self, other)
Arguments

```
self (Tensor) the first input tensor
other (Tensor) the second input tensor
```

**hypot(input, other, *, out=None) -> Tensor**

Given the legs of a right triangle, return its hypotenuse.

\[
\text{out}_i = \sqrt{\text{input}_i^2 + \text{other}_i^2}
\]

The shapes of input and other must be broadcastable.

**Examples**

```
if (torch_is_installed()) {
    torch_hypot(torch_tensor(c(4.0)), torch_tensor(c(3.0, 4.0, 5.0)))
}
```

---

**Description**

I0

**Usage**

```
torch_i0(self)
```

**Arguments**

```
self (Tensor) the input tensor
```

**i0(input, *, out=None) -> Tensor**

Computes the zeroth order modified Bessel function of the first kind for each element of input.

\[
\text{out}_i = I_0(\text{input}_i) = \sum_{k=0}^{\infty} \frac{(\text{input}_i^2/4)^k}{(k!)^2}
\]

**Examples**

```
if (torch_is_installed()) {
    torch_i0(torch_arange(start = 0, end = 5, dtype=torch_float32()))
}
```
torch_iinfo  Integer type info

Description
A list that represents the numerical properties of a integer type.

Usage
torch_iinfo(dtype)

Arguments
dtype  dtype to get information from.

torch_imag  Imag

Description
Imag

Usage
torch_imag(self)

Arguments
self  (Tensor) the input tensor.

imag(input) -> Tensor
Returns the imaginary part of the input tensor.

Warning
Not yet implemented.

    out_i = imag(input_i)

Examples
if (torch_is_installed()) {
    ## Not run:
    torch_imag(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))
    ## End(Not run)
}
torch_index  
Index torch tensors

Description
Helper functions to index tensors.

Usage
torch_index(self, indices)

Arguments
self (Tensor) Tensor that will be indexed.
indices (List[Tensor]) List of indices. Indices are torch tensors with torch_long() dtype.

torch_index_put  
Modify values selected by indices.

Description
Modify values selected by indices.

Usage
torch_index_put(self, indices, values, accumulate = FALSE)

Arguments
self (Tensor) Tensor that will be indexed.
indices (List[Tensor]) List of indices. Indices are torch tensors with torch_long() dtype.
values (Tensor) values that will be replaced the indexed location. Used for torch_index_put and torch_index_put_.
accumulate (bool) Whether instead of replacing the current values with values, you want to add them.
torch_index_put_

In-place version of torch_index_put.

Description
In-place version of torch_index_put.

Usage
torch_index_put_(self, indices, values, accumulate = FALSE)

Arguments

self (Tensor) Tensor that will be indexed.
indices (List[Tensor]) List of indices. Indices are torch tensors with torch_long() dtype.
values (Tensor) values that will be replaced the indexed location. Used for torch_index_put and torch_index_put_.
accumulate (bool) Whether instead of replacing the current values with values, you want to add them.

torch_index_select

Index_select

Description
Index_select

Usage
torch_index_select(self, dim, index)

Arguments

self (Tensor) the input tensor.
dim (int) the dimension in which we index
index (LongTensor) the 1-D tensor containing the indices to index

index_select(input, dim, index, out=NULL) -> Tensor

Returns a new tensor which indexes the input tensor along dimension dim using the entries in index which is a LongTensor.

The returned tensor has the same number of dimensions as the original tensor (input). The dim\th dimension has the same size as the length of index; other dimensions have the same size as in the original tensor.
Note

The returned tensor does **not** use the same storage as the original tensor. If out has a different shape than expected, we silently change it to the correct shape, reallocating the underlying storage if necessary.

Examples

```r
if (torch_is_installed()) {
  x = torch_randn(c(3, 4))
  x
  indices = torch_tensor(c(1, 3), dtype = torch_int64())
  torch_index_select(x, 1, indices)
  torch_index_select(x, 2, indices)
}
```

---

**torch_install_path**

A simple exported version of install_path Returns the torch installation path.

---

**torch_inverse**

Inverse

---

Description

Inverse

Usage

```r
torch_inverse()
```

---

Arguments

- **self** (Tensor) the input tensor of size \((*, n, n)\) where \(*\) is zero or more batch dimensions
inverse(input, out=NULL) -> Tensor

Takes the inverse of the square matrix input. input can be batches of 2D square tensors, in which case this function would return a tensor composed of individual inverses.

Note

Irrespective of the original strides, the returned tensors will be transposed, i.e. with strides like `input.contiguous().transpose(-2, -1).stride()`

Examples

```r
if (torch_is_installed()) {
  ## Not run:
  x = torch_randn(c(4, 4))
  y = torch_inverse(x)
  z = torch_mm(x, y)
  z
  torch_max(torch_abs(z - torch_eye(4))) # Max non-zero
  # Batched inverse example
  x = torch_randn(c(2, 3, 4, 4))
  y = torch_inverse(x)
  z = torch_matmul(x, y)
  torch_max(torch_abs(z - torch_eye(4)$expand_as(x))) # Max non-zero
  ## End(Not run)
}
```

---

torch_isclose | Isclose

Description

Isclose

Usage

torch_isclose(self, other, rtol = 1e-05, atol = 1e-08, equal_nan = FALSE)

Arguments

- `self` (Tensor) first tensor to compare
- `other` (Tensor) second tensor to compare
- `rtol` (float, optional) relative tolerance. Default: 1e-05
- `atol` (float, optional) absolute tolerance. Default: 1e-08
- `equal_nan` (bool, optional) if TRUE, then two NaNs will be considered equal. Default: FALSE
isclose(input, other, rtol=1e-05, atol=1e-08, equal_nan=FALSE) -> Tensor

Returns a new tensor with boolean elements representing if each element of input is "close" to the corresponding element of other. Closeness is defined as:

\[ |input - other| \leq atol + rtol \times |other| \]

where input and other are finite. Where input and/or other are nonfinite they are close if and only if they are equal, with NaNs being considered equal to each other when equal_nan is TRUE.

Examples

```python
if (torch_is_installed()) {
    torch_isclose(torch_tensor(c(1., 2, 3)), torch_tensor(c(1 + 1e-10, 3, 4)))
    torch_isclose(torch_tensor(c(Inf, 4)), torch_tensor(c(Inf, 6)), rtol=.5)
}
```

---

torch_isfinite

Isfinite

Description

Isfinite

Usage

torch_isfinite(self)

Arguments

self (Tensor) A tensor to check

TEST

Returns a new tensor with boolean elements representing if each element is Finite or not.

Examples

```python
if (torch_is_installed()) {
    torch_isfinite(torch_tensor(c(1, Inf, 2, -Inf, NaN)))
}
```
torch_isinf

Description
Isinf

Usage
torch_isinf(self)

Arguments
self (Tensor) A tensor to check

TEST
Returns a new tensor with boolean elements representing if each element is +/-INF or not.

Examples
if (torch_is_installed()) {
    torch_isinf(torch_tensor(c(1, Inf, 2, -Inf, NaN)))
}

torch_isnan

Description
Isnан

Usage
torch_isnan(self)

Arguments
self (Tensor) A tensor to check

TEST
Returns a new tensor with boolean elements representing if each element is NaN or not.
Examples

```python
if (torch_is_installed()) {
    torch_isnan(torch_tensor(c(1, NaN, 2)))
}
```

---

**torch_isneginf**

Description

Isneginf

Usage

```
torch_isneginf(self)
```

Arguments

- **self** *(Tensor)* the input tensor.

isneginf(input, *, out=None) -> Tensor

Tests if each element of `input` is negative infinity or not.

Examples

```python
if (torch_is_installed()) {
    a <- torch_tensor(c(-Inf, Inf, 1.2))
    torch_isneginf(a)
}
```

---

**torch_isposinf**

Description

Isposinf

Usage

```
torch_isposinf(self)
```

Arguments

- **self** *(Tensor)* the input tensor.
isposinf(input, *, out=None) -> Tensor

Tests if each element of input is positive infinity or not.

Examples

```r
if (torch_is_installed()) {
  a <- torch_tensor(c(-Inf, Inf, 1.2))
  torch_isposinf(a)
}
```

isreal(input) -> Tensor

Returns a new tensor with boolean elements representing if each element of input is real-valued or not. All real-valued types are considered real. Complex values are considered real when their imaginary part is 0.

Examples

```r
if (torch_is_installed()) {
  if (FALSE) {
    torch_isreal(torch_tensor(c(1, 1+1i, 2+0i)))
  }
}
```
torch_istft

Description
Inverse short time Fourier Transform. This is expected to be the inverse of `torch_stft()`.

Usage
```
torch_istft(
    self,
    n_fft,
    hop_length = NULL,
    win_length = NULL,
    window = list(),
    center = TRUE,
    normalized = FALSE,
    onesided = NULL,
    length = NULL,
    return_complex = FALSE
)
```

Arguments
- **self** (Tensor) The input tensor. Expected to be output of `torch_stft()`, can either be complex (channel, fft_size, n_frame), or real (channel, fft_size, n_frame, 2) where the channel dimension is optional.
- **n_fft** (int) Size of Fourier transform
- **hop_length** (Optional[int]) The distance between neighboring sliding window frames. (Default: n_fft % 4)
- **win_length** (Optional[int]) The size of window frame and STFT filter. (Default: n_fft)
- **window** (Optional(torch.Tensor)) The optional window function. (Default: `torch_ones(win_length)`)
- **center** (bool) Whether input was padded on both sides so that the t-th frame is centered at time $t \times$ hop_length. (Default: TRUE)
- **normalized** (bool) Whether the STFT was normalized. (Default: FALSE)
- **onesided** (Optional(bool)) Whether the STFT was onesided. (Default: TRUE if n_fft != fft_size in the input size)
- **length** (Optional(int)) The amount to trim the signal by (i.e. the original signal length). (Default: whole signal)
- **return_complex** (Optional(bool)) Whether the output should be complex, or if the input should be assumed to derive from a real signal and window. Note that this is incompatible with onesided=TRUE. (Default: FALSE)
Details

It has the same parameters (+ additional optional parameter of length) and it should return the least squares estimation of the original signal. The algorithm will check using the NOLA condition (nonzero overlap).

Important consideration in the parameters window and center so that the envelop created by the summation of all the windows is never zero at certain point in time. Specifically, $\sum_{t=-\infty}^{\infty} |w|^2 (n - t \times \text{hop_length}) \neq 0$.

Since `torch_stft()` discards elements at the end of the signal if they do not fit in a frame, istft may return a shorter signal than the original signal (can occur if center is FALSE since the signal isn’t padded).

If center is TRUE, then there will be padding e.g. 'constant', 'reflect', etc. Left padding can be trimmed off exactly because they can be calculated but right padding cannot be calculated without additional information.

Example: Suppose the last window is: [c(17, 18, 0, 0, 0) vs c(18, 0, 0, 0, 0)

The n_fft, hop_length, win_length are all the same which prevents the calculation of right padding. These additional values could be zeros or a reflection of the signal so providing length could be useful. If length is None then padding will be aggressively removed (some loss of signal).


---

torch_is_complex  

**Is_complex**

**Description**

Is_complex

**Usage**

torch_is_complex(self)

**Arguments**

self  

(Tensor) the PyTorch tensor to test

**is_complex(input) -> (bool)**

Returns TRUE if the data type of input is a complex data type i.e., one of torch_complex64, and torch.complex128.
torch_is_floating_point

Description

Is_floating_point

Usage

torch_is_floating_point(self)

Arguments

self

(Tensor) the PyTorch tensor to test

is_floating_point(input) -> (bool)

Returns TRUE if the data type of input is a floating point data type i.e., one of torch_float64, torch.float32 and torch.float16.

torch_is_installed

Description

Verifies if torch is installed

Usage

torch_is_installed()
torch_is_nonzero

Description

Is_nonzero

Usage

torch_is_nonzero(self)

Arguments

self (Tensor) the input tensor.

is_nonzero(input) -> (bool)

Returns TRUE if the input is a single element tensor which is not equal to zero after type conversions, i.e. not equal to torch_tensor(c(0)) or torch_tensor(c(0)) or torch_tensor(c(FALSE)). Throws a RuntimeError if torch_numel() != 1 (even in case of sparse tensors).

Examples

if (torch_is_installed()) {

torch_is_nonzero(torch_tensor(c(0.)))
torch_is_nonzero(torch_tensor(c(1.5)))
torch_is_nonzero(torch_tensor(c(FALSE)))
torch_is_nonzero(torch_tensor(c(3)))
if (FALSE) {
	torch_is_nonzero(torch_tensor(c(1, 3, 5)))
torch_is_nonzero(torch_tensor(c())))
}
}

torch_kaiser_window

Kaiser_window

Description

Kaiser_window
torch_kaiser_window

Usage

torch_kaiser_window(
    window_length,
    periodic,
    beta,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = NULL
)

Arguments

window_length (int) length of the window.
periodic (bool, optional) If TRUE, returns a periodic window suitable for use in spectral analysis. If FALSE, returns a symmetric window suitable for use in filter design.
beta (float, optional) shape parameter for the window.
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type). If dtype is not given, infer the data type from the other input arguments. If any of start, end, or stop are floating-point, the dtype is inferred to be the default dtype, see ~torch.get_default_dtype. Otherwise, the dtype is inferred to be torch.int64.
layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

kaiser_window(window_length, periodic=TRUE, beta=12.0, *, dtype=None, layout=torch.strided, device=None, requires_grad=FALSE) -> Tensor

Computes the Kaiser window with window length window_length and shape parameter beta.

Let I_0 be the zeroth order modified Bessel function of the first kind (see torch_i0()) and N = L - 1 if periodic is FALSE and L if periodic is TRUE, where L is the window_length. This function computes:

\[
out_i = I_0 \left( \beta \sqrt{1 - \left( \frac{i - N/2}{N/2} \right)^2} \right) / I_0(\beta)
\]

Calling torch_kaiser_window(L, B, periodic=TRUE) is equivalent to calling torch_kaiser_window(L + 1, B, periodic=FALSE). The periodic argument is intended as a helpful shorthand to produce a periodic window as input to functions like torch_stft().
Note

If `window_length` is one, then the returned window is a single element tensor containing a one.

torch_kron

Kronecker product

Description

Computes the Kronecker product of `self` and `other`.

Usage

```python
torch_kron(self, other)
```

Arguments

- `self` (Tensor) input Tensor
- `other` (Tensor) other tensor.

torch_kthvalue

Kthvalue

Description

Kthvalue

Usage

```python
torch_kthvalue(self, k, dim = -1L, keepdim = FALSE)
```

Arguments

- `self` (Tensor) the input tensor.
- `k` (int) k for the k-th smallest element
- `dim` (int, optional) the dimension to find the kth value along
- `keepdim` (bool) whether the output tensor has dim retained or not.

kthvalue(input, k, dim=NULL, keepdim=False, out=NULL) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the k th smallest element of each row of the input tensor in the given dimension dim. And indices is the index location of each element found.

If `dim` is not given, the last dimension of the input is chosen.

If `keepdim` is TRUE, both the values and indices tensors are the same size as input, except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see `torch_squeeze`), resulting in both the values and indices tensors having 1 fewer dimension than the input tensor.
torch_lcm

### Examples
```r
if (torch_is_installed()) {
  x <- torch_arange(1, 6)
  x
  torch_kthvalue(x, 4)
  x <- torch_arange(1, 6)$resize_(c(2,3))
  x
  torch_kthvalue(x, 2, 1, TRUE)
}
```

---

| torch_layout | *Creates the corresponding layout*
|--------------|---------------------------------|

**Description**

Creates the corresponding layout

**Usage**

- `torch_strided()`
- `torch_sparse_coo()`

---

| torch_lcm | *Lcm*
|-----------|---------------------------------|

**Description**

Lcm

**Usage**

- `torch_lcm(self, other)`

**Arguments**

- `self` (Tensor) the input tensor.
- `other` (Tensor) the second input tensor

**lcm(input, other, *, out=None) -> Tensor**

Computes the element-wise least common multiple (LCM) of `input` and `other`. Both `input` and `other` must have integer types.
Note

This defines \( \text{lcm}(0, 0) = 0 \) and \( \text{lcm}(0, a) = 0 \).

Examples

```r
if (torch_is_installed()) {
  if (torch::cuda_is_available()) {
    a <- torch_tensor(c(5, 10, 15), dtype = torch_long(), device = "cuda")
    b <- torch_tensor(c(3, 4, 5), dtype = torch_long(), device = "cuda")
    torch_lcm(a, b)
    c <- torch_tensor(c(3L), device = "cuda")
    torch_lcm(a, c)
  }
}
```

---

**torch_le**

**Le**

### Description

Le

### Usage

`torch_le(self, other)`

### Arguments

- `self` (Tensor) the tensor to compare
- `other` (Tensor or float) the tensor or value to compare

### lt(input, other, out=NULL) -> Tensor

Computes input \( \leq \) other element-wise.

The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

### Examples

```r
if (torch_is_installed()) {
  torch_le(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
            torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```
torch_lerp  Lerp

Description
Lerp

Usage
torch_lerp(self, end, weight)

Arguments
self (Tensor) the tensor with the starting points
end (Tensor) the tensor with the ending points
weight (float or tensor) the weight for the interpolation formula

lerp(input, end, weight, out=NULL)

Does a linear interpolation of two tensors start (given by input) and end based on a scalar or tensor weight and returns the resulting out tensor.

\[ \text{out}_i = \text{start}_i + \text{weight}_i \times (\text{end}_i - \text{start}_i) \]

The shapes of start and end must be broadcastable. If weight is a tensor, then the shapes of weight, start, and end must be broadcastable.

Examples
if (torch_is_installed()) {
    start = torch.arange(1, 4)
    end = torch.empty(4)$fill_(10)
    start
    end
    torch_lerp(start, end, 0.5)
    torch_lerp(start, end, torch_full_like(start, 0.5))
}
torch_less  

Description
Less

Usage
torch_less(self, other)

Arguments
self  (Tensor) the tensor to compare
other  (Tensor or float) the tensor or value to compare

less(input, other, *, out=None) -> Tensor
Alias for torch_lt().

torch_less_equal  

Description
Less_equal

Usage
torch_less_equal(self, other)

Arguments
self  (Tensor) the tensor to compare
other  (Tensor or float) the tensor or value to compare

less_equal(input, other, *, out=None) -> Tensor
Alias for torch_le().
torch_lgamma

Description
Lgamma

Usage
torch_lgamma(self)

Arguments
self (Tensor) the input tensor.

lgamma(input, out=NULL) -> Tensor
Computes the logarithm of the gamma function on input.

\[ \text{out}_i = \log \Gamma(\text{input}_i) \]

Examples
if (torch_is_installed()) {
  a = torch_arange(0.5, 2, 0.5)
  torch_lgamma(a)
}

torch_linspace

Description
Linspace

Usage
torch_linspace(
  start,
  end,
  steps = 100,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE
)
torch_load

Loads a saved object

Description

Loads a saved object

Usage

torch_load(path, device = "cpu")

Arguments

<table>
<thead>
<tr>
<th>path</th>
<th>a path to the saved object</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>a device to load tensors to. By default we load to the cpu but you can also load them to any cuda device. If NULL then the device where the tensor has been saved will be reused.</td>
</tr>
</tbody>
</table>
### torch_log

**Description**

Log

**Usage**

```r
torch_log(self)
```

**Arguments**

- `self` (Tensor) the input tensor.

**log(input, out=NULL) -> Tensor**

Returns a new tensor with the natural logarithm of the elements of `input`.

\[ y_i = \log_e(x_i) \]

**Examples**

```r
if (torch_is_installed()) {
  a = torch_randn(c(5))
  a
  torch_log(a)
}
```

### torch_log10

**Description**

Log10

**Usage**

```r
torch_log10(self)
```
torch_log1p

Arguments

self (Tensor) the input tensor.

log10(input, out=NULL) -> Tensor

Returns a new tensor with the logarithm to the base 10 of the elements of input.

\[ y_i = \log_{10}(x_i) \]

Examples

```r
if (torch_is_installed()) {
  a = torch_rand(5)
  a
  torch_log10(a)
}
```

torch_log1p

Log1p

Description

Log1p

Usage

torch_log1p(self)

Arguments

self (Tensor) the input tensor.

log1p(input, out=NULL) -> Tensor

Returns a new tensor with the natural logarithm of \((1 + \text{input})\).

\[ y_i = \log_e(x_i + 1) \]

Note

This function is more accurate than torch_log for small values of input.
torch_log2

Examples
if (torch_is_installed()) {
    a = torch_randn(c(5))
a    torch_log1p(a)
}

torch_log2      Log2

Description
Log2

Usage
torch_log2(self)

Arguments
self (Tensor) the input tensor.

log2(input, out=NULL) -> Tensor

Returns a new tensor with the logarithm to the base 2 of the elements of input.

\[ y_i = \log_2(x_i) \]

Examples
if (torch_is_installed()) {
    a = torch_rand(5)
a    torch_log2(a)
}
torch_logaddexp

Description

Logaddexp

Usage

torch_logaddexp(self, other)

Arguments

- **self**: (Tensor) the input tensor.
- **other**: (Tensor) the second input tensor

Logaddexp(input, other, *, out=None) -> Tensor

Logarithm of the sum of exponentiations of the inputs.
Calculates pointwise \( \log(e^x + e^y) \). This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

This op should be disambiguated with torch_logsumexp() which performs a reduction on a single tensor.

Examples

```python
if (torch_is_installed()) {
    torch_logaddexp(torch_tensor(c(-1.0)), torch_tensor(c(-1.0, -2, -3)))
    torch_logaddexp(torch_tensor(c(-100.0, -200, -300)), torch_tensor(c(-1.0, -2, -3)))
    torch_logaddexp(torch_tensor(c(1.0, 2000, 30000)), torch_tensor(c(-1.0, -2, -3)))
}
```

---

torch_logaddexp2

Description

Logaddexp2

Usage

torch_logaddexp2(self, other)
torch_logcumsumexp

Arguments

self  (Tensor) the input tensor.
other  (Tensor) the second input tensor

logaddexp2(input, other, *, out=None) -> Tensor

Logarithm of the sum of exponentiations of the inputs in base-2.
Calculates pointwise $\log_2 (2^x + 2^y)$. See torch_logaddexp() for more details.

torch_logcumsumexp

Description

Logcumsumexp

Usage

torch_logcumsumexp(self, dim)

Arguments

self  (Tensor) the input tensor.
dim  (int) the dimension to do the operation over

logcumsumexp(input, dim, *, out=None) -> Tensor

Returns the logarithm of the cumulative summation of the exponentiation of elements of input in the dimension dim.
For summation index $j$ given by dim and other indices $i$, the result is

$$\logcumsumexp(x)_{ij} = \log \sum_{j=0}^{i} \exp(x_{ij})$$

Examples

if (torch_is_installed()) {

a <- torch_randn(c(10))
torch_logcumsumexp(a, dim=1)
}
**torch_logdet**  

<table>
<thead>
<tr>
<th>torch_logdet</th>
<th>Logdet</th>
</tr>
</thead>
</table>

**Description**  
Logdet

**Usage**  
`torch_logdet(self)`

**Arguments**  

```markdown
self  
(Tensor) the input tensor of size (*, n, n) where * is zero or more batch dimensions.
```

**logdet(input) -> Tensor**

Calculates log determinant of a square matrix or batches of square matrices.

**Note**

Result is `-inf` if `input` has zero log determinant, and is `NaN` if `input` has negative determinant.

Backward through `logdet` internally uses SVD results when `input` is not invertible. In this case, double backward through `logdet` will be unstable in when `input` doesn't have distinct singular values. See `~torch.svd` for details.

**Examples**

```r
if (torch_is_installed()) {
  A = torch_randn(c(3, 3))
  torch_det(A)
  torch_logdet(A)
  A
  A$det()
  A$det()$log()
}
```
### torch_logical_and  
**Logical_and**

**Description**

Logical_and

**Usage**

torch_logical_and(self, other)

**Arguments**

- **self**: (Tensor) the input tensor.
- **other**: (Tensor) the tensor to compute AND with

**logical_and(input, other, out=NULL) -> Tensor**

Computes the element-wise logical AND of the given input tensors. Zeros are treated as FALSE and nonzeros are treated as TRUE.

**Examples**

if (torch_is_installed()) {

    torch_logical_and(torch_tensor(c(TRUE, FALSE, TRUE)), torch_tensor(c(TRUE, FALSE, FALSE)))
    a = torch_tensor(c(0, 1, 10, 0), dtype=torch_int8())
    b = torch_tensor(c(4, 0, 1, 0), dtype=torch_int8())
    torch.logical_and(a, b)
    ## Not run:
    torch.logical_and(a, b, out=torch_empty(4, dtype=torch_bool()))
    ## End(Not run)
}

---

### torch_logical_not  
**Logical_not**

**Description**

Logical_not

**Arguments**

- **self**: (Tensor) the input tensor.
**logical_not(input, out=NULL) -> Tensor**

Computes the element-wise logical NOT of the given input tensor. If not specified, the output tensor will have the bool dtype. If the input tensor is not a bool tensor, zeros are treated as FALSE and non-zeros are treated as TRUE.

**Examples**

```r
if (torch_is_installed()) {
    torch_logical_not(torch_tensor(c(TRUE, FALSE)))
    torch_logical_not(torch_tensor(c(0, 1, -10), dtype=torch_int8()))
    torch_logical_not(torch_tensor(c(0., 1.5, -10.), dtype=torch_double()))
}
```

**torch_logical_or**

**Logical_or**

**Description**

Logical_or

**Usage**

`torch_logical_or(self, other)`

**Arguments**

- `self` (Tensor) the input tensor.
- `other` (Tensor) the tensor to compute OR with

**logical_or(input, other, out=NULL) -> Tensor**

Computes the element-wise logical OR of the given input tensors. Zeros are treated as FALSE and nonzeros are treated as TRUE.

**Examples**

```r
if (torch_is_installed()) {
    torch_logical_or(torch_tensor(c(TRUE, FALSE, TRUE)), torch_tensor(c(TRUE, FALSE, FALSE)))
    a = torch_tensor(c(0, 1, 10, 0), dtype=torch_int8())
    b = torch_tensor(c(4, 0, 1, 0), dtype=torch_int8())
    torch_logical_or(a, b)
    ## Not run:
    torch_logical_or(a$double(), b$double())
    torch_logical_or(a$double(), b)
    torch_logical_or(a, b, out=torch_empty(4, dtype=torch_bool()))
    # End(Not run)
}
```
torch_logical_xor  Logical_xor

Description

Logical_xor

Usage

torch_logical_xor(self, other)

Arguments

self  (Tensor) the input tensor.
other  (Tensor) the tensor to compute XOR with

logical_xor(input, other, out=NULL) -> Tensor

Computes the element-wise logical XOR of the given input tensors. Zeros are treated as FALSE and nonzeros are treated as TRUE.

Examples

if (torch_is_installed()) {
    torch_logical_xor(torch_tensor(c(TRUE, FALSE, TRUE)), torch_tensor(c(TRUE, FALSE, FALSE)))
    a = torch_tensor(c(0, 1, 10, 0), dtype=torch_int8())
    b = torch_tensor(c(4, 0, 1, 0), dtype=torch_int8())
    torch_logical_xor(a, b)
    torch_logical_xor(a$to(dtype=torch_double()), b$to(dtype=torch_double()))
    torch_logical_xor(a$to(dtype=torch_double()), b)
}

torch_logit  Logit

Description

Logit

Usage

torch_logit(self, eps = NULL)
Arguments

self  (Tensor) the input tensor.
eps  (float, optional) the epsilon for input clamp bound. Default: None

logit(input, eps=None, *, out=None) -> Tensor

Returns a new tensor with the logit of the elements of input. input is clamped to [eps, 1 - eps] when eps is not None. When eps is None and input < 0 or input > 1, the function will yields NaN.

\[ y_i = \ln\left( \frac{z_i}{1 - z_i} \right) \]

\[ z_i = \begin{cases} x_i & \text{if } \text{eps is None} \\ \text{eps} & \text{if } x_i < \text{eps} \\ x_i & \text{if } x_i \leq 1 - \text{eps} \\ 1 - \text{eps} & \text{if } x_i > 1 - \text{eps} \end{cases} \]

Examples

```python
if (torch_is_installed()) {
    a <- torch_rand(5)
    a
    torch_logit(a, eps=1e-6)
}
```

torch_logspace

Logspace

Description

Logspace

Usage

```python
torch_logspace(
    start,
    end,
    steps = 100,
    base = 10,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)```
torch_logsumexp

Arguments

- **start** *(float)* the starting value for the set of points
- **end** *(float)* the ending value for the set of points
- **steps** *(int)* number of points to sample between *start* and *end*. Default: 100.
- **base** *(float)* base of the logarithm function. Default: 10.0.
- **dtype** *(torch.dtype, optional)* the desired data type of returned tensor. Default: if NULL, uses a global default (see *torch_set_default_tensor_type*).
- **layout** *(torch.layout, optional)* the desired layout of returned Tensor. Default: *torch_strided*.
- **device** *(torch.device, optional)* the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see *torch_set_default_tensor_type*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** *(bool, optional)* If autograd should record operations on the returned tensor. Default: FALSE.

logspace(*start*, *end*, *steps*=100, *base*=10.0, *out=NULL*, *dtype=NULL*, *layout=torch.strided*, *device=NULL*, *requires_grad=False*) -> Tensor

Returns a one-dimensional tensor of *steps* points logarithmically spaced with base *base* between base*start* and base*end*.

The output tensor is 1-D of size *steps*.

Examples

```python
if (torch_is_installed()) {
    torch_logspace(start=-10, end=10, steps=5)
    torch_logspace(start=0.1, end=1.0, steps=5)
    torch_logspace(start=0.1, end=1.0, steps=1)
    torch_logspace(start=2, end=2, steps=1, base=2)
}
```

torch_logsumexp

Description

Logsumexp

Usage

torch_logsumexp(self, dim, keepdim = FALSE)
torch_lstsq

Arguments

self (Tensor) the input tensor.
dim (int or tuple of ints) the dimension or dimensions to reduce.
keepdim (bool) whether the output tensor has dim retained or not.

logsumexp(input, dim, keepdim=False, out=NULL)

Returns the log of summed exponentials of each row of the input tensor in the given dimension dim. The computation is numerically stabilized.
For summation index \( j \) given by \( \text{dim} \) and other indices \( i \), the result is

\[
\text{logsumexp}(x)_i = \log \sum_j \exp(x_{ij})
\]

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) \( \text{dim} \) where it is of size 1. Otherwise, \( \text{dim} \) is squeezed (see torch_squeeze), resulting in the output tensor having 1 (or \( \text{len}(\text{dim}) \)) fewer dimension(s).

Examples

```r
if (torch_is_installed()) {
    a = torch_randn(c(3, 3))
    torch_logsumexp(a, 1)
}
```

torch_lstsq  Lstsq

Description

Lstsq

Arguments

self (Tensor) the matrix \( B \)
A (Tensor) the \( m \) by \( n \) matrix \( A \)

lstsq(input, A, out=NULL) -> Tensor

Computes the solution to the least squares and least norm problems for a full rank matrix \( A \) of size \((m \times n)\) and a matrix \( B \) of size \((m \times k)\).
If \( m \geq n \), torch_lstsq() solves the least-squares problem:

\[
\min_X \|AX - B\|_2.
\]
If \( m < n \), \texttt{torch_lstsq()} solves the least-norm problem:

\[
\min_X \|X\|_2 \quad \text{subject to} \quad AX = B.
\]

Returned tensor \( X \) has shape \((\max(m,n) \times k)\). The first \( n \) rows of \( X \) contains the solution. If \( m \geq n \), the residual sum of squares for the solution in each column is given by the sum of squares of elements in the remaining \( m - n \) rows of that column.

**Note**

The case when \( m < n \) is not supported on the GPU.

---

**torch_lt**

**Lt**

**Description**

Lt

**Usage**

\texttt{torch\_lt(self, other)}

**Arguments**

- \texttt{self} (Tensor) the tensor to compare
- \texttt{other} (Tensor or float) the tensor or value to compare

**Lt(input, other, out=NULL) -> Tensor**

Computes \( \texttt{input < other} \) element-wise.

The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

**Examples**

```r
if (torch_is_installed()) {
  torch_lt(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
           torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```
torch_lu

Description
Computes the LU factorization of a matrix or batches of matrices A. Returns a tuple containing the LU factorization and pivots of A. Pivoting is done if pivot is set to True.

Usage
torch_lu(A, pivot = TRUE, get_infos = FALSE, out = NULL)

Arguments
A (Tensor) the tensor to factor of size (m, n)
pivot (bool, optional) – controls whether pivoting is done. Default: TRUE
get_infos (bool, optional) – if set to True, returns an info IntTensor. Default: FALSE
out (tuple, optional) – optional output tuple. If get_infos is True, then the elements in the tuple are Tensor, IntTensor, and IntTensor. If get_infos is False, then the elements in the tuple are Tensor, IntTensor. Default: NULL

Examples
if (torch_is_installed()) {
A <- torch_randn(c(2, 3, 3))
torch_lu(A)
}

torch_lu_solve

Description
Lu_solve

Usage
torch_lu_solve(self, LU_data, LU_pivots)
torch_lu_unpack

Arguments

self (Tensor) the RHS tensor of size (*, m, k), where * is zero or more batch dimensions.
LU_data (Tensor) the pivoted LU factorization of A from torch_lu of size (*, m, m), where * is zero or more batch dimensions.
LU_pivots (IntTensor) the pivots of the LU factorization from torch_lu of size (*, m), where * is zero or more batch dimensions. The batch dimensions of LU_pivots must be equal to the batch dimensions of LU_data.

lu_solve(input, LU_data, LU_pivots, out=NULL) -> Tensor

Returns the LU solve of the linear system \(Ax = b\) using the partially pivoted LU factorization of A from torch_lu.

Examples

```r
if (torch_is_installed()) {
  A = torch_randn(c(2, 3, 3))
  b = torch_randn(c(2, 3, 1))
  out = torch_lu(A)
  x = torch_lu_solve(b, out[[1]], out[[2]])
  torch_norm(torch_bmm(A, x) - b)
}
```

torch_lu_unpack

Lu_unpack

Description

Lu_unpack

Usage

torch_lu_unpack(LU_data, LU_pivots, unpack_data = TRUE, unpack_pivots = TRUE)

Arguments

LU_data (Tensor) – the packed LU factorization data
LU_pivots (Tensor) – the packed LU factorization pivots
unpack_data (logical) – flag indicating if the data should be unpacked. If FALSE, then the returned L and U are NULL. Default: TRUE
unpack_pivots (logical) – flag indicating if the pivots should be unpacked into a permutation matrix P. If FALSE, then the returned P is None. Default: TRUE
torch_manual_seed

Sets the seed for generating random numbers.

Description

Sets the seed for generating random numbers.

Usage

torch_manual_seed(seed)

local_torch_manual_seed(seed, .env = parent.frame())

with_torch_manual_seed(code, ..., seed)

Arguments

seed integer seed.

.env environment that will take the modifications from manual_seed.

code expression to run in the context of the seed

... unused currently.

Functions

• local_torch_manual_seed(): Modifies the torch seed in the environment scope.

• with_torch_manual_seed(): A with context to change the seed during the function execution.

Note

Currently the local_torch_manual_seed and with_torch_manual_seed won't work with Tensors in the MPS device. You can sample the tensors on CPU and move them to MPS if reproducibility is required.
torch_masked_select  Masked_select

Description

Masked_select

Usage

torch_masked_select(self, mask)

Arguments

self  (Tensor) the input tensor.
mask  (BoolTensor) the tensor containing the binary mask to index with

masked_select(input, mask, out=NULL) -> Tensor

Returns a new 1-D tensor which indexes the input tensor according to the boolean mask mask which is a BoolTensor.

The shapes of the mask tensor and the input tensor don’t need to match, but they must be broadcastable.

Note

The returned tensor does not use the same storage as the original tensor

Examples

```bash
if (torch_is_installed()) {
  x = torch_randn(c(3, 4))
  x
  mask = x$ge(0.5)
  mask
  torch_masked_select(x, mask)
}
```
Description

Matmul

Usage

torch_matmul(self, other)

Arguments

self (Tensor) the first tensor to be multiplied
other (Tensor) the second tensor to be multiplied

matmul(input, other, out=NULL) -> Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if input is a \((j \times 1 \times n \times m)\) tensor and other is a \((k \times m \times p)\) tensor, out will be an \((j \times k \times n \times p)\) tensor.

Note

The 1-dimensional dot product version of this function does not support an ‘out’ parameter.
Examples

```r
if (torch_is_installed()) {

  # vector x vector
  tensor1 = torch_randn(c(3))
  tensor2 = torch_randn(c(3))
  torch_matmul(tensor1, tensor2)

  # matrix x vector
  tensor1 = torch_randn(c(3, 4))
  tensor2 = torch_randn(c(4))
  torch_matmul(tensor1, tensor2)

  # batched matrix x broadcasted vector
  tensor1 = torch_randn(c(10, 3, 4))
  tensor2 = torch_randn(c(4))
  torch_matmul(tensor1, tensor2)

  # batched matrix x batched matrix
  tensor1 = torch_randn(c(10, 3, 4))
  tensor2 = torch_randn(c(10, 4, 5))
  torch_matmul(tensor1, tensor2)

  # batched matrix x broadcasted matrix
  tensor1 = torch_randn(c(10, 3, 4))
  tensor2 = torch_randn(c(4, 5))
  torch_matmul(tensor1, tensor2)
}
```

---

**torch_matrix_exp**  
**Matrix_exp**

**Description**
Matrix_exp

**Usage**
```
torch_matrix_exp(self)
```

**Arguments**

- `self` (Tensor) the input tensor.

**matrix_power(input) -> Tensor**

Returns the matrix exponential. Supports batched input. For a matrix \( A \), the matrix exponential is defined as

\[
\exp^A = \sum_{k=0}^{\infty} A^k / k!.
\]

The implementation is based on: Bader, P.; Blanes, S.; Casas, F. Computing the Matrix Exponential with an Optimized Taylor Polynomial Approximation. Mathematics 2019, 7, 1174.
torch_matrix_power

Examples

if (torch_is_installed()) {
    a <- torch_randn(c(2, 2, 2))
    a[1, , ] <- torch_eye(2, 2)
    a[2, , ] <- 2 * torch_eye(2, 2)
    a
    torch_matrix_exp(a)

    x <- torch_tensor(rbind(c(0, pi/3), c(-pi/3, 0)))
    x$matrix_exp() # should be [[cos(pi/3), sin(pi/3)], [-sin(pi/3), cos(pi/3)]]
}

Description

Matrix_power

Usage

torch_matrix_power(self, n)

Arguments

self (Tensor) the input tensor.
n (int) the power to raise the matrix to

matrix_power(input, n) -> Tensor

Returns the matrix raised to the power n for square matrices. For batch of matrices, each individual matrix is raised to the power n.

If n is negative, then the inverse of the matrix (if invertible) is raised to the power n. For a batch of matrices, the batched inverse (if invertible) is raised to the power n. If n is 0, then an identity matrix is returned.

Examples

if (torch_is_installed()) {
    a = torch_randn(c(2, 2, 2))
    a
    torch_matrix_power(a, 3)
}
torch_matrix_rank  Matrix_rank

Description
Matrix_rank

Arguments
- self (Tensor) the input 2-D tensor
- tol (float, optional) the tolerance value. Default: NULL
- symmetric (bool, optional) indicates whether input is symmetric. Default: FALSE

matrix_rank(input, tol=NULL, symmetric=False) -> Tensor

Returns the numerical rank of a 2-D tensor. The method to compute the matrix rank is done using SVD by default. If symmetric is TRUE, then input is assumed to be symmetric, and the computation of the rank is done by obtaining the eigenvalues.

tol is the threshold below which the singular values (or the eigenvalues when symmetric is TRUE) are considered to be 0. If tol is not specified, tol is set to S.max() * max(S.size()) * eps where S is the singular values (or the eigenvalues when symmetric is TRUE), and eps is the epsilon value for the datatype of input.

torch_max  Max

Description
Max

Arguments
- self (Tensor) the input tensor.
- dim (int) the dimension to reduce.
- keepdim (bool) whether the output tensor has dim retained or not. Default: FALSE.
- out (tuple, optional) the result tuple of two output tensors (max, max_indices)
- other (Tensor) the second input tensor

max(input) -> Tensor

Returns the maximum value of all elements in the input tensor.
max(input, dim, keepdim=False, out=NULL) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the maximum value of each row of the input tensor in the given dimension dim. And indices is the index location of each maximum value found (argmax).

Warning

indices does not necessarily contain the first occurrence of each maximal value found, unless it is unique. The exact implementation details are device-specific. Do not expect the same result when run on CPU and GPU in general.

If keepdim is TRUE, the output tensors are of the same size as input except in the dimension dim where they are of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output tensors having 1 fewer dimension than input.

max(input, other, out=NULL) -> Tensor

Each element of the tensor input is compared with the corresponding element of the tensor other and an element-wise maximum is taken.

The shapes of input and other don’t need to match, but they must be broadcastable.

\[ \text{out}_i = \max(\text{tensor}_i, \text{other}_i) \]

Note

When the shapes do not match, the shape of the returned output tensor follows the broadcasting rules.

Examples

```python
if (torch_is_installed()) {

    a = torch_randn(c(1, 3))
    a
    torch_max(a)

    a = torch_randn(c(4, 4))
    a
    torch_max(a, dim = 1)

    a = torch_randn(c(4))
    a
    b = torch_randn(c(4))
    b
    torch_max(a, other = b)
}
```
torch_maximum

**Description**

Maximum

**Usage**

torch_maximum(self, other)

**Arguments**

- self (Tensor) the input tensor.
- other (Tensor) the second input tensor

maximum(input, other, *, out=None) -> Tensor

Computes the element-wise maximum of input and other.

**Note**

If one of the elements being compared is a NaN, then that element is returned. torch_maximum() is not supported for tensors with complex dtypes.

**Examples**

```r
if (torch_is_installed()) {

  a <- torch_tensor(c(1, 2, -1))
  b <- torch_tensor(c(3, 0, 4))
  torch_maximum(a, b)
}
```

torch_mean

**Description**

Mean

**Usage**

torch_mean(self, dim, keepdim = FALSE, dtype = NULL)
torch_median

Arguments

- **self** (Tensor) the input tensor.
- **dim** (int or tuple of ints) the dimension or dimensions to reduce.
- **keepdim** (bool) whether the output tensor has dim retained or not.
- **dtype** the resulting data type.

mean(input) -> Tensor

Returns the mean value of all elements in the input tensor.

mean(input, dim, keepdim=False, out=NULL) -> Tensor

Returns the mean value of each row of the input tensor in the given dimension dim. If dim is a list of dimensions, reduce over all of them.

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

Examples

```python
if (torch_is_installed()) {
  a = torch_randn(c(1, 3))
  a
  torch_mean(a)

  a = torch_randn(c(4, 4))
  a
  torch_mean(a, 1)
  torch_mean(a, 1, TRUE)
}
```

torch_median | Median

Description

Median

Usage

torch_median(self, dim, keepdim = FALSE)
### Arguments

- **self** (Tensor) the input tensor.
- **dim** (int) the dimension to reduce.
- **keepdim** (bool) whether the output tensor has dim retained or not.

### median(input) -> Tensor

Returns the median value of all elements in the input tensor.

### median(input, dim=-1, keepdim=False, out=NULL) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the median value of each row of the input tensor in the given dimension dim. And indices is the index location of each median value found.

By default, dim is the last dimension of the input tensor.

If keepdim is TRUE, the output tensors are of the same size as input except in the dimension dim where they are of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the outputs tensor having 1 fewer dimension than input.

### Examples

```python
if (torch_is_installed()) {
    a = torch_randn(c(1, 3))
    a
    torch_median(a)

    a = torch_randn(c(4, 5))
    a
    torch_median(a, 1)
}
```

### torch_memory_format

<table>
<thead>
<tr>
<th>Memory format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Returns the correspondent memory format.</td>
</tr>
</tbody>
</table>

### Usage

- `torch_contiguous_format()`
- `torch_preserve_format()`
- `torch_channels_last_format()`
torch_meshgrid  Meshgrid

Description

Take $N$ tensors, each of which can be either scalar or 1-dimensional vector, and create $N$ N-dimensional grids, where the $i$ th grid is defined by expanding the $i$ th input over dimensions defined by other inputs.

Usage

```
torch_meshgrid(tensors, indexing)
```

Arguments

- **tensors** (list of Tensor) list of scalars or 1 dimensional tensors. Scalars will be treated as (1,).
- **indexing** (str, optional): the indexing mode, either “xy” or “ij”, defaults to “ij”. See warning for future changes. If “xy” is selected, the first dimension corresponds to the cardinality of the second input and the second dimension corresponds to the cardinality of the first input. If “ij” is selected, the dimensions are in the same order as the cardinality of the inputs.

Warning

In the future `torch_meshgrid` will transition to indexing='xy' as the default. This issue tracks this issue with the goal of migrating to NumPy’s behavior.

Examples

```
if (torch_is_installed()) {
    x = torch_tensor(c(1, 2, 3))
    y = torch_tensor(c(4, 5, 6))
    out = torch_meshgrid(list(x, y))
    out
}
```

torch_min  Min

Description

Min
Arguments

self (Tensor) the input tensor.
dim (int) the dimension to reduce.
keepdim (bool) whether the output tensor has dim retained or not.
out (tuple, optional) the tuple of two output tensors (min, min_indices)
other (Tensor) the second input tensor

min(input) -> Tensor

Returns the minimum value of all elements in the input tensor.

min(input, dim, keepdim=False, out=NULL) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the minimum value of each row of the input tensor in the given dimension dim. And indices is the index location of each minimum value found (argmin).

Warning

indices does not necessarily contain the first occurrence of each minimal value found, unless it is unique. The exact implementation details are device-specific. Do not expect the same result when run on CPU and GPU in general.

If keepdim is TRUE, the output tensors are of the same size as input except in the dimension dim where they are of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output tensors having 1 fewer dimension than input.

min(input, other, out=NULL) -> Tensor

Each element of the tensor input is compared with the corresponding element of the tensor other and an element-wise minimum is taken. The resulting tensor is returned.

The shapes of input and other don’t need to match, but they must be broadcastable.

\[
\text{out}_i = \text{min}(\text{tensor}_i, \text{other}_i)
\]

Note

When the shapes do not match, the shape of the returned output tensor follows the broadcasting rules.

Examples

if (torch_is_installed()) {

    a = torch_randn(c(1, 3))
    a
    torch_min(a)
```python
a = torch.randn(c(4, 4))
a
torch_min(a, dim = 1)

a = torch.randn(c(4))
a
b = torch.randn(c(4))
b
torch_min(a, other = b)
}
```

---

**torch_minimum**

**Minimum**

**Description**

Minimum

**Usage**

`torch_minimum(self, other)`

**Arguments**

- **self** *(Tensor)* the input tensor.
- **other** *(Tensor)* the second input tensor

**minimum(input, other, *, out=None) -> Tensor**

Computes the element-wise minimum of `input` and `other`.

**Note**

If one of the elements being compared is a NaN, then that element is returned. `torch_minimum()` is not supported for tensors with complex dtypes.

**Examples**

```python
if (torch_is_installed()) {

a <- torch_tensor(c(1, 2, -1))
b <- torch_tensor(c(3, 0, 4))
torch_minimum(a, b)
}
```
torch_mm

Description
Mm

Usage
torch_mm(self, mat2)

Arguments
self (Tensor) the first matrix to be multiplied
mat2 (Tensor) the second matrix to be multiplied

mm(input, mat2, out=NULL) -> Tensor
Performs a matrix multiplication of the matrices input and mat2.
If input is a \((n \times m)\) tensor, mat2 is a \((m \times p)\) tensor, out will be a \((n \times p)\) tensor.

Note
This function does not broadcast. For broadcasting matrix products, see torch_matmul.

Examples
if (torch_is_installed()) {
  mat1 = torch_randn(c(2, 3))
  mat2 = torch_randn(c(3, 3))
  torch_mm(mat1, mat2)
}

torch_mode

Description
Mode

Usage
torch_mode(self, dim = -1L, keepdim = FALSE)
torch_movedim

Arguments

self (Tensor) the input tensor.
dim (int) the dimension to reduce.
keepdim (bool) whether the output tensor has dim retained or not.

mode(input, dim=-1, keepdim=False, out=NULL) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the mode value of each row of the
input tensor in the given dimension dim, i.e. a value which appears most often in that row, and
indices is the index location of each mode value found.
By default, dim is the last dimension of the input tensor.
If keepdim is TRUE, the output tensors are of the same size as input except in the dimension dim
where they are of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output
tensors having 1 fewer dimension than input.

Note
This function is not defined for torch_cuda.Tensor yet.

Examples

if (torch_is_installed()) {

  a = torch_randint(0, 50, size = list(5))
  a
  torch_mode(a, 1)
}

torch_movedim Movedim

Description
Movedim

Usage
torch_movedim(self, source, destination)

Arguments

self (Tensor) the input tensor.
source (int or tuple of ints) Original positions of the dims to move. These must be unique.
destination (int or tuple of ints) Destination positions for each of the original dims. These must also be unique.
movedim(input, source, destination) -> Tensor

Moves the dimension(s) of input at the position(s) in source to the position(s) in destination.
Other dimensions of input that are not explicitly moved remain in their original order and appear
at the positions not specified in destination.

Examples

```r
if (torch_is_installed()) {
  t <- torch_randn(c(3,2,1))
  t
  torch_movedim(t, 2, 1)$shape
  torch_movedim(t, 2, 1)
  torch_movedim(t, c(2, 3), c(1, 2))$shape
  torch_movedim(t, c(2, 3), c(1, 2))
}
```

torch_mul  

Description

Mul

Usage

```r
torch_mul(self, other)
```

Arguments

- **self**: (Tensor) the first multiplicand tensor
- **other**: (Tensor) the second multiplicand tensor

mul(input, other, out=NULL)

Multiplies each element of the input input with the scalar other and returns a new resulting tensor.

\[ \text{out}_i = \text{other} \times \text{input}_i \]

If input is of type FloatTensor or DoubleTensor, other should be a real number, otherwise it
should be an integer

Each element of the tensor input is multiplied by the corresponding element of the Tensor other.
The resulting tensor is returned.
The shapes of input and other must be broadcastable.

\[ \text{out}_i = \text{input}_i \times \text{other}_i \]
Examples

```r
if (torch_is_installed()) {

a = torch_randn(c(3))
a
torch_mul(a, 100)

a = torch_randn(c(4, 1))
a
b = torch_randn(c(1, 4))
b
torch_mul(a, b)
}
```

--

**torch_multinomial**  
**Multinomial**

**Description**

Multinomial

**Usage**

```r
torch_multinomial(self, num_samples, replacement = FALSE, generator = NULL)
```

**Arguments**

- `self` (Tensor): the input tensor containing probabilities
- `num_samples` (int): number of samples to draw
- `replacement` (bool, optional): whether to draw with replacement or not
- `generator` (torch.Generator, optional): a pseudorandom number generator for sampling

**multinomial(input, num_samples, replacement=False, *, generator=NULL, out=NULL) -> LongTensor**

Returns a tensor where each row contains `num_samples` indices sampled from the multinomial probability distribution located in the corresponding row of tensor `input`.

**Note**

The rows of `input` do not need to sum to one (in which case we use the values as weights), but must be non-negative, finite and have a non-zero sum.

Indices are ordered from left to right according to when each was sampled (first samples are placed in first column).

If input is a vector, out is a vector of size num_samples.
If input is a matrix with \( m \) rows, out is an matrix of shape \( (m \times \text{num}_\text{samples}) \).

If replacement is \text{TRUE}, samples are drawn with replacement.

If not, they are drawn without replacement, which means that when a sample index is drawn for a row, it cannot be drawn again for that row.

When drawn without replacement, `\text{num}_\text{samples}` must be lower than the number of non-zero elements in `input` (or the min number of non-zero elements in each row of `input` if it is a matrix).

**Examples**

```r
if (torch_is_installed()) {
  weights = torch_tensor(c(0, 10, 3, 0), dtype=torch_float())  # create a tensor of weights
  torch_multinomial(weights, 2)
  torch_multinomial(weights, 4, replacement=TRUE)
}
```

---

**torch_multiply**

*Multiply*

**Description**

Multiply

**Usage**

`torch_multiply(self, other)`

**Arguments**

- `self` `(Tensor)` the first multiplicand tensor
- `other` `(Tensor)` the second multiplicand tensor

**multiply**

Alias for `torch_mul()`.
torch_mv

Description

Mv

Usage

torch_mv(self, vec)

Arguments

self (Tensor) matrix to be multiplied
vec (Tensor) vector to be multiplied

mv(input, vec, out=NULL) -> Tensor

Performs a matrix-vector product of the matrix input and the vector vec.
If input is a \((n \times m)\) tensor, vec is a 1-D tensor of size \(m\), out will be 1-D of size \(n\).

Note

This function does not broadcast.

Examples

if (torch_is_installed()) {

    mat = torch_randn(c(2, 3))
    vec = torch_randn(c(3))
    torch_mv(mat, vec)
}

torch_mvlgamma

Description

Mvlgamma

Usage

torch_mvlgamma(self, p)
torch_nanquantile

**Description**

Nanquantile

**Usage**

torch_nanquantile(
    self,
    q,
    dim = NULL,
    keepdim = FALSE,
    interpolation = "linear"
)

**Arguments**

- **self** (Tensor) the input tensor.
- **q** (float or Tensor) a scalar or 1D tensor of quantile values in the range [0, 1].
- **dim** (int) the dimension to reduce.
- **keepdim** (bool) whether the output tensor has dim retained or not.
- **interpolation** The interpolation method.
nanquantile(input, q, dim=None, keepdim=FALSE, *, out=None) -> Tensor

This is a variant of torch_quantile() that "ignores" NaN values, computing the quantiles q as if NaN values in input did not exist. If all values in a reduced row are NaN then the quantiles for that reduction will be NaN. See the documentation for torch_quantile().

Examples

```r
if (torch_is_installed()) {
    t <- torch_tensor(c(NaN, 1, 2))
    t$quantile(0.5)
    t$nanquantile(0.5)
    t <- torch_tensor(rbind(c(NaN, NaN), c(1, 2)))
    t
    t$nanquantile(0.5, dim=1)
    t$nanquantile(0.5, dim=2)
    torch_nanquantile(t, 0.5, dim = 1)
    torch_nanquantile(t, 0.5, dim = 2)
}
```

torch_nansum  Nansum

Description

Nansum

Usage

torch_nansum(self, dim = NULL, keepdim = FALSE, dtype = NULL)

Arguments

- **self**  (Tensor) the input tensor.
- **dim**  (int or tuple of ints) the dimension or dimensions to reduce.
- **keepdim**  (bool) whether the output tensor has dim retained or not.
- **dtype**  the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

nansum(input, *, dtype=None) -> Tensor

Returns the sum of all elements, treating Not a Numbers (NaNs) as zero.
nansum(input, dim, keepdim=FALSE, *, dtype=None) -> Tensor

Returns the sum of each row of the input tensor in the given dimension dim, treating Not a Numbers (NaNs) as zero. If dim is a list of dimensions, reduce over all of them.

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

Examples

```r
if (torch_is_installed()) {
  a <- torch_tensor(c(1., 2., NaN, 4.))
  torch_nansum(a)
  
  torch_nansum(torch_tensor(c(1., NaN)))
  a <- torch_tensor(rbind(c(1, 2), c(3., NaN)))
  torch_nansum(a)
  torch_nansum(a, dim=1)
  torch_nansum(a, dim=2)
}
```

---

torch_narrow

Narrow

Description

Narrow

Usage

```
torch_narrow(self, dim, start, length)
```

Arguments

- `self` (Tensor) the tensor to narrow
- `dim` (int) the dimension along which to narrow
- `start` (int) the starting dimension
- `length` (int) the distance to the ending dimension

narrow(input, dim, start, length) -> Tensor

Returns a new tensor that is a narrowed version of input tensor. The dimension dim is input from start to start + length. The returned tensor and input tensor share the same underlying storage.
torch_ne
Examples
if (torch_is_installed()) {

x = torch_tensor(matrix(c(1:9), ncol = 3, byrow= TRUE))
torch_narrow(x, 1, 1, 2)
torch_narrow(x, 2, 2, 2)
}

torch_ne other

Description
Ne

Usage
torch_ne(self, other)

Arguments
self (Tensor) the tensor to compare
other (Tensor or float) the tensor or value to compare

ne(input, other, out=NULL) -> Tensor

Computes input ≠ other element-wise.
The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

Examples
if (torch_is_installed()) {

torch_ne(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
        torch_tensor(matrix(rep(c(1,4), each = 2), ncol = 2, byrow=TRUE)))
}
torch_neg

Description

Neg

Usage

torch_neg(self)

Arguments

self (Tensor) the input tensor.

Examples

if (torch_is_installed()) {
  a = torch_randn(c(5))
  a
  torch_neg(a)
}

torch_negative

Description

Negative

Usage

torch_negative(self)

Arguments

self (Tensor) the input tensor.

negative(input, *, out=None) -> Tensor

Returns a new tensor with the negative of the elements of input.

\[ \text{out} = -1 \times \text{input} \]

Examples

if (torch_is_installed()) {
  a = torch_randn(c(5))
  a
  torch_neg(a)
  torch_negative(a)
}
torch_nextafter

Description

Nextafter

Usage

torch_nextafter(self, other)

Arguments

self (Tensor) the first input tensor
other (Tensor) the second input tensor

nextafter(input, other, *, out=None) -> Tensor

Return the next floating-point value after input towards other, elementwise.

The shapes of input and other must be broadcastable.

Examples

if (torch_is_installed()) {
    eps <- torch_finfo(torch_float32())$eps
    torch_nextafter(torch_tensor(c(1, 2)), torch_tensor(c(2, 1))) == torch_tensor(c(eps + 1, 2 - eps))
}

torch_nonzero

Description

Nonzero elements of tensors.

Usage

torch_nonzero(self, as_list = FALSE)
torch_norm

Arguments

self (Tensor) the input tensor.

as_list If FALSE, the output tensor containing indices. If TRUE, one 1-D tensor for each dimension, containing the indices of each nonzero element along that dimension.

When as_list is FALSE (default):
Returns a tensor containing the indices of all non-zero elements of input. Each row in the result contains the indices of a non-zero element in input. The result is sorted lexicographically, with the last index changing the fastest (C-style).
If input has \( n \) dimensions, then the resulting indices tensor out is of size \((z \times n)\), where \( z \) is the total number of non-zero elements in the input tensor.

When as_list is TRUE:
Returns a tuple of 1-D tensors, one for each dimension in input, each containing the indices (in that dimension) of all non-zero elements of input.
If input has \( n \) dimensions, then the resulting tuple contains \( n \) tensors of size \( z \), where \( z \) is the total number of non-zero elements in the input tensor.
As a special case, when input has zero dimensions and a nonzero scalar value, it is treated as a one-dimensional tensor with one element.

Examples

```r
if (torch_is_installed()) {

  torch_nonzero(torch_tensor(c(1, 1, 1, 0, 1)))
}
```

Description

Norm

Usage

torch_norm(self, p = 2L, dim, keepdim = FALSE, dtype)

Arguments

self (Tensor) the input tensor

p (int, float, inf, -inf, 'fro', 'nuc', optional) the order of norm. Default: ‘fro’ The following norms can be calculated: ===== ============================ ord matrix norm vector norm ===== ============================ NULL Frobenius norm 2-norm ‘fro’ Frobenius norm – ‘nuc’ nuclear norm – Other as vec norm when dim is NULL sum(abs(x)ord)(1./ord)
torch_normal

**dim**
(int, 2-tuple of ints, 2-list of ints, optional) If it is an int, vector norm will be calculated, if it is 2-tuple of ints, matrix norm will be calculated. If the value is NULL, matrix norm will be calculated when the input tensor only has two dimensions, vector norm will be calculated when the input tensor only has one dimension. If the input tensor has more than two dimensions, the vector norm will be applied to last dimension.

**keepdim**
(bool, optional) whether the output tensors have dim retained or not. Ignored if dim = NULL and out = NULL. Default: FALSE Ignored if dim = NULL and out = NULL.

**dtype**
(torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to 'dtype' while performing the operation. Default: NULL.

**TEST**
Returns the matrix norm or vector norm of a given tensor.

**Examples**

```r
if (torch_is_installed()) {
    a <- torch_arange(1, 9, dtype = torch_float())
    b <- a$reshape(list(3, 3))
    torch_norm(a)
    torch_norm(b)
    torch_norm(a, Inf)
    torch_norm(b, Inf)
}
```

---

**torch_normal**

**Normal**

**Description**
Normal
Normal distributed

**Usage**

```
torch_normal(mean, std, size = NULL, generator = NULL, ...)
```

**Arguments**

**mean**
(tensor or scalar double) Mean of the normal distribution. If this is a `torch_tensor()` then the output has the same dim as `mean` and it represents the per-element mean. If it’s a scalar value, it’s reused for all elements.
std (tensor or scalar double) The standard deviation of the normal distribution. If this is a \texttt{torch_tensor()} then the output has the same size as \texttt{std} and it represents the per-element standard deviation. If it’s a scalar value, it’s reused for all elements.

size (integers, optional) only used if both \texttt{mean} and \texttt{std} are scalars.

generator a random number generator created with \texttt{torch_generator()}. If NULL a default generator is used.

... Tensor option parameters like dtype, layout, and device. Can only be used when \texttt{mean} and \texttt{std} are both scalar numerics.

\texttt{normal(mean, std, *) -> Tensor}

Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.

The \texttt{mean} is a tensor with the mean of each output element’s normal distribution

The \texttt{std} is a tensor with the standard deviation of each output element’s normal distribution

The shapes of \texttt{mean} and \texttt{std} don’t need to match, but the total number of elements in each tensor need to be the same.

\texttt{normal(mean=0.0, std) -> Tensor}

Similar to the function above, but the means are shared among all drawn elements.

\texttt{normal(mean, std=1.0) -> Tensor}

Similar to the function above, but the standard-deviations are shared among all drawn elements.

\texttt{normal(mean, std, size, *) -> Tensor}

Similar to the function above, but the means and standard deviations are shared among all drawn elements. The resulting tensor has size given by \texttt{size}.

\textbf{Note}

When the shapes do not match, the shape of \texttt{mean} is used as the shape for the returned output tensor

\textbf{Examples}

```python
if (torch_is_installed()) {

    torch_normal(mean=0, std=torch_arange(1, 0, -0.1) + 1e-6)
    torch_normal(mean=0.5, std=torch_arange(1., 6.))
    torch_normal(mean=torch_arange(1., 6.))
    torch_normal(2, 3, size=c(1, 4))
}
```
torch_not_equal

Not_equal

Description
Not_equal

Usage
torch_not_equal(self, other)

Arguments
self (Tensor) the tensor to compare
other (Tensor or float) the tensor or value to compare

not_equal(input, other, *, out=None) -> Tensor

Alias for torch_ne().

torch_ones

Ones

Description
Ones

Usage
torch_ones(
    ..., names = NULL,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)

Arguments
... (int...) a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
names optional names for the dimensions
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
torch_ones_like

layout  (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device  (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad  (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

ones(*size, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Returns a tensor filled with the scalar value 1, with the shape defined by the variable argument size.

Examples

if (torch_is_installed()) {
    torch_ones(c(2, 3))
    torch_ones(c(5))
}

torch_ones_like  Ones_like

Description

Ones_like

Usage

torch_ones_like(
    input,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)

Arguments

input  (Tensor) the size of input will determine size of the output tensor.
dtype  (torch.dtype, optional) the desired data type of returned Tensor. Default: if NULL, defaults to the dtype of input.
layout  (torch.layout, optional) the desired layout of returned tensor. Default: if NULL, defaults to the layout of input.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, defaults to the device of input.

requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

memory_format (torch.memory_format, optional) the desired memory format of returned Tensor. Default: torch.preserve_format.

ones_like(input, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, memory_format=torch.preserve_format) -> Tensor

Returns a tensor filled with the scalar value 1, with the same size as input. torch.ones_like(input) is equivalent to torch.ones(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Warning

As of 0.4, this function does not support an out keyword. As an alternative, the old torch.ones_like(input, out=output) is equivalent to torch.ones(input.size(), out=output).

Examples

```python
if (torch_is_installed()) {

    input = torch_empty(c(2, 3))
    torch.ones_like(input)
}
```

orgqr(input, input2) -> Tensor

Computes the orthogonal matrix $Q$ of a QR factorization, from the (input, input2) tuple returned by torch_geqrf.

This directly calls the underlying LAPACK function $\text{orgqr}$. See LAPACK documentation for orgqr_ for further details.
torch.ormqr  

Description

Ormqr

Usage

torch.ormqr(self, input2, input3, left = TRUE, transpose = FALSE)

Arguments

self  (Tensor) the a from torch.geqrf.
input2 (Tensor) the tau from torch.geqrf.
input3 (Tensor) the matrix to be multiplied.
left see LAPACK documentation
transpose see LAPACK documentation

ormqr(input, input2, input3, left=TRUE, transpose=False) -> Tensor

Multiplies mat (given by input3) by the orthogonal Q matrix of the QR factorization formed by torch.geqrf() that is represented by (a, tau) (given by (input, input2)).

This directly calls the underlying LAPACK function ?ormqr.

torch.outer  

Description

Outer

Usage

torch.outer(self, vec2)

Arguments

self  (Tensor) 1-D input vector
vec2  (Tensor) 1-D input vector

outer(input, vec2, *, out=None) -> Tensor

Outer product of input and vec2. If input is a vector of size n and vec2 is a vector of size m, then out must be a matrix of size \(n \times m\).
torch_pdist

Note

This function does not broadcast.

Examples

```r
if (torch_is_installed()) {
  v1 <- torch_arange(1., 5.)
  v2 <- torch_arange(1., 4.)
  torch_outer(v1, v2)
}
```

Description

Pdist

Usage

```r
torch_pdist(self, p = 2L)
```

Arguments

- **self**: NA input tensor of shape $N \times M$.
- **p**: NA p value for the p-norm distance to calculate between each vector pair $\in [0, \infty]$.

pdist(input, p=2) -> Tensor

Computes the p-norm distance between every pair of row vectors in the input. This is identical to the upper triangular portion, excluding the diagonal, of `torch_norm(input[:, NULL] - input, dim=2, p=p)`. This function will be faster if the rows are contiguous.

If input has shape $N \times M$ then the output will have shape $\frac{1}{2}N(N - 1)$.

This function is equivalent to `scipy.spatial.distance.pdist(input, 'minkowski', p=p)` if $p \in (0, \infty)$. When $p = 0$ it is equivalent to `scipy.spatial.distance.pdist(input, 'hamming')` $\times M$. When $p = \infty$, the closest scipy function is `scipy.spatial.distance.pdist(xn, lambda x, y: np.abs(x - y).max(`
torch.pinverse

Description

Pinverse

Usage

torch.pinverse(self, rcond = 1e-15)

Arguments

self (Tensor) The input tensor of size (*, m, n) where * is zero or more batch dimensions
rcond (float) A floating point value to determine the cutoff for small singular values. Default: 1e-15

pinverse(input, rcond=1e-15) -> Tensor

Calculates the pseudo-inverse (also known as the Moore-Penrose inverse) of a 2D tensor. Please look at Moore-Penrose inverse for more details

Note

This method is implemented using the Singular Value Decomposition.

The pseudo-inverse is not necessarily a continuous function in the elements of the matrix \([1]\). Therefore, derivatives are not always existent, and exist for a constant rank only \([2]\). However, this method is backprop-able due to the implementation by using SVD results, and could be unstable. Double-backward will also be unstable due to the usage of SVD internally. See `torch.svd` for more details.

Examples

```plaintext
if (torch_is_installed()) {

    input = torch.randn(c(3, 5))
    input
    torch.pinverse(input)
    # Batched pinverse example
    a = torch.randn(c(2,6,3))
    b = torch.pinverse(a)
    torch.matmul(b, a)
}
```
torch_pixel_shuffle

**Description**

Pixel_shuffle

**Usage**

torch_pixel_shuffle(self, upscale_factor)

**Arguments**

- `self` (Tensor) the input tensor
- `upscale_factor` (int) factor to increase spatial resolution by

**Rearranges elements in a tensor of shape**

\[ (*) \times C \times r^2, H, W \] to:

Rearranges elements in a tensor of shape \((*,C \times r^2,H,W)\) to a tensor of shape \((*,C,H \times r,W \times r)\).

See `torch.nn.PixelShuffle` for details.

**Examples**

```python
if (torch_is_installed()) {

    input = torch_randn(c(1, 9, 4, 4))
    output = nnf_pixel_shuffle(input, 3)
    print(output$size())
}
```

torch_poisson

**Description**

Poisson

**Usage**

torch_poisson(self, generator = NULL)

**Arguments**

- `self` (Tensor) the input tensor containing the rates of the Poisson distribution
- `generator` (torch.Generator, optional) a pseudorandom number generator for sampling
poisson(input *, generator=NULL) -> Tensor

Returns a tensor of the same size as input with each element sampled from a Poisson distribution with rate parameter given by the corresponding element in input i.e.,

$$\text{out}_i \sim \text{Poisson}(\text{input}_i)$$

Examples

```r
if (torch_is_installed()) {
  rates = torch_rand(c(4, 4)) * 5  # rate parameter between 0 and 5
  torch_poisson(rates)
}
```

torch_polar

Polar

Description

Polar

Usage

torch_polar(abs, angle)

Arguments

- `abs` (Tensor) The absolute value the complex tensor. Must be float or double.
- `angle` (Tensor) The angle of the complex tensor. Must be same dtype as abs.

polar(abs, angle, *, out=None) -> Tensor

Constructs a complex tensor whose elements are Cartesian coordinates corresponding to the polar coordinates with absolute value abs and angle angle.

$$\text{out} = \text{abs} \cdot \cos(\text{angle}) + \text{abs} \cdot \sin(\text{angle}) \cdot j$$

Examples

```r
if (torch_is_installed()) {
  abs <- torch_tensor(c(1, 2), dtype=torch_float64())
  angle <- torch_tensor(c(pi / 2, 5 * pi / 4), dtype=torch_float64())
  z <- torch_polar(abs, angle)
  z
}
```
**torch_polygamma**

**Polygamma**

**Description**

Polygamma

**Usage**

torch_polygamma(n, input)

**Arguments**

- **n** (int) the order of the polygamma function
- **input** (Tensor) the input tensor.

polygamma(n, input, out=NULL) -> Tensor

Computes the $n^{th}$ derivative of the digamma function on input. $n \geq 0$ is called the order of the polygamma function.

$$
\psi^{(n)}(x) = \frac{d^{(n)}}{dx^{(n)}} \psi(x)
$$

**Note**

This function is not implemented for $n \geq 2$.

**Examples**

```r
if (torch_is_installed()) {
  ## Not run:
  a = torch_tensor(c(1, 0.5))
  torch_polygamma(1, a)

  ## End(Not run)
}
```
torch_pow

Description

Pow

Usage

torch_pow(self, exponent)

Arguments

self (float) the scalar base value for the power operation
exponent (float or tensor) the exponent value

pow(input, exponent, out=NULL) -> Tensor

Takes the power of each element in input with exponent and returns a tensor with the result.

exponent can be either a single float number or a Tensor with the same number of elements as input.

When exponent is a scalar value, the operation applied is:

\[ \text{out}_i = x_i^{\text{exponent}} \]

When exponent is a tensor, the operation applied is:

\[ \text{out}_i = x_i^{\text{exponent}_i} \]

When exponent is a tensor, the shapes of input and exponent must be broadcastable.

pow(self, exponent, out=NULL) -> Tensor

self is a scalar float value, and exponent is a tensor. The returned tensor out is of the same shape as exponent

The operation applied is:

\[ \text{out}_i = \text{self}^{\text{exponent}_i} \]
torch_prod

Examples

```cpp
if (torch_is_installed()) {
    a = torch_randn(c(4))
a
    torch_pow(a, 2)
exp <- torch_arange(1, 5)
a <- torch_arange(1, 5)
a
exp
    torch_pow(a, exp)

    exp <- torch_arange(1, 5)
base <- 2
    torch_pow(base, exp)
}
```

torch_prod

**Prod**

Description

Prod

Usage

```
torch_prod(self, dim, keepdim = FALSE, dtype = NULL)
```

Arguments

- **self** *(Tensor)* the input tensor.
- **dim** *(int)* the dimension to reduce.
- **keepdim** *(bool)* whether the output tensor has `dim` retained or not.
- **dtype** *(torch.dtype, optional)* the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

prod(input, dtype=NULL) -> Tensor

Returns the product of all elements in the input tensor.

prod(input, dim, keepdim=False, dtype=NULL) -> Tensor

Returns the product of each row of the input tensor in the given dimension `dim`

If `keepdim` is `TRUE`, the output tensor is of the same size as `input` except in the dimension `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 fewer dimension than `input`. 
torch_promote_types

**Examples**

```cpp
if (torch_is_installed()) {
    a = torch_randn(c(1, 3))
    a
    torch_prod(a)

    a = torch_randn(c(4, 2))
    a
    torch_prod(a, 1)
}
```

---

**torch_promote_types**  
*Promote_types*

**Description**

Promote_types

**Usage**

```cpp
torch_promote_types(type1, type2)
```

**Arguments**

- **type1** *(torch.dtype)*
- **type2** *(torch.dtype)*

**promote_types(type1, type2) -> dtype**

Returns the torch_dtype with the smallest size and scalar kind that is not smaller nor of lower kind than either type1 or type2. See type promotion documentation for more information on the type promotion logic.

**Examples**

```cpp
if (torch_is_installed()) {
    torch_promote_types(torch_int32(), torch_float32())
    torch_promote_types(torch_uint8(), torch_long())
}
```
torch_qr

Description
Qr

Usage
torch_qr(self, some = TRUE)

Arguments
self (Tensor) the input tensor of size (*, m, n) where * is zero or more batch dimensions consisting of matrices of dimension m × n.
some (bool, optional) Set to TRUE for reduced QR decomposition and FALSE for complete QR decomposition.

qr(input, some=TRUE, out=NULL) -> (Tensor, Tensor)

Computes the QR decomposition of a matrix or a batch of matrices input, and returns a namedtuple (Q, R) of tensors such that input = QR with Q being an orthogonal matrix or batch of orthogonal matrices and R being an upper triangular matrix or batch of upper triangular matrices.

If some is TRUE, then this function returns the thin (reduced) QR factorization. Otherwise, if some is FALSE, this function returns the complete QR factorization.

Note
precision may be lost if the magnitudes of the elements of input are large
While it should always give you a valid decomposition, it may not give you the same one across platforms - it will depend on your LAPACK implementation.

Examples
if (torch_is_installed()) {
    a = torch_tensor(matrix(c(12., -51, 4, 6, 167, -68, -4, 24, -41), ncol = 3, byrow = TRUE))
    out = torch_qr(a)
    q = out[[1]]
    r = out[[2]]
    torch_mm(q, r)$round()
    torch_mm(q$t(), q)$round()
}
torch_qscheme  
Creates the corresponding Scheme object

**Description**

Creates the corresponding Scheme object

**Usage**

```python
torch_per_channel_affine()
torch_per_tensor_affine()
torch_per_channel_symmetric()
torch_per_tensor_symmetric()
```

**torch_quantile**  
Quantile

**Description**

Quantile

**Usage**

```python
torch_quantile(self, q, dim = NULL, keepdim = FALSE, interpolation = "linear")
```

**Arguments**

- `self` (Tensor) the input tensor.
- `q` (float or Tensor) a scalar or 1D tensor of quantile values in the range [0, 1]
- `dim` (int) the dimension to reduce.
- `keepdim` (bool) whether the output tensor has `dim` retained or not.
- `interpolation` (str) The interpolation method.

**torch_quantile(input, q) -> Tensor**

Returns the q-th quantiles of all elements in the input tensor, doing a linear interpolation when the q-th quantile lies between two data points.
quantile(input, q, dim=None, keepdim=FALSE, *, out=None) -> Tensor

Returns the q-th quantiles of each row of the input tensor along the dimension dim, doing a linear interpolation when the q-th quantile lies between two data points. By default, dim is None resulting in the input tensor being flattened before computation.

If keepdim is TRUE, the output dimensions are of the same size as input except in the dimensions being reduced (dim or all if dim is NULL) where they have size 1. Otherwise, the dimensions being reduced are squeezed (see torch_squeeze). If q is a 1D tensor, an extra dimension is prepended to the output tensor with the same size as q which represents the quantiles.

Examples

```python
if (torch_is_installed()) {
    a <- torch_randn(c(1, 3))
    a
    q <- torch_tensor(c(0, 0.5, 1))
    torch_quantile(a, q)

    a <- torch_randn(c(2, 3))
    a
    q <- torch_tensor(c(0.25, 0.5, 0.75))
    torch_quantile(a, q, dim=1, keepdim=TRUE)
    torch_quantile(a, q, dim=1, keepdim=TRUE)$shape
}
```

torch_quantize_per_channel  

Quantize_per_channel

Description

Quantize_per_channel

Usage

torch_quantize_per_channel(self, scales, zero_points, axis, dtype)

Arguments

- **self** (Tensor) float tensor to quantize
- **scales** (Tensor) float 1D tensor of scales to use, size should match input.size(axis)
- **zero_points** (int) integer 1D tensor of offset to use, size should match input.size(axis)
- **axis** (int) dimension on which apply per-channel quantization
- **dtype** (torch.dtype) the desired data type of returned tensor. Has to be one of the quantized dtypes: torch_quint8, torch qint8, torch qint32
quantize_per_channel(input, scales, zero_points, axis, dtype) -> Tensor

Converts a float tensor to per-channel quantized tensor with given scales and zero points.

Examples

```r
if (torch_is_installed()) {
  x = torch_tensor(matrix(c(-1.0, 0.0, 1.0, 2.0), ncol = 2, byrow = TRUE))
  torch_quantize_per_channel(x, torch_tensor(c(0.1, 0.01)),
                             torch_tensor(c(10L, 0L)), 0, torch_quint8())
  torch_quantize_per_channel(x, torch_tensor(c(0.1, 0.01)),
                             torch_tensor(c(10L, 0L)), 0, torch_quint8())$int_repr()
}
```

torch_quantize_per_tensor

Quantize_per_tensor

Description

Quantize_per_tensor

Usage

```r
torch_quantize_per_tensor(self, scale, zero_point, dtype)
```

Arguments

- `self` (Tensor) float tensor to quantize
- `scale` (float) scale to apply in quantization formula
- `zero_point` (int) offset in integer value that maps to float zero
- `dtype` (torch.dtype) the desired data type of returned tensor. Has to be one of the quantized dtypes: torch_quint8, torch.qint8, torch.qint32

quantize_per_tensor(input, scale, zero_point, dtype) -> Tensor

Converts a float tensor to quantized tensor with given scale and zero point.

Examples

```r
if (torch_is_installed()) {
  torch_quantize_per_tensor(torch_tensor(c(-1.0, 0.0, 1.0, 2.0)), 0.1, 10, torch_quint8())
  torch_quantize_per_tensor(torch_tensor(c(-1.0, 0.0, 1.0, 2.0)), 0.1, 10, torch_quint8())$int_repr()
}
```
torch_rad2deg

Description

Rad2deg

Usage

torch_rad2deg(self)

Arguments

self (Tensor) the input tensor.

rad2deg(input, *, out=None) -> Tensor

Returns a new tensor with each of the elements of input converted from angles in radians to degrees.

Examples

if (torch_is_installed()) {
    a <- torch_tensor(rbind(c(3.142, -3.142), c(6.283, -6.283), c(1.570, -1.570))
    torch_rad2deg(a)
}

torch_rand

Description

Rand

Usage

torch_rand(
    ..., names = NULL, dtype = NULL, layout = NULL, device = NULL,
    requires_grad = FALSE
)
torch_randint

Arguments

... (int...) a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

names optional dimension names
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).

layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

rand(*size, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Returns a tensor filled with random numbers from a uniform distribution on the interval \([0, 1)\). The shape of the tensor is defined by the variable argument size.

Examples

```python
if (torch_is_installed()) {

    torch_rand(4)
    torch_rand(c(2, 3))
}
```

Description

Randint

Usage

```python
torch_randint(
    low,
    high,
    size,
    generator = NULL,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)
```
torch_randint_like

Arguments

low (int, optional) Lowest integer to be drawn from the distribution. Default: 0.
high (int) One above the highest integer to be drawn from the distribution.
size (tuple) a tuple defining the shape of the output tensor.
generator (torch.Generator, optional) a pseudorandom number generator for sampling
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
memory_format memory format for the resulting tensor.

randint(low=0, high, size, *, generator=NULL, out=NULL, \

dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Returns a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive).

.. note: With the global dtype default (torch_float32), this function returns a tensor with dtype torch_int64.

Examples

if (torch_is_installed()) {

torch_randint(3, 5, list(3))
torch_randint(0, 10, size = list(2, 2))
torch_randint(3, 10, list(2, 2))
}

torch_randint_like Randint_like

Description

Randint_like
Usage

```
torch_randint_like(
    input,
    low,
    high,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)
```

Arguments

- **input** (Tensor): The size of input will determine size of the output tensor.
- **low** (int, optional): Lowest integer to be drawn from the distribution. Default: 0.
- **high** (int): One above the highest integer to be drawn from the distribution.
- **dtype** (torch.dtype, optional): The desired data type of returned Tensor. Default: if NULL, defaults to the dtype of input.
- **layout** (torch.layout, optional): The desired layout of returned tensor. Default: if NULL, defaults to the layout of input.
- **device** (torch.device, optional): The desired device of returned tensor. Default: if NULL, defaults to the device of input.
- **requires_grad** (bool, optional): If autograd should record operations on the returned tensor. Default: FALSE.

```
randint_like(input, low=0, high, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False,
memory_format=torch.preserve_format) -> Tensor
```

Returns a tensor with the same shape as Tensor `input` filled with random integers generated uniformly between `low` (inclusive) and `high` (exclusive).

Note: With the global dtype default (torch.float32), this function returns a tensor with dtype `torch.int64`.

```
torch_randn
```

**Description**

Randn
torch_randn

Usage

```
torch_randn(
   ..., 
   names = NULL,
   dtype = NULL,
   layout = NULL,
   device = NULL,
   requires_grad = FALSE
)
```

Arguments

```
... (int...) a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
names optional names for the dimensions
dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
```

```
randn(*size, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor
```

Returns a tensor filled with random numbers from a normal distribution with mean 0 and variance 1 (also called the standard normal distribution).

```
out_i \sim \mathcal{N}(0, 1)
```

The shape of the tensor is defined by the variable argument size.

Examples

```
if (torch_is_installed()) {
   torch_randn(c(4))
   torch_randn(c(2, 3))
}
```
torch_randn_like

Description

Randn_like

Usage

```python
torch_randn_like(
    input,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)
```

Arguments

- **input**: (Tensor) the size of input will determine size of the output tensor.
- **dtype**: (torch.dtype, optional) the desired data type of returned Tensor. Default: if NULL, defaults to the dtype of input.
- **layout**: (torch.layout, optional) the desired layout of returned tensor. Default: if NULL, defaults to the layout of input.
- **device**: (torch.device, optional) the desired device of returned tensor. Default: if NULL, defaults to the device of input.
- **requires_grad**: (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
- **memory_format**: (torch.memory_format, optional) the desired memory format of returned Tensor. Default: torch_preserve_format()

`randn_like(input, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, memory_format=torch.preserve_format())` -> Tensor

Returns a tensor with the same size as input that is filled with random numbers from a normal distribution with mean 0 and variance 1. `torch_randn_like(input)` is equivalent to `torch_randn(input.size(), dtype=input.dtype, layout=input.layout, device=input.device)`. 
**torch_randperm**

**Randperm**

**Description**

Randperm

**Usage**

```python
torch_randperm(
    n,
    dtype = torch_int64(),
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)
```

**Arguments**

- **n** (int) the upper bound (exclusive)
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: torch_int64.
- **layout** (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

**randperm(n, out=NULL, dtype=torch.int64, layout=torch.strided, device=NULL, requires_grad=False)**

-> LongTensor

Returns a random permutation of integers from 0 to n - 1.

**Examples**

```python
if (torch_is_installed()) {
    torch_randperm(4)
}
```
torch_rand_like

```python
torch_rand_like(input, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, memory_format=torch.preserve_format)
```

**Returns a tensor with the same size as input that is filled with random numbers from a uniform distribution on the interval [0, 1). torch_rand_like(input) is equivalent to torch_rand(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).**
torch_range

**Description**

Range

**Usage**

```python
torch_range(
    start,
    end,
    step = 1,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE
)
```

**Arguments**

- **start** (float) the starting value for the set of points. Default: 0.
- **end** (float) the ending value for the set of points
- **step** (float) the gap between each pair of adjacent points. Default: 1.
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type). If dtype is not given, infer the data type from the other input arguments. If any of start, end, or stop are floating-point, the dtype is inferred to be the default dtype, see ~torch.get_default_dtype. Otherwise, the dtype is inferred to be torch.int64.
- **layout** (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.
- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

**range(start=0, end, step=1, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor**

Returns a 1-D tensor of size \( \left\lfloor \frac{\text{end} - \text{start}}{\text{step}} \right\rfloor + 1 \) with values from start to end with step step. Step is the gap between two values in the tensor.

\[ \text{out}_{i+1} = \text{out}_i + \text{step}. \]
Warning

This function is deprecated in favor of torch_arange.

Examples

```cpp
if (torch_is_installed()) {
    torch_range(1, 4)
    torch_range(1, 4, 0.5)
}
```

<table>
<thead>
<tr>
<th>torch_real</th>
<th>Real</th>
</tr>
</thead>
</table>

Description

Real

Usage

torch_real(self)

Arguments

self (Tensor) the input tensor.

real(input) -> Tensor

Returns the real part of the input tensor. If input is a real (non-complex) tensor, this function just returns it.

Warning

Not yet implemented for complex tensors.

```cpp
out_i = real(input_i)
```

Examples

```cpp
if (torch_is_installed()) {
    # Not run:
    torch_real(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))
    # End(Not run)
}
```
torch_reciprocal

Reciprocal

Description
Reciprocal

Usage
torch_reciprocal(self)

Arguments
self (Tensor) the input tensor.

reciprocal(input, out=NULL) -> Tensor
Returns a new tensor with the reciprocal of the elements of input

\[
out_i = \frac{1}{input_i}
\]

Examples
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_reciprocal(a)
}

torch_reduction

Creates the reduction objet

Description
Creates the reduction objet

Usage
torch_reduction_sum()
torch_reduction_mean()
torch_reduction_none()
torch_relu

Description
Relu

Usage
torch_relu(self)

Arguments
self the input tensor

relu(input) -> Tensor
Computes the relu transformation.

torch_relu_

Description
Relu_

Usage
torch_relu_(self)

Arguments
self the input tensor

relu_(input) -> Tensor
In-place version of torch_relu().
torch_remainder

Description

Remainder

Usage

torch_remainder(self, other)

Arguments

self (Tensor) the dividend
other (Tensor or float) the divisor that may be either a number or a Tensor of the same shape as the dividend

remainder(input, other, out=NULL) -> Tensor

Computes the element-wise remainder of division.

The divisor and dividend may contain both for integer and floating point numbers. The remainder has the same sign as the divisor.

When other is a tensor, the shapes of input and other must be broadcastable.

Examples

if (torch_is_installed()) {
    torch_remainder(torch_tensor(c(-3., -2, -1, 1, 2, 3)), 2)
    torch_remainder(torch_tensor(c(1., 2, 3, 4, 5)), 1.5)
}

torch_renorm

Description

Renorm

Usage

torch_renorm(self, p, dim, maxnorm)
torch_repeat_interleave

Repeat_interleave

Description

Repeat_interleave

Usage

torch_repeat_interleave(self, repeats, dim = NULL, output_size = NULL)

Arguments

- **self** (Tensor) the input tensor.
- **repeats** (Tensor or int) The number of repetitions for each element. repeats is broadcasted to fit the shape of the given axis.
- **dim** (int, optional) The dimension along which to repeat values. By default, use the flattened input array, and return a flat output array.
- **output_size** (int, optional) – Total output size for the given axis (e.g. sum of repeats). If given, it will avoid stream synchronization needed to calculate output shape of the tensor.
repeat_interleave(input, repeats, dim=NULL) -> Tensor

Repeat elements of a tensor.

Warning

This is different from `torch_Tensor.repeat` but similar to `numpy.repeat`.

repeat_interleave(repeats) -> Tensor

If the repeats is tensor([n1, n2, n3, ...]), then the output will be tensor([0, 0, ..., 1, 1, ..., 2, 2, ..., ...])
where 0 appears n1 times, 1 appears n2 times, 2 appears n3 times, etc.

Examples

```r
if (torch_is_installed()) {
  ## Not run:
  x = torch_tensor(c(1, 2, 3))
x$repeat_interleave(2)
y = torch_tensor(matrix(c(1, 2, 3, 4), ncol = 2, byrow=TRUE))
torch_repeat_interleave(y, 2)
torch_repeat_interleave(y, 3, dim=1)
torch_repeat_interleave(y, torch_tensor(c(1, 2)), dim=1)

  ## End(Not run)
}
```

torch_reshape

Reshape

Description

Reshape

Usage

torch_reshape(self, shape)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>(Tensor) the tensor to be reshaped</td>
</tr>
<tr>
<td>shape</td>
<td>(tuple of ints) the new shape</td>
</tr>
</tbody>
</table>
reshape(input, shape) -> Tensor

Returns a tensor with the same data and number of elements as input, but with the specified shape. When possible, the returned tensor will be a view of input. Otherwise, it will be a copy. Contiguous inputs and inputs with compatible strides can be reshaped without copying, but you should not depend on the copying vs. viewing behavior.

See torch.Tensor.view on when it is possible to return a view.

A single dimension may be -1, in which case it's inferred from the remaining dimensions and the number of elements in input.

Examples

```r
if (torch_is_installed()) {
  a <- torch_arange(0, 3)
  torch_reshape(a, list(2, 2))
  b <- torch_tensor(matrix(c(0, 1, 2, 3), ncol = 2, byrow=TRUE))
  torch_reshape(b, list(-1))
}
```

<table>
<thead>
<tr>
<th>torch_result_type</th>
<th>Result_type</th>
</tr>
</thead>
</table>

Description

Result_type

Usage

torch_result_type(tensor1, tensor2)

Arguments

tensor1 (Tensor or Number) an input tensor or number
tensor2 (Tensor or Number) an input tensor or number

result_type(tensor1, tensor2) -> dtype

Returns the torch.dtype that would result from performing an arithmetic operation on the provided input tensors. See type promotion documentation for more information on the type promotion logic.

Examples

```r
if (torch_is_installed()) {
  torch_result_type(tensor1 = torch_tensor(c(1, 2), dtype=torch_int()), tensor2 = 1)
}
```
torch_roll

Description

Roll

Usage

torch_roll(self, shifts, dims = list())

Arguments

self (Tensor) the input tensor.
shifts (int or tuple of ints) The number of places by which the elements of the tensor are shifted. If shifts is a tuple, dims must be a tuple of the same size, and each dimension will be rolled by the corresponding value.
dims (int or tuple of ints) Axis along which to roll.

roll(input, shifts, dims=NULL) -> Tensor

Roll the tensor along the given dimension(s). Elements that are shifted beyond the last position are re-introduced at the first position. If a dimension is not specified, the tensor will be flattened before rolling and then restored to the original shape.

Examples

if (torch_is_installed()) {
  x = torch_tensor(c(1, 2, 3, 4, 5, 6, 7, 8))$view(c(4, 2))
  x
  torch_roll(x, 1, 1)
  torch_roll(x, -1, 1)
  torch_roll(x, shifts=list(2, 1), dims=list(1, 2))
}

torch_rot90

Description

Rot90

Usage

torch_rot90(self, k = 1L, dims = c(0, 1))
torch_round

Arguments

Arguments:

- `self` (Tensor): the input tensor.
- `k` (int): number of times to rotate.
- `dims` (a list or tuple): axis to rotate.

**rot90(input, k, dims) -> Tensor**

Rotate a n-D tensor by 90 degrees in the plane specified by dims axis. Rotation direction is from the first towards the second axis if k > 0, and from the second towards the first for k < 0.

**Examples**

```r
if (torch_is_installed()) {
    x <- torch_arange(1, 4)$view(c(2, 2))
    x
    torch_rot90(x, 1, c(1, 2))
    x <- torch_arange(1, 8)$view(c(2, 2, 2))
    x
    torch_rot90(x, 1, c(1, 2))
}
```

torch_round

**Round**

**Description**

Round

**Usage**

torch_round(self, decimals)

**Arguments**

- `self` (Tensor): the input tensor.
- `decimals` (int): Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

**round(input, out=NULL) -> Tensor**

Returns a new tensor with each of the elements of input rounded to the closest integer.
torch_rrelu_

Examples

if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_round(a)
}

torch_rrelu_ Rrelu_

Description

Rrelu_

Usage

torch_rrelu_(
    self, 
    lower = 0.125, 
    upper = 0.333333333333333, 
    training = FALSE, 
    generator = NULL 
)

Arguments

self the input tensor
lower lower bound of the uniform distribution. Default: 1/8
upper upper bound of the uniform distribution. Default: 1/3
training bool wether it's a training pass. Default: FALSE
generator random number generator

rrelu_(input, lower=1./8, upper=1./3, training=False) -> Tensor

In-place version of torch_rrelu.
torch_rsqrt  \( R_{sqrt} \)

Description

\( R_{sqrt} \)

Usage

\texttt{torch_rsqrt(self)}

Arguments

\begin{itemize}
\item \texttt{self} (Tensor) the input tensor.
\end{itemize}

\texttt{rsqrt(input, out=NULL) \rightarrow Tensor}

Returns a new tensor with the reciprocal of the square-root of each of the elements of \texttt{input}.

\[
\text{out}_i = \frac{1}{\sqrt{\text{input}_i}}
\]

Examples

\begin{verbatim}
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_rsqrt(a)
}
\end{verbatim}

torch_save  \( \text{Saves an object to a disk file.} \)

Description

This function is experimental, don’t use for long term storage.

Usage

\texttt{torch_save(obj, path, ..., compress = TRUE)}
torch_scalar_tensor

Arguments

- **obj**: the saved object
- **path**: a connection or the name of the file to save.
- **compress**: a logical specifying whether saving to a named file is to use "gzip" compression, or one of "gzip", "bzip2" or "xz" to indicate the type of compression to be used. Ignored if file is a connection.

See Also

Other torch_save: torch_load(), torch_serialize()

torch_scalar_tensor  Scalar tensor

Description

Creates a singleton dimension tensor.

Usage

torch_scalar_tensor(value, dtype = NULL, device = NULL, requires_grad = FALSE)

Arguments

- **value**: the value you want to use
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).
- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
Description
Searchsorted

Usage

```
torch_searchsorted(  
    sorted_sequence,  
    self,  
    out_int32 = FALSE,  
    right = FALSE,  
    side = NULL,  
    sorter = list()  
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sorted_sequence</code></td>
<td>(Tensor) N-D or 1-D tensor, containing monotonically increasing sequence on the innermost dimension.</td>
</tr>
<tr>
<td><code>self</code></td>
<td>(Tensor or Scalar) N-D tensor or a Scalar containing the search value(s).</td>
</tr>
<tr>
<td><code>out_int32</code></td>
<td>(bool, optional) – indicate the output data type. <code>torch_int32()</code> if True, <code>torch_int64()</code> otherwise. Default value is FALSE, i.e. default output data type is <code>torch_int64()</code></td>
</tr>
<tr>
<td><code>right</code></td>
<td>(bool, optional) – if False, return the first suitable location that is found. If True, return the last such index. If no suitable index found, return 0 for non-numerical value (eg. <code>nan</code>, <code>inf</code>) or the size of boundaries (one pass the last index). In other words, if False, gets the lower bound index for each value in input from boundaries. If True, gets the upper bound index instead. Default value is False.</td>
</tr>
<tr>
<td><code>side</code></td>
<td>the same as <code>right</code> but preferred. “left” corresponds to <code>FALSE</code> for <code>right</code> and “right” corresponds to <code>TRUE</code> for <code>right</code>. It will error if this is set to “left” while <code>right</code> is <code>TRUE</code>.</td>
</tr>
<tr>
<td><code>sorter</code></td>
<td>if provided, a tensor matching the shape of the unsorted sorted_sequence containing a sequence of indices that sort it in the ascending order on the innermost dimension.</td>
</tr>
</tbody>
</table>

`torch_searchsorted(sorted_sequence, values, *, out_int32=FALSE, right=FALSE, out=None) -> Tensor`

Find the indices from the innermost dimension of `sorted_sequence` such that, if the corresponding values in `values` were inserted before the indices, the order of the corresponding innermost dimension within `sorted_sequence` would be preserved. Return a new tensor with the same size as `values`. If `right` is `FALSE` (default), then the left boundary of `sorted_sequence` is closed.
torch_selu

Examples

if (torch_is_installed()) {

  sorted_sequence <- torch_tensor(rbind(c(1, 3, 5, 7, 9), c(2, 4, 6, 8, 10)))
  sorted_sequence

  values <- torch_tensor(rbind(c(3, 6, 9), c(3, 6, 9)))
  values

  torch_searchsorted(sorted_sequence, values)
  torch_searchsorted(sorted_sequence, values, right=TRUE)

  sorted_sequence_1d <- torch_tensor(c(1, 3, 5, 7, 9))
  sorted_sequence_1d

  torch_searchsorted(sorted_sequence_1d, values)
}

torch_selu | Selu

Description

Selu

Usage

torch_selu(self)

Arguments

self | the input tensor

selu(input) -> Tensor

Computes the selu transformation.

torch_selu_ | Selu_

Description

Selu_

Usage

torch_selu_(self)

Arguments

self | the input tensor
selu_(input) -> Tensor

In-place version of torch_selu().

torch_serialize

Serialize a torch object returning a raw object

Description

It's just a wrapper around torch_save().

Usage

torch_serialize(obj, ...)

Arguments

<table>
<thead>
<tr>
<th>obj</th>
<th>the saved object</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Additional arguments passed to torch_save(). obj and path are not accepted as they are set by torch_serialize().</td>
</tr>
</tbody>
</table>

Value

A raw vector containing the serialized object. Can be reloaded using torch_load().

See Also

Other torch_save: torch_load(), torch_save()

torch_set_default_dtype

Gets and sets the default floating point dtype.

Description

Gets and sets the default floating point dtype.

Usage

torch_set_default_dtype(d)
torch_set_default_dtype()  

Arguments

| d            | The default floating point dtype to set. Initially set to torch_float(). |
torch_sgn

Description
Sgn

Usage
torch_sgn(self)

Arguments
self (Tensor) the input tensor.

sgn(input, *, out=None) -> Tensor
For complex tensors, this function returns a new tensor whose elements have the same angle as that of the elements of input and absolute value 1. For a non-complex tensor, this function returns the signs of the elements of input (see torch_sign).

out_i = 0, if |input_i| == 0 out_i = \frac{input_i}{|input_i|}, otherwise

Examples
if (torch_is_installed()) {
  if (FALSE) {
    x <- torch_tensor(c(3+4i, 7-24i, 0, 1+2i))
    x$sgn()
    torch_sgn(x)
  }
}

torch_sigmoid

Description
Sigmoid

Usage
torch_sigmoid(self)

Arguments
self (Tensor) the input tensor.
sigmoid(input, out=NULL) -> Tensor

Returns a new tensor with the sigmoid of the elements of input.

\[
out_i = \frac{1}{1 + e^{-input_i}}
\]

Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_sigmoid(a)
}
```

torch_sign

<table>
<thead>
<tr>
<th>torch_sign</th>
<th>Sign</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

Sign

Usage

torch_sign(self)

Arguments

self (Tensor) the input tensor.

sign(input, out=NULL) -> Tensor

Returns a new tensor with the signs of the elements of input.

\[
out_i = \text{sgn}(input_i)
\]

Examples

```r
if (torch_is_installed()) {
  a = torch_tensor(c(0.7, -1.2, 0., 2.3))
  a
  torch_sign(a)
}
```
torch_signbit

### Description
Signbit

### Usage
torch_signbit(self)

### Arguments
- **self** (Tensor) the input tensor.

#### signbit(input, *, out=None) -> Tensor
Tests if each element of input has its sign bit set (is less than zero) or not.

#### Examples
```c
if (torch_is_installed()) {
    a <- torch_tensor(c(0.7, -1.2, 0., 2.3))
    torch_signbit(a)
}
```

---

torch_sin

### Description
Sin

### Usage
torch_sin(self)

### Arguments
- **self** (Tensor) the input tensor.

#### sin(input, out=NULL) -> Tensor
Returns a new tensor with the sine of the elements of input.

\[
out_i = \sin(input_i)
\]
torch_sinh

Description

Sinh

Usage

torch_sinh(self)

Arguments

self (Tensor) the input tensor.

sinh(input, out=NULL) -> Tensor

Returns a new tensor with the hyperbolic sine of the elements of input.

\[
\text{out}_i = \sinh(\text{input}_i)
\]

Examples

if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_sin(a)
    }


if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_sin(a)
    }

\[
\text{out}_i = \sinh(\text{input}_i)
\]
torch_slogdet  Slogdet

Description
Slogdet

Usage
torch_slogdet(self)

Arguments
self (Tensor) the input tensor of size (*, n, n) where * is zero or more batch dimensions.

slogdet(input) -> (Tensor, Tensor)
Calculates the sign and log absolute value of the determinant(s) of a square matrix or batches of square matrices.

Note
If `input` has zero determinant, this returns `(0, -inf)`.

Backward through `slogdet` internally uses SVD results when `input` is not invertible. In this case, double backward through `slogdet` will be unstable in when `input` doesn't have distinct singular values. See `~torch.svd` for details.

Examples
if (torch_is_installed()) {
A = torch.randn(c(3, 3))
A
torch.det(A)
torch.logdet(A)
torch_slogdet(A)
}
torch_sort  

Sort

Arguments

self  (Tensor) the input tensor.
dim  (int, optional) the dimension to sort along
descending  (bool, optional) controls the sorting order (ascending or descending)
stable  (bool, optional) – makes the sorting routine stable, which guarantees that the order of equivalent elements is preserved.

sort(input, dim=-1, descending=False) -> (Tensor, LongTensor)

Sorts the elements of the input tensor along a given dimension in ascending order by value.

If dim is not given, the last dimension of the input is chosen.

If descending is TRUE then the elements are sorted in descending order by value.

A namedtuple of (values, indices) is returned, where the values are the sorted values and indices are the indices of the elements in the original input tensor.

Examples

if (torch_is_installed()) {

x = torch.randn(c(3, 4))
out = torch_sort(x)
out
out = torch_sort(x, 1)
out
}

torch_sparse_coo_tensor

Sparse_coo_tensor

Description

Sparse_coo_tensor
torch_sparse_coo_tensor

Usage

```
torch_sparse_coo_tensor(
    indices,
    values,
    size = NULL,
    dtype = NULL,
    device = NULL,
    requires_grad = FALSE
)
```

Arguments

- **indices** (array_like) Initial data for the tensor. Can be a list, tuple, NumPy ndarray, scalar, and other types. Will be cast to a torch.LongTensor internally. The indices are the coordinates of the non-zero values in the matrix, and thus should be two-dimensional where the first dimension is the number of tensor dimensions and the second dimension is the number of non-zero values.

- **values** (array_like) Initial values for the tensor. Can be a list, tuple, NumPy ndarray, scalar, and other types.

- **size** (list, tuple, or torch.Size, optional) Size of the sparse tensor. If not provided the size will be inferred as the minimum size big enough to hold all non-zero elements.

- **dtype** (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, infers data type from values.

- **device** (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

- **requires_grad** (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

```
sparse_coo_tensor(indices, values, size=NULL, dtype=NULL, device=NULL, requires_grad=False) -> Tensor
```

Constructs a sparse tensors in COO(rdinate) format with non-zero elements at the given indices with the given values. A sparse tensor can be uncoalesced, in that case, there are duplicate coordinates in the indices, and the value at that index is the sum of all duplicate value entries:

```
torch_sparse_coo_tensor(indices, values, size=NULL, dtype=NULL, device=NULL, requires_grad=False)
```

Examples

```
if (torch_is_installed()) {

    i = torch_tensor(matrix(c(1, 2, 2, 3, 1, 3), ncol = 3, byrow = TRUE), dtype=torch_int64())
    v = torch_tensor(c(3, 4, 5), dtype=torch_float32())
    torch_sparse_coo_tensor(i, v)
    torch_sparse_coo_tensor(i, v, c(2, 4))
}
```
# create empty sparse tensors
S = torch_sparse_coo_tensor(
    torch_empty(c(1, 0), dtype = torch_int64()),
    torch_tensor(numeric(), dtype = torch_float32()),
    c(1)
)
S = torch_sparse_coo_tensor(
    torch_empty(c(1, 0), dtype = torch_int64()),
    torch_empty(c(0, 2)),
    c(1, 2)
)

---

**torch_split**

*Split*

**Description**

Splits the tensor into chunks. Each chunk is a view of the original tensor.

**Usage**

`torch_split(self, split_size, dim = 1L)`

**Arguments**

- `self` *(Tensor)* tensor to split.
- `split_size` *(int)* size of a single chunk or list of sizes for each chunk
- `dim` *(int)* dimension along which to split the tensor.

**Details**

If `split_size` is an integer type, then tensor will be split into equally sized chunks (if possible). Last chunk will be smaller if the tensor size along the given dimension `dim` is not divisible by `split_size`.

If `split_size` is a list, then tensor will be split into `length(split_size)` chunks with sizes in `dim` according to `split_size_or_sections`. 
torch_sqrt

Description
Sqrt

Usage
torch_sqrt(self)

Arguments
self (Tensor) the input tensor.

sqrt(input, out=NULL) -> Tensor

Returns a new tensor with the square-root of the elements of input.

\[ \text{out}_i = \sqrt{\text{input}_i} \]

Examples
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_sqrt(a)
} 

torch_square

Description
Square

Usage
torch_square(self)

Arguments
self (Tensor) the input tensor.
square(input, out=NULL) -> Tensor

Returns a new tensor with the square of the elements of input.

Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_square(a)
}
```

torch_squeeze

Squeeze

Description

Squeeze

Usage

torch_squeeze(self, dim)

Arguments

self (Tensor) the input tensor.

dim (int, optional) if given, the input will be squeezed only in this dimension

squeeze(input, dim=NULL, out=NULL) -> Tensor

Returns a tensor with all the dimensions of input of size 1 removed.

For example, if input is of shape: \(A \times 1 \times B \times C \times 1 \times D\) then the out tensor will be of shape: \(A \times B \times C \times D\).

When dim is given, a squeeze operation is done only in the given dimension. If input is of shape: \(A \times 1 \times B\), squeeze(input, 0) leaves the tensor unchanged, but squeeze(input, 1) will squeeze the tensor to the shape \(A \times B\).

Note

The returned tensor shares the storage with the input tensor, so changing the contents of one will change the contents of the other.
Examples

```r
if (torch_is_installed()) {
  x = torch_zeros(c(2, 1, 2, 1, 2))
  x
  y = torch_squeeze(x)
  y
  y = torch_squeeze(x, 1)
  y
  y = torch_squeeze(x, 2)
  y
}
```

torch_stack  

Stack

Description

Stack

Usage

torch_stack(tensors, dim = 1L)

Arguments

tensors  
(sequence of Tensors) sequence of tensors to concatenate

dim  
(int) dimension to insert. Has to be between 0 and the number of dimensions of
concatenated tensors (inclusive)

stack(tensors, dim=0, out=NULL) -> Tensor

Concatenates sequence of tensors along a new dimension.

All tensors need to be of the same size.

torch_std  

Std

Description

Std

Usage

torch_std(self, dim, unbiased = TRUE, keepdim = FALSE)
Arguments

- **self**: (Tensor) the input tensor.
- **dim**: (int or tuple of ints) the dimension or dimensions to reduce.
- **unbiased**: (bool) whether to use the unbiased estimation or not
- **keepdim**: (bool) whether the output tensor has `dim` retained or not.

**std(input, unbiased=TRUE) -> Tensor**

Returns the standard-deviation of all elements in the input tensor.

If `unbiased` is `FALSE`, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel’s correction will be used.

**std(input, dim, unbiased=TRUE, keepdim=False, out=NULL) -> Tensor**

Returns the standard-deviation of each row of the input tensor in the dimension `dim`. If `dim` is a list of dimensions, reduce over all of them.

If `keepdim` is `TRUE`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).

If `unbiased` is `FALSE`, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel’s correction will be used.

Examples

```r
if (torch_is_installed()) {

  a = torch_randn(c(1, 3))
  a
  torch_std(a)

  a = torch_randn(c(4, 4))
  a
  torch_std(a, dim=1)
}
```

**Description**

- **Std_mean**

**Usage**

`torch_std_mean(self, dim, unbiased = TRUE, keepdim = FALSE)`
torch_stft

Arguments

- **self** (Tensor) the input tensor.
- **dim** (int or tuple of ints) the dimension or dimensions to reduce.
- **unbiased** (bool) whether to use the unbiased estimation or not
- **keepdim** (bool) whether the output tensor has dim retained or not.

**std_mean(input, unbiased=TRUE) -> (Tensor, Tensor)**

Returns the standard-deviation and mean of all elements in the input tensor.

If unbiased is FALSE, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel’s correction will be used.

**std_mean(input, dim, unbiased=TRUE, keepdim=False) -> (Tensor, Tensor)**

Returns the standard-deviation and mean of each row of the input tensor in the dimension dim. If dim is a list of dimensions, reduce over all of them.

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

If unbiased is FALSE, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel’s correction will be used.

Examples

```python
if (torch_is_installed()) {
    a = torch.randn(c(1, 3))
a
    torch_std_mean(a)

    a = torch.randn(c(4, 4))
a
    torch_std_mean(a, 1)
}
```

---

**torch_stft**  

**Stft**

**Description**

Stft
Usage

```
torch_stft(
    input,  
    n_fft,  
    hop_length = NULL,  
    win_length = NULL,  
    window = NULL,  
    center = TRUE,  
    pad_mode = "reflect",  
    normalized = FALSE,  
    onesided = NULL,  
    return_complex = NULL
)
```

Arguments

- **input** (Tensor) the input tensor
- **n_fft** (int) size of Fourier transform
- **hop_length** (int, optional) the distance between neighboring sliding window frames. Default: NULL (treated as equal to floor(n_fft / 4))
- **win_length** (int, optional) the size of window frame and STFT filter. Default: NULL (treated as equal to n_fft)
- **window** (Tensor, optional) the optional window function. Default: NULL (treated as window of all 1s)
- **center** (bool, optional) whether to pad input on both sides so that the t-th frame is centered at time t × hop_length. Default: TRUE
- **pad_mode** (string, optional) controls the padding method used when center is TRUE. Default: "reflect"
- **normalized** (bool, optional) controls whether to return the normalized STFT results. Default: FALSE
- **onesided** (bool, optional) controls whether to return half of results to avoid redundancy. Default: TRUE
- **return_complex** (bool, optional) controls whether to return complex tensors or not.

Short-time Fourier transform (STFT).

Ignoring the optional batch dimension, this method computes the following expression:

\[ X[m, \omega] = \sum_{k=0}^{\text{win_length}-1} \text{window}[k] \text{input}[m \times \text{hop_length} + k] \exp \left( -j \frac{2\pi \cdot \omega k}{\text{win_length}} \right), \]
where $m$ is the index of the sliding window, and $\omega$ is the frequency that $0 \leq \omega < n_{\text{fft}}$. When \texttt{onesided} is the default value \texttt{TRUE},

* \texttt{input} must be either a 1-D time sequence or a 2-D batch of time sequences.

* If \texttt{hop_length} is \texttt{NULL} (default), it is treated as equal to \texttt{\text{floor}(n_{\text{fft}} / 4)}.

* If \texttt{win_length} is \texttt{NULL} (default), it is treated as equal to \texttt{n_{\text{fft}}}.

* \texttt{window} can be a 1-D tensor of size \texttt{win_length}, e.g., from \texttt{torch_hann_window}. If \texttt{window} is \texttt{NULL} (default), it is treated as if having \texttt{1} everywhere in the window. If \texttt{\text{win_length} < \text{n_fft}}, \texttt{window} will be padded on both sides to length \texttt{n_{\text{fft}}} before being applied.

* If \texttt{center} is \texttt{TRUE} (default), \texttt{input} will be padded on both sides so that the \texttt{t}-th frame is centered at time \texttt{t \times \text{hop_length}}. Otherwise, the \texttt{t}-th frame begins at time \texttt{t \times \text{hop_length}}.

* \texttt{pad_mode} determines the padding method used on \texttt{input} when \texttt{center} is \texttt{TRUE}. See \texttt{torch.nn.functional.pad} for all available options. Default is \texttt{"reflect"}.

* If \texttt{onesided} is \texttt{TRUE} (default), only values for \texttt{\omega} in \texttt{\left[0, 1, 2, \ldots, \left\lfloor \frac{n_{\text{fft}}}{2} \right\rfloor + 1\right]} are returned because the real-to-complex Fourier transform satisfies the conjugate symmetry, i.e., \texttt{X[m, \omega] = X[m, n_{\text{fft}} - \omega]^\ast}.

* If \texttt{normalized} is \texttt{TRUE} (default is \texttt{FALSE}), the function returns the normalized STFT results, i.e., multiplied by \texttt{\text{(frame_length)}^{-0.5}}}.

Returns the real and the imaginary parts together as one tensor of size \texttt{(* \times N \times T \times 2)}, where \texttt{*} is the optional batch size of \texttt{input}, \texttt{N} is the number of frequencies where STFT is applied, \texttt{T} is the total number of frames used, and each pair in the last dimension represents a complex number as the real part and the imaginary part.

\textbf{Warning}

This function changed signature at version 0.4.1. Calling with the previous signature may cause error or return incorrect result.
torch_sub

Sub

Description

Sub

Usage

torch_sub(self, other, alpha = 1L)

Arguments

self (Tensor) the input tensor.
other (Tensor or Scalar) the tensor or scalar to subtract from input
alpha the scalar multiplier for other

\[
\text{sub}(\text{input}, \text{other}, *, \alpha=1, \text{out}=\text{None}) \rightarrow \text{Tensor}
\]

Subtracts \text{other}, scaled by \alpha, from \text{input}.

\[
\text{out}_i = \text{input}_i - \alpha \times \text{other}_i
\]

Supports broadcasting to a common shape, type promotion, and integer, float, and complex inputs.

Examples

```python
if (torch_is_installed()) {
    a <- torch_tensor(c(1, 2))
    b <- torch_tensor(c(0, 1))
    torch_sub(a, b, alpha=2)
}
```

torch_subtract

Subtract

Description

Subtract

Usage

torch_subtract(self, other, alpha = 1L)
torch_sum

Arguments

- **self** (Tensor) the input tensor.
- **other** (Tensor or Scalar) the tensor or scalar to subtract from input
- **alpha** the scalar multiplier for other

**subtract(input, other, *, alpha=1, out=None) -> Tensor**

Alias for `torch_sub()`.

torch_sum

<table>
<thead>
<tr>
<th>Sum</th>
</tr>
</thead>
</table>

Description

Sum

Usage

`torch_sum(self, dim, keepdim = FALSE, dtype = NULL)`

Arguments

- **self** (Tensor) the input tensor.
- **dim** (int or tuple of ints) the dimension or dimensions to reduce.
- **keepdim** (bool) whether the output tensor has `dim` retained or not.
- **dtype** (torch.dtype, optional) the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: NULL.

**sum(input, dtype=NULL) -> Tensor**

Returns the sum of all elements in the input tensor.

**sum(input, dim, keepdim=False, dtype=NULL) -> Tensor**

Returns the sum of each row of the input tensor in the given dimension `dim`. If `dim` is a list of dimensions, reduce over all of them.

If `keepdim` is `TRUE`, the output tensor is of the same size as input except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).
torch_svd

Examples
if (torch_is_installed()) {
    a = torch_randn(c(1, 3))
    a
    torch_sum(a)

    a <- torch_randn(c(4, 4))
    a
    torch_sum(a, 1)
    b <- torch_arange(1, 4 * 5 * 6)$view(c(4, 5, 6))
    torch_sum(b, list(2, 1))
}

torch_svd

Svd

Description
Svd

Usage
torch_svd(self, some = TRUE, compute_uv = TRUE)

Arguments
self (Tensor) the input tensor of size (*, m, n) where * is zero or more batch dimensions consisting of m \times n matrices.
some (bool, optional) controls the shape of returned U and V
compute_uv (bool, optional) option whether to compute U and V or not

svd(input, some=TRUE, compute_uv=TRUE) -> (Tensor, Tensor, Tensor)

This function returns a namedtuple \((U, S, V)\) which is the singular value decomposition of a input real matrix or batches of real matrices \(input\) such that \(input = U \times diag(S) \times V^T\).

If some is TRUE (default), the method returns the reduced singular value decomposition i.e., if the last two dimensions of input are \(m\) and \(n\), then the returned \(U\) and \(V\) matrices will contain only \(\min(n, m)\) orthonormal columns.

If compute_uv is FALSE, the returned \(U\) and \(V\) matrices will be zero matrices of shape \((m \times m)\) and \((n \times n)\) respectively. some will be ignored here.
Note

The singular values are returned in descending order. If input is a batch of matrices, then the singular values of each matrix in the batch is returned in descending order.

The implementation of SVD on CPU uses the LAPACK routine ?gesdd (a divide-and-conquer algorithm) instead of ?gesvd for speed. Analogously, the SVD on GPU uses the MAGMA routine gesdd as well.

Irrespective of the original strides, the returned matrix U will be transposed, i.e. with strides U.contiguous().transpose(-2, -1).stride()

Extra care needs to be taken when backward through U and V outputs. Such operation is really only stable when input is full rank with all distinct singular values. Otherwise, NaN can appear as the gradients are not properly defined. Also, notice that double backward will usually do an additional backward through U and V even if the original backward is only on S.

When some = FALSE, the gradients on U[...,:, min(m, n):] and V[...,:, min(m, n):] will be ignored in backward as those vectors can be arbitrary bases of the subspaces.

When compute_uv = FALSE, backward cannot be performed since U and V from the forward pass is required for the backward operation.

Examples

```python
if (torch_is_installed()) {
    a = torch_randn(c(5, 3))
    a
    out = torch_svd(a)
    u = out[[1]]
    s = out[[2]]
    v = out[[3]]
    torch_dist(a, torch_mm(torch_mm(u, torch_diag(s)), v$transposed()))
    a_big = torch_randn(c(7, 5, 3))
    out = torch_svd(a_big)
    u = out[[1]]
    s = out[[2]]
    v = out[[3]]
    torch_dist(a_big, torch_matmul(torch_matmul(u, torch_diag_embed(s)), v$transposed(-2, -1))))
}
```
Arguments
    self (Tensor) the input tensor.

t(input) -> Tensor

    Expects input to be <= 2-D tensor and transposes dimensions 0 and 1.
    0-D and 1-D tensors are returned as is. When input is a 2-D tensor this is equivalent to transpose(input, 0, 1).

Examples
    if (torch_is_installed()) {
        x = torch_randn(c(2,3))
        x
torch_t(x)
        x = torch_randn(c(3))
        x
torch_t(x)
        x = torch_randn(c(2,3))
        x
torch_t(x)
    }

torch_take

Description
    Take

Usage
    torch_take(self, index)

Arguments
    self (Tensor) the input tensor.
    index (LongTensor) the indices into tensor

take(input, index) -> Tensor

    Returns a new tensor with the elements of input at the given indices. The input tensor is treated as if it were viewed as a 1-D tensor. The result takes the same shape as the indices.
torch_tan

Examples

```r
if (torch_is_installed()) {
    src = torch_tensor(matrix(c(4,3,5,6,7,8), ncol = 3, byrow = TRUE))
    torch_take(src, torch_tensor(c(1, 2, 5), dtype = torch_int64()))
}
```

Description

Tan

Usage

`torch_tan(self)`

Arguments

- `self` (Tensor) the input tensor.

```
tan(input, out=NULL) -> Tensor
```

Returns a new tensor with the tangent of the elements of `input`. 

```
outᵢ = tan(inputᵢ)
```

Examples

```r
if (torch_is_installed()) {
    a = torch_randn(c(4))
    a
    torch_tan(a)
}
```
## torch_tanh

**Tanh**

### Description

Tanh

### Usage

```r
torch_tanh(self)
```

### Arguments

- **self** (Tensor): the input tensor.

### tanh(input, out=NULL) -> Tensor

Returns a new tensor with the hyperbolic tangent of the elements of `input`.

```r
out_i = tanh(input_i)
```

### Examples

```r
if (torch_is_installed()) {

  a = torch_randn(c(4))

  a
  torch_tanh(a)
}
```

## torch_tensor

**Converts R objects to a torch tensor**

### Description

Converts R objects to a torch tensor

### Usage

```r
torch_tensor(
  data,
  dtype = NULL,
  device = NULL,
  requires_grad = FALSE,
  pin_memory = FALSE
)
```
torch_tensordot

Arguments

- data: an R atomic vector, matrix or array
- dtype: a torch_dtype instance
- device: a device created with torch_device()
- requires_grad: if autograd should record operations on the returned tensor.
- pin_memory: If set, returned tensor would be allocated in the pinned memory.

Examples

```r
if (torch_is_installed()) {
  torch_tensor(c(1, 2, 3, 4))
  torch_tensor(c(1, 2, 3, 4), dtype = torch_int())
}
```

Description

Returns a contraction of a and b over multiple dimensions. tensordot implements a generalized matrix product.

Usage

torch_tensordot(a, b, dims = 2)

Arguments

- a: (Tensor) Left tensor to contract
- b: (Tensor) Right tensor to contract
- dims: (int or tuple of two lists of integers) number of dimensions to contract or explicit lists of dimensions for a and b respectively

Examples

```r
if (torch_is_installed()) {
  a <- torch_arange(start = 1, end = 60)$reshape(c(3, 4, 5))
  b <- torch_arange(start = 1, end = 24)$reshape(c(4, 3, 2))
  torch_tensordot(a, b, dims = list(c(2, 1), c(1, 2)))
  ## Not run:
  a = torch_randn(3, 4, 5, device='cuda')
  b = torch_randn(4, 5, 6, device='cuda')
  c = torch_tensordot(a, b, dims=2)$cpu()
  ## End(Not run)
}
```
torch_tensor_from_buffer

*Creates a tensor from a buffer of memory*

**Description**

It creates a tensor without taking ownership of the memory it points to. You must call clone if you want to copy the memory over a new tensor.

**Usage**

```python
torch_tensor_from_buffer(buffer, shape, dtype = "float")
```

```python
buffer_from_torch_tensor(tensor)
```

**Arguments**

- **buffer**: An R atomic object containing the data in a contiguous array.
- **shape**: The shape of the resulting tensor.
- **dtype**: A torch data type for the resulting tensor.
- **tensor**: Tensor object that will be converted into a buffer.

**Functions**

- `buffer_from_torch_tensor()`: Creates a raw vector containing the tensor data. Causes a data copy.

---

torch_threshold_

*Threshold_*

**Description**

Threshold_

**Usage**

```python
torch_threshold_(self, threshold, value)
```

**Arguments**

- **self**: input tensor
- **threshold**: The value to threshold at
- **value**: The value to replace with

```python
threshold_(input, threshold, value) -> Tensor
```

In-place version of torch_threshold.
torch_topk

Description

Topk

Usage

torch_topk(self, k, dim = -1L, largest = TRUE, sorted = TRUE)

Arguments

self (Tensor) the input tensor.
k (int) the k in "top-k"
dim (int, optional) the dimension to sort along
largest (bool, optional) controls whether to return largest or smallest elements
sorted (bool, optional) controls whether to return the elements in sorted order

topk(input, k, dim=NULL, largest=TRUE, sorted=TRUE) -> (Tensor, LongTensor)

Returns the k largest elements of the given input tensor along a given dimension.

If dim is not given, the last dimension of the input is chosen.

If largest is FALSE then the k smallest elements are returned.

A namedtuple of (values, indices) is returned, where the indices are the indices of the elements in the original input tensor.

The boolean option sorted if TRUE, will make sure that the returned k elements are themselves sorted

Examples

if (torch_is_installed()) {

    x = torch_arange(1., 6.)
    x
    torch_topk(x, 3)
}


torch_trace

Description
Trace

Usage
torch_trace(self)

Arguments
self the input tensor

trace(input) -> Tensor
Returns the sum of the elements of the diagonal of the input 2-D matrix.

Examples
if (torch_is_installed()) {
    x <- torch_arange(1, 9)$view(c(3, 3))
x
torch_trace(x)
}

torch_transpose

Description
Transpose

Usage
torch_transpose(self, dim0, dim1)

Arguments
self (Tensor) the input tensor.
dim0 (int) the first dimension to be transposed
dim1 (int) the second dimension to be transposed
**torch_trapz**

**torch_transpose(input, dim0, dim1) -> Tensor**

Returns a tensor that is a transposed version of input. The given dimensions `dim0` and `dim1` are swapped.

The resulting output tensor shares its underlying storage with the input tensor, so changing the content of one would change the content of the other.

**Examples**

```python
if (torch_is_installed()) {
    x = torch_randn(c(2, 3))
    x
    torch_transpose(x, 1, 2)
}
```

---

**torch_trapz**

**Description**

Trapz

**Usage**

torch_trapz(y, dx = 1L, x, dim = -1L)

**Arguments**

- **y** (Tensor) The values of the function to integrate
- **dx** (float) The distance between points at which `y` is sampled.
- **x** (Tensor) The points at which the function `y` is sampled. If `x` is not in ascending order, intervals on which it is decreasing contribute negatively to the estimated integral (i.e., the convention \( \int_b^a f = - \int_a^b f \) is followed).
- **dim** (int) The dimension along which to integrate. By default, use the last dimension.

**trapz(y, x, *, dim=-1) -> Tensor**

Estimate \( \int y \, dx \) along \( \text{dim} \), using the trapezoid rule.

**trapz(y, *, dx=1, dim=-1) -> Tensor**

As above, but the sample points are spaced uniformly at a distance of `dx`. 
torch_triangular_solve

Examples

```r
if (torch_is_installed()) {
    y = torch_randn(list(2, 3))
    y
    x = torch_tensor(matrix(c(1, 3, 4, 1, 2, 3), ncol = 3, byrow=TRUE))
    torch_trapz(y, x = x)
}
```

Description

Triangular_solve

Usage

```r
torch_triangular_solve(
    self, A, upper = TRUE, transpose = FALSE, unitriangular = FALSE
)
```

Arguments

- **self**: (Tensor) multiple right-hand sides of size 
  \((*, m, k)\) where \(*\) is zero of more batch 
  dimensions \((b)\)
- **A**: (Tensor) the input triangular coefficient matrix of size 
  \((*, m, m)\) where \(*\) is zero 
  or more batch dimensions
- **upper**: (bool, optional) whether to solve the upper-triangular system of 
  equations (default) or the lower-triangular system of equations. Default: TRUE.
- **transpose**: (bool, optional) whether \(A\) should be transposed before being 
  sent into the solver. Default: FALSE.
- **unitriangular**: (bool, optional) whether \(A\) is unit triangular. If TRUE, the diagonal 
  elements of \(A\) are assumed to be 1 and not referenced from \(A\). Default: FALSE.
triangular_solve(input, A, upper=TRUE, transpose=False, unitriangular=False) -> (Tensor, Tensor)

Solves a system of equations with a triangular coefficient matrix A and multiple right-hand sides b.
In particular, solves \(AX = b\) and assumes A is upper-triangular with the default keyword arguments.

\[\text{torch\_triangular\_solve}(b, A)\]
can take in 2D inputs \(b, A\) or inputs that are batches of 2D matrices. If the inputs are batches, then returns batched outputs \(X\).

**Examples**

```python
if (torch_is_installed()) {
    A = torch_randn(c(2, 2))$triu()
    A
    b = torch_randn(c(2, 3))
    b
    torch_triangular_solve(b, A)
}
```

---

**torch_tril**

**Tril**

**Description**
Tril

**Usage**

\[\text{torch\_tril}(\text{self}, \text{diagonal} = 0L)\]

**Arguments**

- **self** (Tensor) the input tensor.
- **diagonal** (int, optional) the diagonal to consider

**tril(input, diagonal=0, out=NULL) -> Tensor**

Returns the lower triangular part of the matrix (2-D tensor) or batch of matrices input, the other elements of the result tensor out are set to 0.

The lower triangular part of the matrix is defined as the elements on and below the diagonal.

The argument diagonal controls which diagonal to consider. If diagonal = 0, all elements on and below the main diagonal are retained. A positive value includes just as many diagonals above the main diagonal, and similarly a negative value excludes just as many diagonals below the main diagonal. The main diagonal are the set of indices \{(i, i)\} for \(i \in [0, \min\{d_1, d_2\} - 1]\) where \(d_1, d_2\) are the dimensions of the matrix.
torch_tril_indices

**Examples**

```r
if (torch_is_installed()) {
    a = torch_randn(c(3, 3))
    torch_tril(a)
    b = torch_randn(c(4, 6))
    torch_tril(b, diagonal=1)
    torch_tril(b, diagonal=-1)
}
```

**torch_tril_indices**

**Tril_indices**

**Description**

Tril_indices

**Usage**

```r
torch_tril_indices(
    row,
    col,
    offset = 0,
    dtype = NULL,
    device = NULL,
    layout = NULL
)
```

**Arguments**

- `row` (int) number of rows in the 2-D matrix.
- `col` (int) number of columns in the 2-D matrix.
- `offset` (int) diagonal offset from the main diagonal. Default: if not provided, 0.
- `dtype` (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, torch_long.
- `device` (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- `layout` (torch.layout, optional) currently only support torch_strided.
torch_triu

tril_indices(row, col, offset=0, dtype=torch.long, device='cpu', layout=torch.strided) -> Tensor

Returns the indices of the lower triangular part of a row-by-col matrix in a 2-by-N Tensor, where the first row contains row coordinates of all indices and the second row contains column coordinates. Indices are ordered based on rows and then columns.

The lower triangular part of the matrix is defined as the elements on and below the diagonal.

The argument offset controls which diagonal to consider. If offset = 0, all elements on and below the main diagonal are retained. A positive value includes just as many diagonals above the main diagonal, and similarly a negative value excludes just as many diagonals below the main diagonal.

The main diagonal are the set of indices \{ (i, i) \} for \( i \in [0, \min\{d_1, d_2\} - 1] \) where \( d_1, d_2 \) are the dimensions of the matrix.

Note

When running on CUDA, `row * col` must be less than \( 2^{59} \) to prevent overflow during calculation.

Examples

```python
if (torch_is_installed()) {
    ## Not run:
    a = torch_tril_indices(3, 3)
    a
    a = torch_tril_indices(4, 3, -1)
    a
    a = torch_tril_indices(4, 3, 1)
    a

    ## End(Not run)
}
```

---

torch_triu | Triu

**Description**

Triu

**Usage**

torch_triu(self, diagonal = 0L)

**Arguments**

- self (Tensor) the input tensor.
- diagonal (int, optional) the diagonal to consider
torch_triu_indices

Usage

```r
torch_triu_indices(
    row,
    col,
    offset = 0,
    dtype = NULL,
    device = NULL,
    layout = NULL
)
```
torch_triu_indices

Arguments

- **row**: (int) number of rows in the 2-D matrix.
- **col**: (int) number of columns in the 2-D matrix.
- **offset**: (int) diagonal offset from the main diagonal. Default: if not provided, 0.
- **dtype**: (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, torch.long.
- **device**: (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **layout**: (torch.layout, optional) currently only support torch_strided.

**triu_indices(row, col, offset=0, dtype=torch.long, device='cpu', layout=torch.strided) -> Tensor**

Returns the indices of the upper triangular part of a row by col matrix in a 2-by-N Tensor, where the first row contains row coordinates of all indices and the second row contains column coordinates. Indices are ordered based on rows and then columns.

The upper triangular part of the matrix is defined as the elements on and above the diagonal.

The argument **offset** controls which diagonal to consider. If **offset** = 0, all elements on and above the main diagonal are retained. A positive value excludes just as many diagonals above the main diagonal, and similarly a negative value includes just as many diagonals below the main diagonal.

The main diagonal are the set of indices \{ (i, i) \} for \( i \in [0, \min(d_1, d_2) - 1] \) where \( d_1, d_2 \) are the dimensions of the matrix.

**Note**

When running on CUDA, `row * col` must be less than \( 2^{59} \) to prevent overflow during calculation.

**Examples**

```python
if (torch_is_installed()) {
    ## Not run:
    a = torch_triu_indices(3, 3)
    a
    a = torch_triu_indices(4, 3, -1)
    a
    a = torch_triu_indices(4, 3, 1)
    a

    ## End(Not run)
}
```
torch_true_divide  TRUE_divide

Description
TRUE_divide

Usage
torch_true_divide(self, other)

Arguments
self (Tensor) the dividend
other (Tensor or Scalar) the divisor

true_divide(dividend, divisor) -> Tensor

Performs "true division" that always computes the division in floating point. Analogous to division in Python 3 and equivalent to torch_div except when both inputs have bool or integer scalar types, in which case they are cast to the default (floating) scalar type before the division.

\[
\text{out}_i = \frac{\text{dividend}_i}{\text{divisor}}
\]

Examples
if (torch_is_installed()) {
    dividend = torch_tensor(c(5, 3), dtype=torch_int())
    divisor = torch_tensor(c(3, 2), dtype=torch_int())
    torch_true_divide(dividend, divisor)
    torch_true_divide(dividend, 2)
}

torch_trunc  Trunc

Description
Trunc

Usage
torch_trunc(self)
Arguments

self (Tensor) the input tensor.

trunc(input, out=NULL) -> Tensor

Returns a new tensor with the truncated integer values of the elements of input.

Examples

```r
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_trunc(a)
}
```

Description

Unbind

Usage

torch_unbind(self, dim = 1L)

Arguments

self (Tensor) the tensor to unbind
dim (int) dimension to remove

unbind(input, dim=0) -> seq

Removes a tensor dimension.

Returns a tuple of all slices along a given dimension, already without it.

Examples

```r
if (torch_is_installed()) {
  torch_unbind(torch_tensor(matrix(1:9, ncol = 3, byrow=TRUE)))
}
```
**torch_unique_consecutive**

*Unique_consecutive*

**Description**

Unique_consecutive

**Usage**

```python
torch_unique_consecutive(
    self,
    return_inverse = FALSE,
    return_counts = FALSE,
    dim = NULL
)
```

**Arguments**

- **self**: (Tensor) the input tensor
- **return_inverse**: (bool) Whether to also return the indices for where elements in the original input ended up in the returned unique list.
- **return_counts**: (bool) Whether to also return the counts for each unique element.
- **dim**: (int) the dimension to apply unique. If NULL, the unique of the flattened input is returned. default: NULL

**TEST**

Eliminates all but the first element from every consecutive group of equivalent elements.

.. note:: This function is different from ['torch_unique'] in the sense that this function only eliminates consecutive duplicate values. This semantics is similar to 'std::unique' in C++.

**Examples**

```python
if (torch_is_installed()) {
    x = torch_tensor(c(1, 1, 2, 2, 3, 1, 1, 2))
    output = torch_unique_consecutive(x)
    output
    torch_unique_consecutive(x, return_inverse=TRUE)
    torch_unique_consecutive(x, return_counts=TRUE)
}
```
**torch_unsafe_chunk**  
*Unsafe_chunk*

### Description

Unsafe_chunk

### Usage

torch_unsafe_chunk(self, chunks, dim = 1L)

### Arguments

- **self** *(Tensor)*: the tensor to split
- **chunks** *(int)*: number of chunks to return
- **dim** *(int)*: dimension along which to split the tensor

**unsafe_chunk(input, chunks, dim=0) -> List of Tensors**

Works like `torch_chunk()` but without enforcing the autograd restrictions on inplace modification of the outputs.

### Warning

This function is safe to use as long as only the input, or only the outputs are modified inplace after calling this function. It is user’s responsibility to ensure that is the case. If both the input and one or more of the outputs are modified inplace, gradients computed by autograd will be silently incorrect.

---

**torch_unsafe_split**  
*Unsafe_split*

### Description

Unsafe_split

### Usage

torch_unsafe_split(self, split_size, dim = 1L)

### Arguments

- **self** *(Tensor)*: tensor to split.
- **split_size** *(int)*: size of a single chunk or list of sizes for each chunk
- **dim** *(int)*: dimension along which to split the tensor.
unsafe_split(tensor, split_size_or_sections, dim=0) -> List of Tensors

Works like torch_split() but without enforcing the autograd restrictions on inplace modification of the outputs.

Warning

This function is safe to use as long as only the input, or only the outputs are modified inplace after calling this function. It is user’s responsibility to ensure that is the case. If both the input and one or more of the outputs are modified inplace, gradients computed by autograd will be silently incorrect.

torch_unsqueeze

Description

Unsqueeze

Usage

torch_unsqueeze(self, dim)

Arguments

self (Tensor) the input tensor.
dim (int) the index at which to insert the singleton dimension

unsqueeze(input, dim) -> Tensor

Returns a new tensor with a dimension of size one inserted at the specified position.
The returned tensor shares the same underlying data with this tensor.
A dim value within the range [-input.dim() - 1, input.dim() + 1) can be used. Negative
dim will correspond to unsqueeze applied at dim = dim + input.dim() + 1.

Examples

if (torch_is_installed()) {
    x = torch_tensor(c(1, 2, 3, 4))
    torch_unsqueeze(x, 1)
    torch_unsqueeze(x, 2)
}
torch_vander

Description
Vander

Usage
torch_vander(x, N = NULL, increasing = FALSE)

Arguments
x (Tensor) 1-D input tensor.
N (int, optional) Number of columns in the output. If N is not specified, a square array is returned ($N = \text{len}(x)$).
increasing (bool, optional) Order of the powers of the columns. If TRUE, the powers increase from left to right, if FALSE (the default) they are reversed.

vander(x, N=None, increasing=False) -> Tensor
Generates a Vandermonde matrix.
The columns of the output matrix are elementwise powers of the input vector $x^{(N-1)}, x^{(N-2)}, \ldots, x^0$. If increasing is TRUE, the order of the columns is reversed $x^0, x^1, \ldots, x^{(N-1)}$. Such a matrix with a geometric progression in each row is named for Alexandre-Theophile Vandermonde.

Examples
if (torch_is_installed()) {
  x <- torch_tensor(c(1, 2, 3, 5))
torch_vander(x)
torch_vander(x, N=3)
torch_vander(x, N=3, increasing=TRUE)
}

torch_var

Description
Var

Usage
torch_var(self, dim, unbiased = TRUE, keepdim = FALSE)
torch_var_mean

Arguments

- **self**: (Tensor) the input tensor.
- **dim**: (int or tuple of ints) the dimension or dimensions to reduce.
- **unbiased**: (bool) whether to use the unbiased estimation or not.
- **keepdim**: (bool) whether the output tensor has dim retained or not.

**var(input, unbiased=TRUE) -> Tensor**

Returns the variance of all elements in the input tensor.

If unbiased is FALSE, then the variance will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

**var(input, dim, keepdim=False, unbiased=TRUE, out=NULL) -> Tensor**

Returns the variance of each row of the input tensor in the given dimension dim.

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

If unbiased is FALSE, then the variance will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

**Examples**

```r
if (torch_is_installed()) {
  a = torch_randn(c(1, 3))
  a
  torch_var(a)

  a = torch_randn(c(4, 4))
  a
  torch_var(a, 1)
}
```

---

**torch_var_mean**  
**Var_mean**

**Description**

Var_mean

**Usage**

```r
torch_var_mean(self, dim, unbiased = TRUE, keepdim = FALSE)
```
torch_vdot

Arguments

- **self** (Tensor) the input tensor.
- **dim** (int or tuple of ints) the dimension or dimensions to reduce.
- **unbiased** (bool) whether to use the unbiased estimation or not
- **keepdim** (bool) whether the output tensor has dim retained or not.

**var_mean**(input, unbiased=TRUE) -> (Tensor, Tensor)

Returns the variance and mean of all elements in the input tensor.

If unbiased is FALSE, then the variance will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

**var_mean**(input, dim, keepdim=False, unbiased=TRUE) -> (Tensor, Tensor)

Returns the variance and mean of each row of the input tensor in the given dimension dim.

If keepdim is TRUE, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see torch_squeeze), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

If unbiased is FALSE, then the variance will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

Examples

```python
if (torch_is_installed()) {
    a = torch_randn(c(1, 3))
    a
    torch_var_mean(a)

    a = torch_randn(c(4, 4))
    a
    torch_var_mean(a, 1)
}
```

---

**torch_vdot**

**Vdot**

Description

Vdot

Usage

torch_vdot(self, other)
torch_view_as_complex

Arguments

self (Tensor) first tensor in the dot product. Its conjugate is used if it’s complex.
other (Tensor) second tensor in the dot product.

vdot(input, other, *, out=None) -> Tensor

Computes the dot product (inner product) of two tensors. The vdot(a, b) function handles complex
numbers differently than dot(a, b). If the first argument is complex, the complex conjugate of the
first argument is used for the calculation of the dot product.

Note

This function does not broadcast.

Examples

if (torch_is_installed()) {
    torch_vdot(torch_tensor(c(2, 3)), torch_tensor(c(2, 1))
if (FALSE) {
    a <- torch_tensor(list(1 +2i, 3 - 1i))
    b <- torch_tensor(list(2 +1i, 4 - 0i))
    torch_vdot(a, b)
    torch_vdot(b, a)
} }

view_as_complex(input) -> Tensor

Returns a view of input as a complex tensor. For an input complex tensor of size m1, m2, ..., mi, 2,
this function returns a new complex tensor of size m1, m2, ..., mi where the last dimension of
the input tensor is expected to represent the real and imaginary components of complex numbers.
**torch_view_as_real**

**Warning**

`torch_view_as_complex` is only supported for tensors with `torch_dtype` `torch_float64()` and `torch_float32()`. The input is expected to have the last dimension of size 2. In addition, the tensor must have a stride of 1 for its last dimension. The strides of all other dimensions must be even numbers.

**Examples**

```r
if (torch_is_installed()) {
  if (FALSE) {
    x <- torch_randn(4, 2)
    x
    torch_view_as_complex(x)
  }
}
```

**Description**

View_as_real

**Usage**

`torch_view_as_real(self)`

**Arguments**

- `self` (Tensor) the input tensor.

`view_as_real(input) -> Tensor`

Returns a view of `input` as a real tensor. For an input complex tensor of size `m1,m2,...,mi`, this function returns a new real tensor of size `m1,m2,...,mi,2`, where the last dimension of size 2 represents the real and imaginary components of complex numbers.

**Warning**

`torch_view_as_real()` is only supported for tensors with complex dtypes.

**Examples**

```r
if (torch_is_installed()) {
  if (FALSE) {
    x <- torch_randn(4, dtype=torch_cfloat())
    x
    torch_view_as_real(x)
  }
}
```
torch_vstack  Vstack

Description
Vstack

Usage
torch_vstack(tensors)

Arguments
tensors (sequence of Tensors) sequence of tensors to concatenate

vstack(tensors, *, out=None) -> Tensor
Stack tensors in sequence vertically (row wise).
This is equivalent to concatenation along the first axis after all 1-D tensors have been reshaped by torch_atleast_2d().

Examples
if (torch_is_installed()) {

  a <- torch_tensor(c(1, 2, 3))
  b <- torch_tensor(c(4, 5, 6))
  torch_vstack(list(a,b))

  a <- torch_tensor(rbind(1,2,3))
  b <- torch_tensor(rbind(4,5,6))
  torch_vstack(list(a,b))
}

torch_where  Where

Description
Where

Usage
torch_where(condition, self = NULL, other = NULL)
torch_zeros

Zeros

Description

Zeros

Arguments

- **condition** (BoolTensor) When TRUE (nonzero), yield x, otherwise yield y
- **self** (Tensor) values selected at indices where condition is TRUE
- **other** (Tensor) values selected at indices where condition is FALSE

where(condition, x, y) -> Tensor

Return a tensor of elements selected from either x or y, depending on condition. The operation is defined as:

\[
\text{out}_i = \begin{cases} 
  x_i & \text{if condition}_i \\
  y_i & \text{otherwise}
\end{cases}
\]

where(condition) -> tuple of LongTensor

torch_where(condition) is identical to torch_nonzero(condition, as_tuple=TRUE).

Note

The tensors `condition`, `x`, `y` must be broadcastable.

See also torch_nonzero().

Examples

```python
if (torch_is_installed()) {

  ## Not run:
  x = torch_randn(c(3, 2))
  y = torch_ones(c(3, 2))
  x
  torch_where(x > 0, x, y)

  ## End(Not run)
}
```

```
Usage

torch_zeros(
    ..., 
    names = NULL, 
    dtype = NULL, 
    layout = NULL, 
    device = NULL, 
    requires_grad = FALSE 
)

Arguments

... a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

names optional dimension names

dtype (torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see torch_set_default_tensor_type).

layout (torch.layout, optional) the desired layout of returned Tensor. Default: torch_strided.

device (torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.

zeros(*size, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires_grad=False) -> Tensor

Returns a tensor filled with the scalar value 0, with the shape defined by the variable argument size.

Examples

    if (torch_is_installed()) {
        torch.zeros(c(2, 3))
        torch.zeros(c(5))
    }

torch_zeros_like Zeros_like

Description

Zeros_like
torch_zeros_like

Usage

torch_zeros_like(
    input,
    dtype = NULL,
    layout = NULL,
    device = NULL,
    requires_grad = FALSE,
    memory_format = torch_preserve_format()
)

Arguments

input (Tensor) the size of input will determine size of the output tensor.
dtype (torch.dtype, optional) the desired data type of returned Tensor. Default: if NULL, defaults to the dtype of input.
layout (torch.layout, optional) the desired layout of returned tensor. Default: if NULL, defaults to the layout of input.
device (torch.device, optional) the desired device of returned tensor. Default: if NULL, defaults to the device of input.
requires_grad (bool, optional) If autograd should record operations on the returned tensor. Default: FALSE.
memory_format (torch.memory_format, optional) the desired memory format of returned Tensor. Default: torch_preserve_format.

zeros_like(input, dtype=NULL, layout=NULL, device=NULL, requires_grad=False, memory_format=torch.preserve_format) -> Tensor

Returns a tensor filled with the scalar value 0, with the same size as input. torch_zeros_like(input) is equivalent to torch_zeros(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Warning

As of 0.4, this function does not support an out keyword. As an alternative, the old torch_zeros_like(input, out=output) is equivalent to torch_zeros(input.size(), out=output).

Examples

if (torch_is_installed()) {

    input = torch_empty(c(2, 3))
    torch_zeros_like(input)
}
with_detect_anomaly

Context-manager that enable anomaly detection for the autograd engine.

Description

This does two things:

Usage

\[
\text{with} \_\text{detect} \_\text{anomaly}(\text{code})
\]

Arguments

code Code that will be executed in the detect anomaly context.

Details

- Running the forward pass with detection enabled will allow the backward pass to print the traceback of the forward operation that created the failing backward function.
- Any backward computation that generate "nan" value will raise an error.

Warning

This mode should be enabled only for debugging as the different tests will slow down your program execution.

Examples

```r
if (torch_is_installed()) {
  x <- torch_randn(2, requires_grad = TRUE)
  y <- torch_randn(1)
  b <- (x*y)$sum()
  y$add_(1)

  try({
    b$backward()

    with_detect_anomaly({
      b$backward()
    })
  })
}
```
**with_enable_grad**

**Enable grad**

---

**Description**

Context-manager that enables gradient calculation. Enables gradient calculation, if it has been disabled via `with_no_grad`.

**Usage**

```r
with_enable_grad(code)

local_enable_grad(.env = parent.frame())
```

**Arguments**

- `code` code to be executed with gradient recording.
- `env` The environment to use for scoping.

**Details**

This context manager is thread local; it will not affect computation in other threads.

**Functions**

- `local_enable_grad()`: Locally enable gradient computations.

**Examples**

```r
if (torch_is_installed()) {

  x <- torch_tensor(1, requires_grad = TRUE)
  with_no_grad({
    with_enable_grad({
      y <- x * 2
    })
  })

  y$backward()
  x$grad
}
```
with_no_grad

Temporarily modify gradient recording.

Description

Temporarily modify gradient recording.

Usage

with_no_grad(code)
local_no_grad(.env = parent.frame())

Arguments

code code to be executed with no gradient recording.
.env The environment to use for scoping.

Functions

- local_no_grad(): Disable autograd until it goes out of scope

Examples

if (torch_is_installed()) {
  x <- torch_tensor(runif(5), requires_grad = TRUE)
  with_no_grad(
    x$sub_(torch_tensor(as.numeric(1:5)))
  )
  x
  x$grad
}
Index

* datasets
  * jit_ops, 55

* distributions
  * distr_bernoulli, 39
  * distr_chi2, 41
  * distr_gamma, 41
  * distr_multivariate_normal, 43
  * distr_normal, 44
  * distr_poisson, 45

* linalg
  * linalg_cholesky, 60
  * linalg_cholesky_ex, 61
  * linalg_det, 64
  * linalg_eig, 64
  * linalg_eigh, 66
  * linalg_eigvals, 68
  * linalg_eigvalsh, 69
  * linalg_householder_product, 70
  * linalg_inv, 71
  * linalg_inv_ex, 72
  * linalg_lstsq, 73
  * linalg_matrix_norm, 75
  * linalg_matrix_power, 77
  * linalg_matrix_rank, 78
  * linalg_multi_dot, 79
  * linalg_norm, 80
  * linalg_pinv, 82
  * linalgqr, 83
  * linalg_slogdet, 84
  * linalg_solve, 85
  * linalg_svd, 86
  * linalg_svdvals, 88
  * linalg_tensorinv, 89
  * linalg_tensorsolve, 90
  * linalg_vector_norm, 91

* serialization
  * load_state_dict, 93
  * torch_load, 447
  * torch_save, 526
  * torch_serialized, 530

* tensor-attributes
  * is_torch_device, 51
  * is_torch_dtype, 51
  * torch_dtype, 389
  * torch_finfo, 406
  * torch_iinfo, 427
  * torch_qscheme, 504
  * torch_reduction, 517
  * torch_set_default_dtype, 530

* tensor-attributes
  * torch_device, 379

* torch_save
  * torch_load, 447
  * torch_save, 526
  * torch_serialized, 530

as_array, 17
autograd_backward, 19
autograd_function, 20
autograd_grad, 21
autograd_set_grad_mode, 22
AutogradContext, 17, 20
backends_cudnn_is_available, 23
backends_cudnn_version, 23
backends_mkl_is_available, 24
backends_mkldnn_is_available, 23
backends_mps_is_available, 24
backends_openmp_is_available, 24
broadcast_all, 25
buffer_from_torch_tensor
  * (torch_tensor_from_buffer), 554
Constraint, 25
contrib_sort_vertices, 26
cuda_amp_grad_scaler, 27
cuda_amp_grad_scaler, 94
cuda_current_device, 28
cuda_current_device, 31
cuda_device_count, 28
cuda_empty_cache, 28
cuda_empty_cache(), 28
cuda_get_device_capability, 29
cuda_get_rng_state
  (torch.get_rng_state), 419
cuda_is_available, 29
cuda_memory_stats, 29
_cuda_memory_summary
  (cuda_memory_stats), 29
cuda_runtime_version, 31
cuda_set_rng_state
  (torch.get_rng_state), 419
cuda_synchronize, 31
dataloader, 31
dataloader(), 34, 35, 309
dataloader_make_iter, 33, 34
dataloader_next, 34
dataset, 34
dataset(), 33, 309, 310
dataset_subset, 35
distr_bernoulli, 39, 41, 42, 44, 45
distr_categorical, 40
distr_chi2, 39, 41, 42, 44, 45
distr_gamma, 39, 41, 41, 44, 45
distr_mixture_same_family, 42
distr_multivariate_normal, 39, 41, 42, 43, 45
distr_normal, 39, 41, 42, 44, 45
distr_poisson, 39, 41, 42, 44, 45
distribution, 36, 39, 41, 42, 44, 45
distribution(), 33, 309, 310
distribution_bernoulli, 39, 41, 42, 44, 45
distribution_categorical, 40
distribution_chi2, 39, 41, 42, 44, 45
distribution_gamma, 39, 41, 41, 44, 45
distribution_mixture_same_family, 42
distribution_multivariate_normal, 39, 41, 42, 43, 45
distribution_normal, 39, 41, 42, 44, 45
distribution_poisson, 39, 41, 42, 44, 45
distribution(), 33, 309, 310
dataloader, 31
enumerate, 46
enumerate(dataloader, 46
get_install_libs_url, 47
install_torch, 48
install_torch_from_file
  (get_install_libs_url), 47
int, 76, 80, 90, 91
is_dataloader, 49
is_nn_buffer, 49
is_nn_module, 50
is_nn_parameter, 50
is_optimizer, 50
is_torch_device, 51
is_torch_dtype, 51
is_torch_layout, 51
is_torch_memory_format, 52
is_torch_qscheme, 52
is_undefined_tensor, 52
iterable_dataset, 53
jit_compile, 54
jit_load, 54
jit_ops, 55
jit_save, 56
jit_save(), 55
jit_save_for_mobile, 56
jit_scalar, 57
jit_trace, 57, 59
jit_trace(), 59
jit_trace_module, 59
jit_trace_module(), 58, 59
jit_tuple, 60
linalg_cholesky(), 61, 62, 67
linalg_cholesky_ex, 61, 61, 64, 65, 67–69, 71–73, 75–77, 79–81, 83–86, 88–92
linalg_cholesky_ex(), 61
linalg_cond, 62
linalg_eig(), 65, 67, 68, 88
linalg_eigh(), 65, 67, 69, 82, 88
linalg_eigvals(), 65
linalg_eigvalsh(), 67
linalg_householder_product, 61, 62, 64, 65, 67–69, 70, 72, 73, 75–77, 79–81, 83–86, 88–92
linalg_inv(), 63, 73, 83
linalg_inv_ex, 61, 62, 64, 65, 67–69, 71, 72, 72, 75–77, 79–81, 83–86, 88–92
INDEX

linalg_matrix_power, 61, 62, 64, 65, 67–69, 71–73, 73, 76, 77, 79–81, 83–86, 88–92
linalg_lstsq(), 82, 83
linalg_matrix_norm, 61, 62, 64, 65, 67–69, 71–73, 75, 75, 77, 79–81, 83–86, 88–92
linalg_matrix_power, 61, 62, 64, 65, 67–69, 71–73, 75, 76, 77, 79–81, 83–86, 88–92
linalg_matrix_rank, 61, 62, 64, 65, 67–69, 71–73, 75–77, 78, 80, 81, 83–86, 88–92
linalg_norm, 61, 62, 64, 65, 67–69, 71–73, 75–77, 79, 80, 80, 83–86, 88–92
linalg_norm(), 63, 92
linalg_pinv, 61, 62, 64, 65, 67–69, 71–73, 75–77, 79–81, 82, 84–86, 88–92
linalg_pinv(), 72
linalg_qr(), 65, 67, 70, 88
linalg_slogdet, 61, 62, 64, 65, 67–69, 71–73, 75–77, 79–81, 83, 84, 84, 86, 88–92
linalg_solve(), 71, 72, 77
linalg_svd(), 63, 65, 67, 82, 88, 89
linalg_svdvals(), 88
linalg_tensorinv(), 91
linalg_tensorsolve, 61, 62, 64, 65, 67–69, 71–73, 75–77, 79–81, 83–86, 88–90, 90, 92
linalg_tensorsolve(), 89, 90
load_state_dict, 93
local_autocast, 93
local_device, 95
local_enable_grad (with_enable_grad), 579
local_no_grad (with_no_grad), 580
local_torch_manual_seed
(local_torch_manual_seed), 463
lr_cosine_annealing, 95
lr_lambda, 96
lr_multiplicative, 97
lr_one_cycle, 97
lr_reduce_on_plateau, 100
lr_scheduler, 101
lr_step, 102
nn_adaptive_avg_pool1d, 167
nn_adaptive_avg_pool1d(), 313
nn_adaptive_avg_pool2d, 167
nn_adaptive_avg_pool3d, 168
nn_adaptive_log_softmax_with_loss, 169
nn_adaptive_max_pool1d, 171
nn_adaptive_max_pool2d, 171
nn_adaptive_max_pool3d, 172
nn_avg_pool1d, 173
nn_avg_pool1d(), 338
nn_avg_pool2d, 174
nn_avg_pool3d, 176
nn_batch_norm1d, 177
nn_batch_norm2d, 179
nn_batch_norm3d, 180
nn_bce_loss, 182
nn_bce_with_logits_loss, 183
nn_bilinear, 185
nn_buffer, 186
nn_buffer(), 254
nn_celu, 186
nn_contrib_sparsemax, 187
nn_conv1d, 187
nn_conv1d(), 363
nn_conv2d, 189, 198, 207, 208
nn_conv2d(), 364
nn_conv3d, 192
nn_conv3d(), 365
nn_conv_transpose1d, 194
nn_conv_transpose1d(), 367
nn_conv_transpose2d, 196, 198
nn_conv_transpose2d(), 368
nn_conv_transpose3d, 199
nn_conv_transpose3d(), 369
nn_cosine_embedding_loss, 202
nn_cross_entropy_loss, 204
nn_dropout, 206
nn_dropout2d, 207, 207
nn_dropout3d, 208, 208
nn_elu, 209
nn_embedding, 209, 210
nn_embedding_bag, 211
nn_flatten, 212
nn_fractional_max_pool2d, 213
nn_fractional_max_pool3d, 214
nn_gelu, 215
nn_glu, 216
nn_group_norm, 216
nn_gru, 218
nn_hardshrink, 220
nn_hardsigmoid, 220
nn_hardtanh, 222
nn_hinge_embedding_loss, 223
nn_identity, 224
nn_init_calculate_gain, 224
nn_init_constant_, 225
nn_init_dirac_, 225
nn_init_eye_, 226
nn_init_kaiming_normal_, 226
nn_init_kaiming_uniform_, 227
nn_init_normal_, 228
nn_init_ones_, 229
nn_init_orthogonal_, 229
nn_init_sparse_, 230
nn_init_trunc_normal_, 230
nn_init_uniform_, 231
nn_init_xavier_normal_, 232
nn_init_xavier_uniform_, 232
nn_init_zeros_, 233
nn_kl_div_loss, 233
nn_l1_loss, 234
nn_layer_norm, 235
nn_leaky_relu, 237
nn_linear, 238
nn_log_sigmoid, 239
nn_log_softmax, 239
nn_log_softmax(), 202
nn_lp_pool1d, 240
nn_lq_pool2d, 241
nn_lstm, 243
nn_margin_ranking_loss, 245
nn_max_pool1d, 246
nn_max_pool1d(), 250
nn_max_pool2d, 247
nn_max_pool2d(), 251
nn_max_pool3d, 248
nn_max_pool3d(), 253
nn_max_unpool1d, 250
nn_max_unpool1d(), 171
nn_max_unpool2d, 251
nn_max_unpool2d(), 172, 214
nn_max_unpool3d, 252
nn_max_unpool3d(), 172, 214
nn_module, 253
nn_module(), 58
nn_module_dict, 255
nn_module_dict(), 256
nn_module_list, 256, 256
nn_module_list(), 255
nn_mse_loss, 257
nn_multi_margin_loss, 262
nn_multitask_attention, 258
nn_multilabel_margin_loss, 260
nn_multilabel_soft_margin_loss, 261
nn_nll_loss, 263
nn_nll_loss(), 234
nn_pairwise_distance, 264
nn_pairwise_distance(), 165, 286
nn_parameter, 265
nn_parameter(), 254
nn_poisson_nll_loss, 266
nn_prelu, 267
nn_prune_head, 268
nn_relu, 269
nn_relu6, 269
nn_rnn, 270
nn_rrelu, 272
nn_selu, 273
nn_sequential, 212, 274, 287
nn_sigmoid, 275
nn_silu, 275
nn_silu(), 159
nn_smooth_l1_loss, 276
nn_soft_margin_loss, 281
nn_softmax, 277
nn_softmax2d, 278
nn_softmin, 278
<table>
<thead>
<tr>
<th>Function</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnn_max_unpool2d</td>
<td>145</td>
</tr>
<tr>
<td>nnn_max_unpool3d</td>
<td>146</td>
</tr>
<tr>
<td>nnn_mse_loss</td>
<td>146</td>
</tr>
<tr>
<td>nnn_multi_head_attention_forward</td>
<td>148</td>
</tr>
<tr>
<td>nnn_multi_margin_loss</td>
<td>150</td>
</tr>
<tr>
<td>nnn_multilabel_margin_loss</td>
<td>147</td>
</tr>
<tr>
<td>nnn_multilabel_soft_margin_loss</td>
<td>147</td>
</tr>
<tr>
<td>nnn_nll_loss</td>
<td>151</td>
</tr>
<tr>
<td>nnn_normalize</td>
<td>152</td>
</tr>
<tr>
<td>nnn_one_hot</td>
<td>152</td>
</tr>
<tr>
<td>nnn_pad</td>
<td>153</td>
</tr>
<tr>
<td>nnn_pairwise_distance</td>
<td>154</td>
</tr>
<tr>
<td>nnn_pdist</td>
<td>154</td>
</tr>
<tr>
<td>nnn_pixel_shuffle</td>
<td>155</td>
</tr>
<tr>
<td>nnn_poisson_nll_loss</td>
<td>155</td>
</tr>
<tr>
<td>nnn_prelu</td>
<td>156</td>
</tr>
<tr>
<td>nnn_relu</td>
<td>156</td>
</tr>
<tr>
<td>nnn_relu6</td>
<td>157</td>
</tr>
<tr>
<td>nnn_relu_ (nnn_relu)</td>
<td>156</td>
</tr>
<tr>
<td>nnn_rrelu</td>
<td>157</td>
</tr>
<tr>
<td>nnn_rrelu_ (nnn_rrelu)</td>
<td>157</td>
</tr>
<tr>
<td>nnn_selu</td>
<td>158</td>
</tr>
<tr>
<td>nnn_selu_ (nnn_selu)</td>
<td>158</td>
</tr>
<tr>
<td>nnn_sigmoid</td>
<td>158</td>
</tr>
<tr>
<td>nnn_silu</td>
<td>159</td>
</tr>
<tr>
<td>nnn_smooth_l1_loss</td>
<td>159</td>
</tr>
<tr>
<td>nnn_soft_margin_loss</td>
<td>163</td>
</tr>
<tr>
<td>nnn_softmax</td>
<td>160, 161</td>
</tr>
<tr>
<td>nnn_softmax_(nnn_softmax)</td>
<td>160</td>
</tr>
<tr>
<td>nnn_softmin</td>
<td>160</td>
</tr>
<tr>
<td>nnn_softplus</td>
<td>161</td>
</tr>
<tr>
<td>nnn_softshrink</td>
<td>162</td>
</tr>
<tr>
<td>nnn_softsign</td>
<td>162</td>
</tr>
<tr>
<td>nnn_tanhshrink</td>
<td>163</td>
</tr>
<tr>
<td>nnn_threshold</td>
<td>164</td>
</tr>
<tr>
<td>nnn_threshold_ (nnn_threshold)</td>
<td>164</td>
</tr>
<tr>
<td>nnn_triplet_margin_loss</td>
<td>164</td>
</tr>
<tr>
<td>nnn_triplet_margin_loss_ with_distance_loss</td>
<td>165</td>
</tr>
<tr>
<td>nnn_unfold</td>
<td>166</td>
</tr>
<tr>
<td>optim_adadelta</td>
<td>298</td>
</tr>
<tr>
<td>optim_adagrad</td>
<td>299</td>
</tr>
<tr>
<td>optim_adam</td>
<td>301</td>
</tr>
<tr>
<td>optim_adamw</td>
<td>302</td>
</tr>
<tr>
<td>optim_asgd</td>
<td>302</td>
</tr>
<tr>
<td>optim_lbfgs</td>
<td>303</td>
</tr>
<tr>
<td>optim_required</td>
<td>305</td>
</tr>
<tr>
<td>optim_rmsprop</td>
<td>305</td>
</tr>
<tr>
<td>optim_rprop</td>
<td>307</td>
</tr>
<tr>
<td>optim_sgd</td>
<td>308</td>
</tr>
<tr>
<td>optimizor</td>
<td>297</td>
</tr>
<tr>
<td>R6::R6Class()</td>
<td>34, 53, 101, 254</td>
</tr>
<tr>
<td>sampler</td>
<td>309</td>
</tr>
<tr>
<td>sampler_ (nnn_sampler)</td>
<td>32, 33</td>
</tr>
<tr>
<td>tensor_dataset</td>
<td>310</td>
</tr>
<tr>
<td>threads</td>
<td>310</td>
</tr>
<tr>
<td>torch_abs</td>
<td>311</td>
</tr>
<tr>
<td>torch_abs_ (nnn_torch_abs)</td>
<td>311</td>
</tr>
<tr>
<td>torch_absolute</td>
<td>311</td>
</tr>
<tr>
<td>torch_acos</td>
<td>312</td>
</tr>
<tr>
<td>torch_acos_ (nnn_torch_acos)</td>
<td>326</td>
</tr>
<tr>
<td>torch_acosh</td>
<td>312</td>
</tr>
<tr>
<td>torch_acosh_ (nnn_torch_acosh)</td>
<td>326</td>
</tr>
<tr>
<td>torch_adaptive_avg_pool1d</td>
<td>313</td>
</tr>
<tr>
<td>torch_add</td>
<td>314</td>
</tr>
<tr>
<td>torch_addbmm</td>
<td>315</td>
</tr>
<tr>
<td>torch_addcdiv</td>
<td>316</td>
</tr>
<tr>
<td>torch_addcmul</td>
<td>317</td>
</tr>
<tr>
<td>torch_addmm</td>
<td>318</td>
</tr>
<tr>
<td>torch_addmv</td>
<td>319</td>
</tr>
<tr>
<td>torch_addr</td>
<td>320</td>
</tr>
<tr>
<td>torch_allclose</td>
<td>321</td>
</tr>
<tr>
<td>torch_amax</td>
<td>322</td>
</tr>
<tr>
<td>torch_amin</td>
<td>323</td>
</tr>
<tr>
<td>torch_angle</td>
<td>324</td>
</tr>
<tr>
<td>torch_arange</td>
<td>324, 516</td>
</tr>
<tr>
<td>torch_arccos</td>
<td>326</td>
</tr>
<tr>
<td>torch_arccosh</td>
<td>326</td>
</tr>
<tr>
<td>torch_arcsin</td>
<td>327</td>
</tr>
<tr>
<td>torch_arcsinh</td>
<td>327</td>
</tr>
<tr>
<td>torch_arctan</td>
<td>328</td>
</tr>
<tr>
<td>torch_arctanh</td>
<td>328</td>
</tr>
<tr>
<td>torch_argmax</td>
<td>329</td>
</tr>
<tr>
<td>torch_argmin</td>
<td>330</td>
</tr>
<tr>
<td>torch_argsort</td>
<td>331</td>
</tr>
<tr>
<td>torch_as_strided</td>
<td>333</td>
</tr>
<tr>
<td>torch_asin</td>
<td>331</td>
</tr>
<tr>
<td>torch_asin_ (nnn_torch_asin)</td>
<td>327</td>
</tr>
<tr>
<td>torch_asinh</td>
<td>332</td>
</tr>
<tr>
<td>torch_asinh_ (nnn_torch_asinh)</td>
<td>327</td>
</tr>
<tr>
<td>torch_atan</td>
<td>334</td>
</tr>
<tr>
<td>torch_atan_ (nnn_torch_atan)</td>
<td>328</td>
</tr>
<tr>
<td>torch_atan2</td>
<td>334</td>
</tr>
<tr>
<td>torch_atanh</td>
<td>335</td>
</tr>
<tr>
<td>torch_atanh_ (nnn_torch_atanh)</td>
<td>328</td>
</tr>
</tbody>
</table>
torch_erfc, 397
torch_erfinv, 398
torch_exp, 398
torch_exp2, 399
torch_expm1, 400
torch_eye, 400
torch_fft.fft, 401
torch_fft.fft(), 403, 406
torch_fft.fftfreq, 402
torch_fft.ifft, 403
torch_fft.irfft, 404
torch_fft.irfft(), 405
torch_fft.rfft, 405
torch_fft.rfft(), 404
torch_finfo, 406
torch_finfo(), 78
torch_fix, 406
torch_flatten, 407
torch_flip, 407
torch_fliplr, 408
torch_flipud, 409
torch_float(torch_dtype), 389
torch_float16(torch_dtype), 389
torch_float32(torch_dtype), 389
torch_float64(torch_dtype), 389
torch_floor, 410
torch_floor_divide, 410
torch_floor_divide(), 316, 386
torch_fmod, 411
torch_frac, 412
torch_full, 412
torch_full_like, 413
torch_gather, 414
torch_gcd, 415
torch_ge, 416
torch_ge(), 420
torch_generator, 417
torch_generator(), 490
torch_geqr() 417, 493, 494
torch_geqr(), 70, 494
torch ger, 418
torch_get_default_dtype
   (torch_set_default_dtype), 530
torch_get_default_dtype(), 402
torch_get_num_interop_threads
   (threads), 310
torch_get_num_threads(threads), 310
torch_get_rng_state, 419
torch_greater, 419
torch_greater_equal, 420
torch.gt, 420
torch.gt(), 420
torch_half(torch_dtype), 389
torch_hamming_window, 421
torch_hann_window, 422
torch_heaviside, 423
torch_histc, 424
torch_hstack, 425
torch_hypot, 425
torch_i0, 426
torch_i0(), 440
torch_iinfo, 427
torch imag, 427
torch_index, 428
torch_index_put, 428
torch_index_put(), 429
torch_index_select, 429
torch_install_path, 430
torch.int(torch_dtype), 389
torch.int16(torch_dtype), 389
torch.int32(torch_dtype), 389
torch.int64(torch_dtype), 389
torch.int64(), 348, 528
torch.int64(), 348, 528
torch.int8(torch_dtype), 389
torch.invert, 430
torch.is_complex, 437
torch.is_floating_point, 438
torch.is_installed, 438
torch.is_nonzero, 439
torch.isclose, 431
torch.isfinite, 432
torch.isinf, 433
torch.isnan, 433
torch.isneginf, 434
torch.isposinf, 434
torch.isposinf, 434
torch.isreal, 435
torch.isreal, 435
torch.istft, 436
torch.kaiser_window, 439
torch.kron, 441
torch.kthvalue, 441
torch.layout, 442
torch.lcm, 442
torch.le, 443
torch.le(), 445
torch.lerp, 444
torch_less, 445
torch_less_equal, 445
torch_loggamma, 446
torch_linspace, 446
torch_load, 447, 527, 530
torch_load(), 530
torch_log, 448, 449
torch_log10, 448
torch_log1p, 449
torch_log2, 450
torch_logaddexp, 451
torch_logaddexp(), 452
torch_logcumsumexp, 452
torch_logdet, 453
torch_logical_and, 454
torch_logical_not, 454
torch_logical_or, 455
torch_logical_xor, 456
torch_logit, 456
torch_logspace, 457
torch_logsumexp, 458
torch_long(), 389
torch_lstsq, 459
torch_lstsq(), 459, 460
torch_lt, 460
torch_lt(), 445
torch_lu, 461
torch_lu_solve, 461
torch_lu_unpack, 462
torch_manual_seed, 463
torch_manual_seed(), 419
torch_masked_select, 464
torch_matmul, 347, 465, 476
torch_matrix_exp, 466
torch_matrix_power, 467
torch_matrix_rank, 468
torch_max, 468
torch_maximum, 479
torch_mm, 476
torch_mm(), 80
torch_mode, 476
torch_movedim, 477
torch_multinomial, 479
torch_multinomial(), 40
torch_multiply, 480
torch_mv, 481
torch_mvlgamma, 482
torch_nanquantile, 482
torch_nansum, 483
torch_narrow, 484
torch_ne, 485
torch_ne(), 491
torch_neg, 486
torch_neg(), 486
torch_negative, 486
torch_nextafter, 487
torch_nonzero, 487
torch_nonzero(), 575
torch_norm, 488
torch_normal, 489
torch_not_equal, 491
torch_ones, 491
torch_ones_like, 492
torch_orgqr, 493
torch_ormqr, 494
torch_ormqr(), 71
torch_outer, 494
torch_pdist, 495
torch_per_channel_affine
  (torch_qscheme), 504
torch_per_channel_symmetric
  (torch_qscheme), 504
torch_per_tensor_affine
  (torch_qscheme), 504
torch_per_tensor_symmetric
  (torch_qscheme), 504
torch_pinverse, 496
torch_pixel_shuffle, 497
torch_poisson, 497
torch_polar, 498
torch_polygamma, 499
torch_pow, 500
torch_preserve_format
  (torch_memory_format), 472
torch_tensordot, 553
torch_tensordot(), 89, 91
torch_threshold, 554
torch_topk, 555
torch_trace, 556
torch_transpose, 556
torch_trapz, 557
torch_triangular_solve, 558
torch_tril, 559
torch_tril_indices, 560
torch_triu, 561
torch_triu_indices, 562
torch_true_divide, 564
torch_true_divide(), 316, 386
torch_trunc, 564
torch_trunc(), 406
torch_uint8 (torch_dtype), 389
torch_unbind, 565
torch_unique_consecutive, 566
torch_unsafe_chunk, 567
torch_unsafe_split, 567
torch_unsqueeze, 568
torch_vander, 569
torch_var, 569
torch_var_mean, 570
torch_vdot, 571
torch_view_as_complex, 572, 573
torch_view_as_real, 573
torch_view_as_real(), 573
torch_vstack, 574
torch_where, 574
torch_zeros, 575
torch_zeros_like, 576

with_autocast (local_autocast), 93
with_detect_anomaly, 578
with_device (local_device), 95
with_enable_grad, 579
with_no_grad, 579, 580
with_torch_manual_seed
(torch_manual_seed), 463