

# Package ‘torch’

February 14, 2025

**Type** Package

**Title** Tensors and Neural Networks with 'GPU' Acceleration

**Version** 0.14.2

**Description** Provides functionality to define and train neural networks similar to 'PyTorch' by Paszke et al (2019) <[doi:10.48550/arXiv.1912.01703](https://doi.org/10.48550/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

**URL** <https://torch.mlverse.org/docs>, <https://github.com/mlverse/torch>

**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 (>= 1.0.0), methods, utils, stats, bit64, magrittr, tools, coro (>= 1.0.2), callr, cli (>= 3.0.0), glue, desc, safetensors (>= 0.1.1), jsonlite, scales

**RoxygenNote** 7.3.2

**Suggests** testthat (>= 3.0.0), covr, knitr (>= 1.36), rmarkdown, palmerpenguins, mvtnorm, numDeriv, katex

**VignetteBuilder** knitr

**Collate** 'R7.R' 'RcppExports.R' 'autocast.R' 'tensor.R' 'autograd.R' 'backends.R' 'call\_torch\_function.R' 'codegen-utils.R' 'compare.R' 'compat-purrr.R' 'compilation\_unit.R' 'conditions.R' 'contrib.R' 'creation-ops.R' 'cuda.R' 'device.R' 'dimname\_list.R' 'utils.R' 'distributions-constraints.R' 'distributions-utils.R' 'distributions-exp-family.R' 'distributions.R' 'distributions-bernoulli.R' 'distributions-categorical.R' 'distributions-gamma.R'

'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' 'utils-data.R'  
'optim.R' 'optim-adam.R' 'optim-sgd.R' 'optim-rmsprop.R'  
'optim-adagrad.R' 'optim-adamw.R' 'ignite.R' 'indexing.R'  
'install.R' 'ivalue.R' 'jit-compile.R' 'jit-execute.R'  
'jit-ops.R' 'lantern\_load.R' 'lantern\_sync.R' 'layout.R'  
'linalg.R' 'memory\_format.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'  
'nfn-activation.R' 'nfn-batchnorm.R' 'nfn-conv.R'  
'nfn-distance.R' 'nfn-dropout.R' 'nfn-embedding.R' 'nfn-fold.R'  
'nfn-instancenorm.R' 'nfn-linear.R' 'nfn-loss.R'  
'nfn-normalization.R' 'nfn-padding.R' 'nfn-pixelshuffle.R'  
'nfn-pooling.R' 'nfn-upsampling.R' 'nfn-vision.R' 'operators.R'  
'optim-adadelta.R' 'optim-asgd.R' 'optim-lbfgs.R'  
'optim-lr\_scheduler.R' 'optim-rprop.R' 'positron.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],  
Sebastian Fischer [ctb],  
Maximilian Pichler [ctb],  
RStudio [cph]

**Maintainer** Daniel Falbel <daniel@rstudio.com>

**Repository** CRAN

**Date/Publication** 2025-02-14 22:40:02 UTC

## Contents

as\_array . . . . . 17

AutogradContext . . . . .	17
autograd_backward . . . . .	19
autograd_function . . . . .	20
autograd_grad . . . . .	21
autograd_set_grad_mode . . . . .	23
backends_cudnn_is_available . . . . .	23
backends_cudnn_version . . . . .	23
backends_mkldnn_is_available . . . . .	24
backends_mkl_is_available . . . . .	24
backends_mps_is_available . . . . .	24
backends_openmp_is_available . . . . .	25
broadcast_all . . . . .	25
clone_module . . . . .	26
Constraint . . . . .	26
contrib_sort_vertices . . . . .	27
cuda_amp_grad_scaler . . . . .	28
cuda_current_device . . . . .	29
cuda_device_count . . . . .	29
cuda_empty_cache . . . . .	29
cuda_get_device_capability . . . . .	30
cuda_is_available . . . . .	30
cuda_memory_stats . . . . .	30
cuda_runtime_version . . . . .	32
cuda_synchronize . . . . .	32
dataloader . . . . .	32
dataloader_make_iter . . . . .	34
dataloader_next . . . . .	35
dataset . . . . .	35
dataset_subset . . . . .	36
Distribution . . . . .	37
distr_bernoulli . . . . .	40
distr_categorical . . . . .	41
distr_chi2 . . . . .	42
distr_gamma . . . . .	42
distr_mixture_same_family . . . . .	43
distr_multivariate_normal . . . . .	44
distr_normal . . . . .	45
distr_poisson . . . . .	46
enumerate . . . . .	47
enumerate.dataloader . . . . .	47
get_install_libs_url . . . . .	48
install_torch . . . . .	49
is_dataloader . . . . .	50
is_nn_buffer . . . . .	50
is_nn_module . . . . .	51
is_nn_parameter . . . . .	51
is_optimizer . . . . .	51
is_torch_device . . . . .	52

is_torch_dtype . . . . .	52
is_torch_layout . . . . .	52
is_torch_memory_format . . . . .	53
is_torch_qscheme . . . . .	53
is_undefined_tensor . . . . .	53
iterable_dataset . . . . .	54
jit_compile . . . . .	55
jit_load . . . . .	55
jit_ops . . . . .	56
jit_save . . . . .	57
jit_save_for_mobile . . . . .	57
jit_scalar . . . . .	58
jit_serialize . . . . .	58
jit_trace . . . . .	59
jit_trace_module . . . . .	60
jit_tuple . . . . .	61
jit_unserialize . . . . .	62
linalg_cholesky . . . . .	62
linalg_cholesky_ex . . . . .	63
linalg_cond . . . . .	64
linalg_det . . . . .	66
linalg_eig . . . . .	67
linalg_eigh . . . . .	68
linalg_eigvals . . . . .	70
linalg_eigvalsh . . . . .	71
linalg_householder_product . . . . .	72
linalg_inv . . . . .	73
linalg_inv_ex . . . . .	74
linalg_lstsq . . . . .	75
linalg_matrix_norm . . . . .	77
linalg_matrix_power . . . . .	79
linalg_matrix_rank . . . . .	80
linalg_multi_dot . . . . .	81
linalg_norm . . . . .	82
linalg_pinv . . . . .	84
linalg_qr . . . . .	85
linalg_slogdet . . . . .	86
linalg_solve . . . . .	87
linalg_solve_triangular . . . . .	88
linalg_svd . . . . .	89
linalg_svdvals . . . . .	91
linalg_tensorinv . . . . .	92
linalg_tensorsolve . . . . .	93
linalg_vector_norm . . . . .	94
load_state_dict . . . . .	96
local_autocast . . . . .	96
local_device . . . . .	98
lr_cosine_annealing . . . . .	99

lr_lambda . . . . .	99
lr_multiplicative . . . . .	100
lr_one_cycle . . . . .	101
lr_reduce_on_plateau . . . . .	103
lr_scheduler . . . . .	105
lr_step . . . . .	105
nnf_adaptive_avg_pool1d . . . . .	106
nnf_adaptive_avg_pool2d . . . . .	106
nnf_adaptive_avg_pool3d . . . . .	107
nnf_adaptive_max_pool1d . . . . .	107
nnf_adaptive_max_pool2d . . . . .	108
nnf_adaptive_max_pool3d . . . . .	108
nnf_affine_grid . . . . .	109
nnf_alpha_dropout . . . . .	109
nnf_avg_pool1d . . . . .	110
nnf_avg_pool2d . . . . .	110
nnf_avg_pool3d . . . . .	111
nnf_batch_norm . . . . .	112
nnf_bilinear . . . . .	113
nnf_binary_cross_entropy . . . . .	113
nnf_binary_cross_entropy_with_logits . . . . .	114
nnf_celu . . . . .	115
nnf_contrib_sparsemax . . . . .	115
nnf_conv1d . . . . .	116
nnf_conv2d . . . . .	116
nnf_conv3d . . . . .	117
nnf_conv_tbc . . . . .	118
nnf_conv_transpose1d . . . . .	119
nnf_conv_transpose2d . . . . .	120
nnf_conv_transpose3d . . . . .	121
nnf_cosine_embedding_loss . . . . .	122
nnf_cosine_similarity . . . . .	122
nnf_cross_entropy . . . . .	123
nnf_ctc_loss . . . . .	124
nnf_dropout . . . . .	125
nnf_dropout2d . . . . .	125
nnf_dropout3d . . . . .	126
nnf_elu . . . . .	126
nnf_embedding . . . . .	127
nnf_embedding_bag . . . . .	128
nnf_fold . . . . .	129
nnf_fractional_max_pool2d . . . . .	130
nnf_fractional_max_pool3d . . . . .	131
nnf_gelu . . . . .	132
nnf_glu . . . . .	132
nnf_grid_sample . . . . .	133
nnf_group_norm . . . . .	134
nnf_gumbel_softmax . . . . .	135

nnf_hardshrink	135
nnf_hardsigmoid	136
nnf_hardswish	136
nnf_hardtanh	137
nnf_hinge_embedding_loss	137
nnf_instance_norm	138
nnf_interpolate	138
nnf_kl_div	140
nnf_l1_loss	140
nnf_layer_norm	141
nnf_leaky_relu	141
nnf_linear	142
nnf_local_response_norm	142
nnf_logsigmoid	143
nnf_log_softmax	143
nnf_lp_pool1d	144
nnf_lp_pool2d	144
nnf_margin_ranking_loss	145
nnf_max_pool1d	145
nnf_max_pool2d	146
nnf_max_pool3d	147
nnf_max_unpool1d	147
nnf_max_unpool2d	148
nnf_max_unpool3d	149
nnf_mse_loss	149
nnf_multilabel_margin_loss	150
nnf_multilabel_soft_margin_loss	150
nnf_multi_head_attention_forward	151
nnf_multi_margin_loss	153
nnf_nll_loss	154
nnf_normalize	155
nnf_one_hot	155
nnf_pad	156
nnf_pairwise_distance	157
nnf_pdist	157
nnf_pixel_shuffle	158
nnf_poisson_nll_loss	158
nnf_prelu	159
nnf_relu	159
nnf_relu6	160
nnf_rrelu	160
nnf_selu	161
nnf_sigmoid	161
nnf_silu	162
nnf_smooth_l1_loss	162
nnf_softmax	163
nnf_softmin	163
nnf_softplus	164

nnf_softshrink . . . . .	165
nnf_softsign . . . . .	165
nnf_soft_margin_loss . . . . .	166
nnf_tanhshrink . . . . .	166
nnf_threshold . . . . .	167
nnf_triplet_margin_loss . . . . .	167
nnf_triplet_margin_with_distance_loss . . . . .	168
nnf_unfold . . . . .	169
nn_adaptive_avg_pool1d . . . . .	170
nn_adaptive_avg_pool2d . . . . .	170
nn_adaptive_avg_pool3d . . . . .	171
nn_adaptive_log_softmax_with_loss . . . . .	172
nn_adaptive_max_pool1d . . . . .	174
nn_adaptive_max_pool2d . . . . .	174
nn_adaptive_max_pool3d . . . . .	175
nn_avg_pool1d . . . . .	176
nn_avg_pool2d . . . . .	177
nn_avg_pool3d . . . . .	179
nn_batch_norm1d . . . . .	180
nn_batch_norm2d . . . . .	182
nn_batch_norm3d . . . . .	183
nn_bce_loss . . . . .	185
nn_bce_with_logits_loss . . . . .	186
nn_bilinear . . . . .	188
nn_buffer . . . . .	189
nn_celu . . . . .	189
nn_contrib_sparsemax . . . . .	190
nn_conv1d . . . . .	190
nn_conv2d . . . . .	192
nn_conv3d . . . . .	195
nn_conv_transpose1d . . . . .	197
nn_conv_transpose2d . . . . .	199
nn_conv_transpose3d . . . . .	202
nn_cosine_embedding_loss . . . . .	205
nn_cross_entropy_loss . . . . .	205
nn_ctc_loss . . . . .	207
nn_dropout . . . . .	209
nn_dropout2d . . . . .	210
nn_dropout3d . . . . .	211
nn_elu . . . . .	212
nn_embedding . . . . .	212
nn_embedding_bag . . . . .	214
nn_flatten . . . . .	215
nn_fractional_max_pool2d . . . . .	216
nn_fractional_max_pool3d . . . . .	217
nn_gelu . . . . .	218
nn_glu . . . . .	219
nn_group_norm . . . . .	219

nn_gru . . . . .	221
nn_hardshrink . . . . .	223
nn_hardsigmoid . . . . .	223
nn_hardswish . . . . .	224
nn_hardtanh . . . . .	225
nn_hinge_embedding_loss . . . . .	226
nn_identity . . . . .	227
nn_init_calculate_gain . . . . .	227
nn_init_constant_ . . . . .	228
nn_init_dirac_ . . . . .	228
nn_init_eye_ . . . . .	229
nn_init_kaiming_normal_ . . . . .	229
nn_init_kaiming_uniform_ . . . . .	230
nn_init_normal_ . . . . .	231
nn_init_ones_ . . . . .	232
nn_init_orthogonal_ . . . . .	232
nn_init_sparse_ . . . . .	233
nn_init_trunc_normal_ . . . . .	233
nn_init_uniform_ . . . . .	234
nn_init_xavier_normal_ . . . . .	235
nn_init_xavier_uniform_ . . . . .	235
nn_init_zeros_ . . . . .	236
nn_kl_div_loss . . . . .	236
nn_l1_loss . . . . .	237
nn_layer_norm . . . . .	238
nn_leaky_relu . . . . .	240
nn_linear . . . . .	241
nn_log_sigmoid . . . . .	242
nn_log_softmax . . . . .	242
nn_lp_pool1d . . . . .	243
nn_lp_pool2d . . . . .	244
nn_lstm . . . . .	246
nn_margin_ranking_loss . . . . .	248
nn_max_pool1d . . . . .	249
nn_max_pool2d . . . . .	250
nn_max_pool3d . . . . .	251
nn_max_unpool1d . . . . .	253
nn_max_unpool2d . . . . .	254
nn_max_unpool3d . . . . .	255
nn_module . . . . .	256
nn_module_dict . . . . .	258
nn_module_list . . . . .	259
nn_mse_loss . . . . .	260
nn_multihead_attention . . . . .	261
nn_multilabel_margin_loss . . . . .	263
nn_multilabel_soft_margin_loss . . . . .	264
nn_multi_margin_loss . . . . .	265
nn_nll_loss . . . . .	266



nn_pairwise_distance . . . . .	267
nn_parameter . . . . .	268
nn_poisson_nll_loss . . . . .	269
nn_prelu . . . . .	270
nn_prune_head . . . . .	271
nn_relu . . . . .	272
nn_relu6 . . . . .	272
nn_rnn . . . . .	273
nn_rrelu . . . . .	275
nn_selu . . . . .	276
nn_sequential . . . . .	277
nn_sigmoid . . . . .	278
nn_silu . . . . .	278
nn_smooth_l1_loss . . . . .	279
nn_softmax . . . . .	280
nn_softmax2d . . . . .	281
nn_softmin . . . . .	281
nn_softplus . . . . .	282
nn_softshrink . . . . .	283
nn_softsign . . . . .	284
nn_soft_margin_loss . . . . .	284
nn_tanh . . . . .	285
nn_tanhshrink . . . . .	286
nn_threshold . . . . .	286
nn_triplet_margin_loss . . . . .	287
nn_triplet_margin_with_distance_loss . . . . .	288
nn_unflatten . . . . .	290
nn_upsample . . . . .	291
nn_utils_clip_grad_norm_ . . . . .	292
nn_utils_clip_grad_value_ . . . . .	293
nn_utils_rnn_pack_padded_sequence . . . . .	293
nn_utils_rnn_pack_sequence . . . . .	294
nn_utils_rnn_pad_packed_sequence . . . . .	295
nn_utils_rnn_pad_sequence . . . . .	296
nn_utils_weight_norm . . . . .	297
optimizer . . . . .	300
OptimizerIgnite . . . . .	301
optimizer_ignite . . . . .	303
optim_adadelta . . . . .	304
optim_adagrad . . . . .	305
optim_adam . . . . .	306
optim_adamw . . . . .	307
optim_asgd . . . . .	308
optim_ignite_adagrad . . . . .	309
optim_ignite_adam . . . . .	310
optim_ignite_adamw . . . . .	311
optim_ignite_rmsprop . . . . .	312
optim_ignite_sgd . . . . .	313

optim_lbfgs . . . . .	314
optim_required . . . . .	315
optim_rmsprop . . . . .	316
optim_rprop . . . . .	317
optim_sgd . . . . .	318
sampler . . . . .	319
tensor_dataset . . . . .	320
threads . . . . .	320
torch_abs . . . . .	321
torch_absolute . . . . .	321
torch_acos . . . . .	322
torch_acosh . . . . .	322
torch_adaptive_avg_pool1d . . . . .	323
torch_add . . . . .	324
torch_addbmm . . . . .	325
torch_addcddiv . . . . .	326
torch_addcmul . . . . .	327
torch_addmm . . . . .	328
torch_addmv . . . . .	329
torch_addr . . . . .	330
torch_allclose . . . . .	331
torch_amax . . . . .	332
torch_amin . . . . .	333
torch_angle . . . . .	334
torch_arange . . . . .	334
torch_arccos . . . . .	336
torch_arccosh . . . . .	336
torch_arcsin . . . . .	337
torch_arcsinh . . . . .	337
torch_arctan . . . . .	338
torch_arctanh . . . . .	338
torch_argmax . . . . .	339
torch_argmin . . . . .	340
torch_argsort . . . . .	341
torch_asin . . . . .	341
torch_asinh . . . . .	342
torch_as_strided . . . . .	343
torch_atan . . . . .	344
torch_atan2 . . . . .	344
torch_atanh . . . . .	345
torch_atleast_1d . . . . .	346
torch_atleast_2d . . . . .	347
torch_atleast_3d . . . . .	347
torch_avg_pool1d . . . . .	348
torch_baddbmm . . . . .	349
torch_bartlett_window . . . . .	350
torch_bernoulli . . . . .	351
torch_bincount . . . . .	352

torch_bitwise_and . . . . .	353
torch_bitwise_not . . . . .	353
torch_bitwise_or . . . . .	354
torch_bitwise_xor . . . . .	354
torch_blackman_window . . . . .	355
torch_block_diag . . . . .	356
torch_bmm . . . . .	356
torch_broadcast_tensors . . . . .	357
torch_bucketize . . . . .	358
torch_can_cast . . . . .	359
torch_cartesian_prod . . . . .	359
torch_cat . . . . .	360
torch_cdists . . . . .	361
torch_ceil . . . . .	361
torch_celu . . . . .	362
torch_celu_ . . . . .	363
torch_chain_matmul . . . . .	363
torch_channel_shuffle . . . . .	364
torch_cholesky . . . . .	365
torch_cholesky_inverse . . . . .	366
torch_cholesky_solve . . . . .	367
torch_chunk . . . . .	368
torch_clamp . . . . .	368
torch_clip . . . . .	369
torch_clone . . . . .	370
torch_combinations . . . . .	371
torch_complex . . . . .	371
torch_conj . . . . .	372
torch_conv1d . . . . .	373
torch_conv2d . . . . .	374
torch_conv3d . . . . .	375
torch_conv_tbc . . . . .	376
torch_conv_transpose1d . . . . .	376
torch_conv_transpose2d . . . . .	377
torch_conv_transpose3d . . . . .	379
torch_cos . . . . .	380
torch_cosh . . . . .	381
torch_cosine_similarity . . . . .	381
torch_count_nonzero . . . . .	382
torch_cross . . . . .	383
torch_cummax . . . . .	384
torch_cummin . . . . .	384
torch_cumprod . . . . .	385
torch_cumsum . . . . .	386
torch_deg2rad . . . . .	387
torch_dequantize . . . . .	387
torch_det . . . . .	388
torch_device . . . . .	389

torch_diag . . . . .	389
torch_diagflat . . . . .	390
torch_diagonal . . . . .	391
torch_diag_embed . . . . .	392
torch_diff . . . . .	393
torch_digamma . . . . .	394
torch_dist . . . . .	394
torch_div . . . . .	395
torch_divide . . . . .	396
torch_dot . . . . .	397
torch_dstack . . . . .	398
torch_dtype . . . . .	399
torch_eig . . . . .	400
torch_einsum . . . . .	400
torch_empty . . . . .	402
torch_empty_like . . . . .	403
torch_empty_strided . . . . .	404
torch_eq . . . . .	405
torch_equal . . . . .	406
torch_erf . . . . .	406
torch_erfc . . . . .	407
torch_erfinv . . . . .	408
torch_exp . . . . .	408
torch_exp2 . . . . .	409
torch_expm1 . . . . .	410
torch_eye . . . . .	410
torch_fft_fft . . . . .	411
torch_fft_fftfreq . . . . .	412
torch_fft_ifft . . . . .	413
torch_fft_irfft . . . . .	414
torch_fft_rfft . . . . .	415
torch_finfo . . . . .	416
torch_fix . . . . .	416
torch_flatten . . . . .	417
torch_flip . . . . .	417
torch_fliplr . . . . .	418
torch_flipud . . . . .	419
torch_floor . . . . .	420
torch_floor_divide . . . . .	420
torch_fmod . . . . .	421
torch_frac . . . . .	422
torch_full . . . . .	422
torch_full_like . . . . .	423
torch_gather . . . . .	424
torch_gcd . . . . .	425
torch_ge . . . . .	426
torch_generator . . . . .	427
torch_geqrf . . . . .	427

<code>torch_ger</code>	428
<code>torch_get_rng_state</code>	429
<code>torch_greater</code>	429
<code>torch_greater_equal</code>	430
<code>torch_gt</code>	430
<code>torch_hamming_window</code>	431
<code>torch_hann_window</code>	432
<code>torch_heaviside</code>	433
<code>torch_hisc</code>	434
<code>torch_hstack</code>	435
<code>torch_hypot</code>	435
<code>torch_i0</code>	436
<code>torch_iinfo</code>	437
<code>torch_imag</code>	437
<code>torch_index</code>	438
<code>torch_index_put</code>	438
<code>torch_index_put_</code>	439
<code>torch_index_select</code>	439
<code>torch_install_path</code>	440
<code>torch_inverse</code>	440
<code>torch_isclose</code>	441
<code>torch_isfinite</code>	442
<code>torch_isinf</code>	443
<code>torch_isnan</code>	443
<code>torch_isneginf</code>	444
<code>torch_isposinf</code>	444
<code>torch_isreal</code>	445
<code>torch_istft</code>	446
<code>torch_is_complex</code>	447
<code>torch_is_floating_point</code>	448
<code>torch_is_installed</code>	448
<code>torch_is_nonzero</code>	449
<code>torch_kaiser_window</code>	449
<code>torch_kron</code>	451
<code>torch_kthvalue</code>	451
<code>torch_layout</code>	452
<code>torch_lcm</code>	452
<code>torch_le</code>	453
<code>torch_lerp</code>	454
<code>torch_less</code>	455
<code>torch_less_equal</code>	455
<code>torch_lgamma</code>	456
<code>torch_linspace</code>	456
<code>torch_load</code>	457
<code>torch_log</code>	458
<code>torch_log10</code>	458
<code>torch_log1p</code>	459
<code>torch_log2</code>	460

torch_logaddexp	461
torch_logaddexp2	461
torch_logcumsumexp	462
torch_logdet	463
torch_logical_and	464
torch_logical_not	464
torch_logical_or	465
torch_logical_xor	466
torch_logit	466
torch_logspace	467
torch_logsumexp	468
torch_lstsq	469
torch_lt	470
torch_lu	471
torch_lu_solve	471
torch_lu_unpack	472
torch_manual_seed	473
torch_masked_select	474
torch_matmul	475
torch_matrix_exp	476
torch_matrix_power	477
torch_matrix_rank	478
torch_max	478
torch_maximum	480
torch_mean	480
torch_median	481
torch_memory_format	482
torch_meshgrid	483
torch_min	483
torch_minimum	485
torch_mm	486
torch_mode	486
torch_movedim	487
torch_mul	488
torch_multinomial	489
torch_multiply	490
torch_mv	491
torch_mvlgamma	491
torch_nanquantile	492
torch_nansum	493
torch_narrow	494
torch_ne	495
torch_neg	496
torch_negative	496
torch_nextafter	497
torch_nonzero	497
torch_norm	498
torch_normal	499

torch_not_equal . . . . .	501
torch_ones . . . . .	501
torch_ones_like . . . . .	502
torch_orgqr . . . . .	503
torch_ormqr . . . . .	504
torch_outer . . . . .	504
torch_pdist . . . . .	505
torch_pinverse . . . . .	506
torch_pixel_shuffle . . . . .	507
torch_poisson . . . . .	507
torch_polar . . . . .	508
torch_polygamma . . . . .	509
torch_pow . . . . .	510
torch_prod . . . . .	511
torch_promote_types . . . . .	512
torch_qr . . . . .	513
torch_qscheme . . . . .	514
torch_quantile . . . . .	514
torch_quantize_per_channel . . . . .	515
torch_quantize_per_tensor . . . . .	516
torch_rad2deg . . . . .	517
torch_rand . . . . .	517
torch_randint . . . . .	518
torch_randint_like . . . . .	519
torch_randn . . . . .	520
torch_randn_like . . . . .	522
torch_randperm . . . . .	523
torch_rand_like . . . . .	524
torch_range . . . . .	525
torch_real . . . . .	526
torch_reciprocal . . . . .	527
torch_reduction . . . . .	527
torch_relu . . . . .	528
torch_relu_ . . . . .	528
torch_remainder . . . . .	529
torch_renorm . . . . .	529
torch_repeat_interleave . . . . .	530
torch_reshape . . . . .	531
torch_result_type . . . . .	532
torch_roll . . . . .	533
torch_rot90 . . . . .	533
torch_round . . . . .	534
torch_rrelu_ . . . . .	535
torch_rsqr . . . . .	536
torch_save . . . . .	536
torch_scalar_tensor . . . . .	537
torch_searchsorted . . . . .	538
torch_selu . . . . .	539

torch_selu_ . . . . .	539
torch_serialize . . . . .	540
torch_set_default_dtype . . . . .	540
torch_sgn . . . . .	541
torch_sigmoid . . . . .	541
torch_sign . . . . .	542
torch_signbit . . . . .	543
torch_sin . . . . .	543
torch_sinh . . . . .	544
torch_slogdet . . . . .	545
torch_sort . . . . .	546
torch_sparse_coo_tensor . . . . .	546
torch_split . . . . .	548
torch_sqrt . . . . .	549
torch_square . . . . .	549
torch_squeeze . . . . .	550
torch_stack . . . . .	551
torch_std . . . . .	551
torch_std_mean . . . . .	552
torch_stft . . . . .	553
torch_sub . . . . .	556
torch_subtract . . . . .	556
torch_sum . . . . .	557
torch_svd . . . . .	558
torch_t . . . . .	559
torch_take . . . . .	560
torch_take_along_dim . . . . .	561
torch_tan . . . . .	562
torch_tanh . . . . .	562
torch_tensor . . . . .	563
torch_tensordot . . . . .	564
torch_tensor_from_buffer . . . . .	564
torch_threshold_ . . . . .	565
torch_topk . . . . .	566
torch_trace . . . . .	567
torch_transpose . . . . .	567
torch_trapz . . . . .	568
torch_triangular_solve . . . . .	569
torch_tril . . . . .	570
torch_tril_indices . . . . .	571
torch_triu . . . . .	572
torch_triu_indices . . . . .	573
torch_true_divide . . . . .	575
torch_trunc . . . . .	575
torch_unbind . . . . .	576
torch_unique_consecutive . . . . .	577
torch_unsafe_chunk . . . . .	578
torch_unsafe_split . . . . .	578



torch_unsqueeze . . . . .	579
torch_vander . . . . .	580
torch_var . . . . .	580
torch_var_mean . . . . .	581
torch_vdot . . . . .	582
torch_view_as_complex . . . . .	583
torch_view_as_real . . . . .	584
torch_vstack . . . . .	585
torch_where . . . . .	585
torch_zeros . . . . .	586
torch_zeros_like . . . . .	587
with_detect_anomaly . . . . .	589
with_enable_grad . . . . .	590
with_no_grad . . . . .	591

**Index****592**


---

as_array	<i>Converts to array</i>
----------	--------------------------

---

**Description**

Converts to array

**Usage**

as\_array(x)

**Arguments**

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

---

AutogradContext	<i>Class representing the context.</i>
-----------------	--

---

**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 computation. 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(...)
```

*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 value would be acceptable for all grad_tensors, then this argument is optional.
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**

```
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**

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

**Arguments**

forward	Performs the operation. It must accept a context <code>ctx</code> 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 <a href="#">AutogradContext</a> for more information about context methods.
backward	Defines a formula for differentiating the operation. It must accept a context <code>ctx</code> as the first argument, followed by as many outputs as <code>forward()</code> returned (as a <code>list()</code> ). The names of the arguments don't matter and they are passed in the order in which they were returned by <code>forward()</code> . 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 <code>ctx\$needs_input_grad</code> as a named list of booleans representing whether each input needs gradient. E.g., <code>backward()</code> will have <code>ctx\$needs_input_grad\$input = TRUE</code> if the input argument to <code>forward()</code> needs gradient computed w.r.t. the output. See <a href="#">AutogradContext</a> for more information about context methods.

**Examples**

```
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**

```
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.
create_graph	(bool, optional) – If TRUE, graph of the derivative will be constructed, allowing to compute h
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. Defaults to 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

```

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 wether to enable or disable the gradient recording.

---

backends\_cudnn\_is\_available  
*CuDNN is available*

---

**Description**

CuDNN is available

**Usage**

backends\_cudnn\_is\_available()

---

backends\_cudnn\_version  
*CuDNN version*

---

**Description**

CuDNN version

**Usage**

backends\_cudnn\_version()

backends\_mkldnn\_is\_available

*MKLDNN is available*

---

**Description**

MKLDNN is available

**Usage**

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.

---

broadcast\_all

*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/m>

**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**)

---

clone_module	<i>Clone a torch module.</i>
--------------	------------------------------

---

**Description**

Clones a module.

**Usage**

```
clone_module(module, deep = FALSE, ..., replace_values = TRUE)
```

**Arguments**

module	( <a href="#">nn_module</a> ) The module to clone
deep	(logical(1)) Whether to create a deep clone.
...	(any) Additional parameters, currently unused.
replace_values	(logical(1)) Whether to replace parameters and buffers with the cloned values.

**Examples**

```
if (torch_is_installed()) {
  clone_module(nn_linear(1, 1), deep = TRUE)
  # is the same as
  nn_linear(1, 1)$clone(deep = TRUE)
}
```

---

Constraint	<i>Abstract base class for constraints.</i>
------------	---

---

**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**

<code>vertices</code>	A Tensor with the vertices.
<code>mask</code>	A tensors containing the masks.
<code>num_valid</code>	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.

**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**

```
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  $\geq$  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, <code>shuffle</code> must be False. Custom samplers can be created with <a href="#">sampler()</a> .
batch_sampler	(Sampler, optional): like sampler, but returns a batch of indices at a time. Mutually exclusive with <code>batch_size</code> , <code>shuffle</code> , <code>sampler</code> , and <code>drop_last</code> . Custom samplers can be created with <a href="#">sampler()</a> .
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 <code>collate_fn</code> 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 <code>num_workers &gt; 0</code> . 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 through 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_make_iter` *Creates an iterator from a DataLoader*

---

### Description

Creates an iterator from a `DataLoader`

### Usage

```
dataloader_make_iter(dataloader)
```

### Arguments

`dataloader` a `dataloader` object.

---

dataloader_next	<i>Get the next element of a dataloader iterator</i>
-----------------	--

---

**Description**

Get the next element of a dataloader iterator

**Usage**

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

**Arguments**

iter	a DataLoader iter created with <a href="#">dataloader_make_iter</a> .
completed	the returned value when the iterator is exhausted.

---

dataset	<i>Helper function to create an function that generates R6 instances of class dataset</i>
---------	---

---

**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 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 <a href="#">R6::R6Class()</a> .
active	passed to <a href="#">R6::R6Class()</a> .
parent_env	An environment to use as the parent of newly-created objects.

**Value**

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 `fs` 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 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**

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

**Arguments**

<code>dataset</code>	(Dataset): The whole Dataset
<code>indices</code>	(sequence): Indices in the whole set selected for subset

---

 Distribution

*Generic R6 class representing distributions*


---

**Description**

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_rsampl` 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$new()`
- `Distribution$expand()`
- `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\$cdf(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 (c(0), c(1, 1), ...). With expand=FALSE, enumeration happens along dim 0, but with the remaining batch dimensions in lock-step (c(1), ...).

*Usage:*

Distribution\$icdf(value)

*Arguments:*

value values to evaluate the density on.

**Method** enumerate\_support():

*Usage:*

Distribution\$enumerate\_support(expand = 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\$entropy()

*Returns:* Tensor of shape batch\_shape.

**Method** perplexity(): Returns perplexity of distribution, batched over batch\_shape.

*Usage:*

Distribution\$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\$.extended\_shape(sample\_shape = 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:*

deep Whether to make a deep clone.

---

distr_bernoulli	<i>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.</i>
-----------------	--

---

**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\\_chi2\(\)](#), [distr\\_gamma\(\)](#), [distr\\_multivariate\\_normal\(\)](#), [distr\\_normal\(\)](#), [distr\\_poisson\(\)](#)

**Examples**

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



---

distr_categorical	<i>Creates a categorical distribution parameterized by either probs or logits (but not both).</i>
-------------------	---

---

### Description

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

### Usage

```
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  $\{0, \dots, K - 1\}$  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

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

---

distr_chi2	<i>Creates a Chi2 distribution parameterized by shape parameter df. This is exactly equivalent to <code>distr_gamma(alpha=0.5*df, beta=0.5)</code></i>
------------	--

---

### 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

```
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

```
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	<i>Creates a Gamma distribution parameterized by shape concentration and rate.</i>
-------------	--

---

### Description

Creates a Gamma distribution parameterized by shape concentration and rate.

### Usage

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

**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**

```
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 component 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**

```
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]

`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 whether 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  $\mathbf{L}$  with positive-valued diagonal entries, such that  $\Sigma = \mathbf{L}\mathbf{L}^\top$ . 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\\_bernoulli\(\)](#), [distr\\_chi2\(\)](#), [distr\\_gamma\(\)](#), [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	<i>Creates a normal (also called Gaussian) distribution parameterized by loc and scale.</i>
--------------	---

---

**Description**

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

**Usage**

```
distr_normal(loc, scale, validate_args = NULL)
```

**Arguments**

<code>loc</code>	(float or Tensor): mean of the distribution (often referred to as $\mu$ )
<code>scale</code>	(float or Tensor): standard deviation of the distribution (often referred to as $\sigma$ )
<code>validate_args</code>	Additional arguments

**Value**

Object of `torch_Normal` class

**See Also**

[Distribution](#) for details on the available methods.

Other distributions: [distr\\_bernoulli\(\)](#), [distr\\_chi2\(\)](#), [distr\\_gamma\(\)](#), [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	<i>Creates a Poisson distribution parameterized by rate, the rate parameter.</i>
---------------	--

---

**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\\_chi2\(\)](#), [distr\\_gamma\(\)](#), [distr\\_multivariate\\_normal\(\)](#), [distr\\_normal\(\)](#)

**Examples**

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

---

enumerate	<i>Enumerate an iterator</i>
-----------	------------------------------

---

**Description**

Enumerate an iterator

**Usage**

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

**Arguments**

x	the generator to enumerate.
...	passed to specific methods.

---

enumerate.dataloader	<i>Enumerate an iterator</i>
----------------------	------------------------------

---

**Description**

Enumerate an iterator

**Usage**

```
## 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.

---

get\_install\_libs\_url *Install Torch from files*

---

### 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

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

```
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

```
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	<i>Install Torch</i>
---------------	----------------------

---

## 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 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	<i>Checks if the object is a dataloader</i>
---------------	---

---

### **Description**

Checks if the object is a dataloader

### **Usage**

```
is_dataloader(x)
```

### **Arguments**

x	object to check
---	-----------------

---

is_nn_buffer	<i>Checks if the object is a nn_buffer</i>
--------------	--

---

### **Description**

Checks if the object is a nn\_buffer

### **Usage**

```
is_nn_buffer(x)
```

### **Arguments**

x	object to check
---	-----------------

---

is_nn_module	<i>Checks if the object is an nn_module</i>
--------------	---

---

**Description**

Checks if the object is an nn\_module

**Usage**

```
is_nn_module(x)
```

**Arguments**

x	object to check
---	-----------------

---

is_nn_parameter	<i>Checks if an object is a nn_parameter</i>
-----------------	--

---

**Description**

Checks if an object is a nn\_parameter

**Usage**

```
is_nn_parameter(x)
```

**Arguments**

x	the object to check
---	---------------------

---

is_optimizer	<i>Checks if the object is a torch optimizer</i>
--------------	--

---

**Description**

Checks if the object is a torch optimizer

**Usage**

```
is_optimizer(x)
```

**Arguments**

x	object to check
---	-----------------

---

is_torch_device	<i>Checks if object is a device</i>
-----------------	-------------------------------------

---

**Description**

Checks if object is a device

**Usage**

```
is_torch_device(x)
```

**Arguments**

x	object to check
---	-----------------

---

is_torch_dtype	<i>Check if object is a torch data type</i>
----------------	---

---

**Description**

Check if object is a torch data type

**Usage**

```
is_torch_dtype(x)
```

**Arguments**

x	object to check.
---	------------------

---

is_torch_layout	<i>Check if an object is a torch layout.</i>
-----------------	--

---

**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**`is_torch_memory_format(x)`**Arguments**

<code>x</code>	object to check
----------------	-----------------

---

`is_torch_qscheme`*Checks if an object is a QScheme*

---

**Description**

Checks if an object is a QScheme

**Usage**`is_torch_qscheme(x)`**Arguments**

<code>x</code>	object to check
----------------	-----------------

---

`is_undefined_tensor`*Checks if a tensor is undefined*

---

**Description**

Checks if a tensor is undefined

**Usage**`is_undefined_tensor(x)`**Arguments**

<code>x</code>	tensor to check
----------------	-----------------

---

iterable_dataset	<i>Creates an iterable dataset</i>
------------------	------------------------------------

---

## 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 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 <code>R6::R6Class()</code> .
active	passed to <code>R6::R6Class()</code> .
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  
        }  
      }  
    }  
  )  
}
```

```
)  
coro::collect(ids()$.iter())  
}
```

---

jit_compile	<i>Compile TorchScript code into a graph</i>
-------------	--

---

### 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	<i>Loads a script_function or script_module previously saved with jit_save</i>
----------	--

---

### Description

Loads a script\_function or script\_module previously saved with jit\_save

### Usage

```
jit_load(path, ...)
```

**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**

```
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	<i>Saves a script_function to a path</i>
----------	--

---

**Description**

Saves a script\_function to a path

**Usage**

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

**Arguments**

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

**Examples**

```
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	<i>Saves a script_function or script_module in bytecode form, to be loaded on a mobile device</i>
---------------------	---

---

**Description**

Saves a script\_function or script\_module in bytecode form, to be loaded on a mobile device

**Usage**

```
jit_save_for_mobile(obj, path, ...)
```

**Arguments**

obj	An script_function or script_module to save
path	The path to save the serialized function.
...	currently unused

**Examples**

```
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	<i>Adds the 'jit_scalar' class to the input</i>
------------	---

---

**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_serialize	<i>Serialize a Script Module</i>
---------------	----------------------------------

---

**Description**

Serializes a script module and returns it as a raw vector. You can read the object again using [jit\\_unserialize](#).

**Usage**

```
jit_serialize(obj)
```

**Arguments**

obj (script\_module)  
Model to be serialized.

**Value**

raw()

**Examples**

```
if (torch_is_installed()) {
  model <- jit_trace(nn_linear(1, 1), torch_randn(1))
  serialized <- jit_serialize(model)
}
```

---

`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, respect_mode = 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.

... example 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.

respect\_mode (logical(1))  
Whether both modes ('train' or 'eval') should be traced. If TRUE (default), the underlying C++ ScriptModule will have two methods `trainforward()` and `evalforward()`. The `$forward()` method of the R torch module will then select either based on the mode. If FALSE, only the current mode of the module will be jitted and hence only one `forward()` method exists.

**Details**

The resulting recording of a standalone function produces a `script_function`.

**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. For scripting, see [jit\\_compile](#).

**Examples**

```
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**

```
jit_trace_module(mod, ..., strict = TRUE, respect_mode = 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.
respect_mode	(logical(1)) Whether both modes ('train' or 'eval') should be traced. If TRUE (default), the underlying C++ ScriptModule will have two methods trainforward() and evalforward(). The \$forward() method of the R torch module will then select either based on the mode. If FALSE, only the current mode of the module will be jitted and hence only one forward() method exists.

**Details**

See [jit\\_trace](#) for more information on tracing.

**Examples**

```
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.

---

jit\_unserialize *Unserialize a Script Module*

---

**Description**

Unserializes a script module from a raw vector (generated with `jit_serialize`).

**Usage**

```
jit_unserialize(obj)
```

**Arguments**

obj (raw)  
Serialized model.

**Value**

```
script_module model <- jit_trace(nn_linear(1, 1), torch_randn(1))
serialized <- jit_serialize(model)
model2 <- jit_unserialize(serialized)
```

---

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. Please find it in '<https://torch.mlverse.org/docs>'

where  $L$  is a lower triangular matrix and  $A$  is the conjugate transpose when  $A$  is complex, and the transpose when  $A$  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\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

## Examples

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

---

linalg_cholesky_ex	<i>Computes the Cholesky decomposition of a complex Hermitian or real symmetric positive-definite matrix.</i>
--------------------	---

---

## 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**

```
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\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

**Examples**

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

---

`linalg_cond`

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

---

**Description**

Letting  $\kappa$  be  $\kappa_p$  or  $\kappa_2$ , the **condition number**  $\kappa$  of a matrix  $A$  is defined as

**Usage**

```
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

**Details**

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

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  $(\text{'fro'}, \text{'nuc'}, \text{inf}, -\text{inf}, 1, -1)$ , this function uses [linalg\\_norm\(\)](#) and [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. Please find it in '<https://torch.mlverse.org/docs>'

In these cases, it is computed using [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	$\max(\text{sum}(\text{abs}(x), \text{dim}=2))$
-Inf	$\min(\text{sum}(\text{abs}(x), \text{dim}=2))$
1	$\max(\text{sum}(\text{abs}(x), \text{dim}=1))$
-1	$\min(\text{sum}(\text{abs}(x), \text{dim}=1))$
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  $(\text{'fro'}, \text{'nuc'}, \text{inf}, -\text{inf}, 1, -1)$ .

**Examples**

```

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\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

**Examples**

```

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  $\lambda$  be  $\lambda_1$  or  $\lambda_2$ , the **eigenvalue decomposition** of a square matrix  $A$  (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. Please find it in '<https://torch.mlverse.org/docs>'

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 (eigenvalues, eigenvectors) which corresponds to  $\lambda$  and  $V$  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  $V$  will be incorrect.
- 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  $\lambda$  through the computation of  $V$ .

**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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

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

---

`linalg_eigh`

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

---

**Description**

Letting `.` be `.` or `.`, the **eigenvalue decomposition** of a complex Hermitian or real symmetric matrix `.` is defined as

**Usage**

```
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. Please find it in '<https://torch.mlverse.org/docs>'

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  $UPLO = 'L'$  (default), only the lower triangular part of the matrix is used in the computation.
- If  $UPLO = 'U'$ , 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  $\cdot$  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  $\cdot$  through the computation of  $\cdot$ .

**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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

## Examples

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

---

<code>linalg_eigvals</code>	<i>Computes the eigenvalues of a square matrix.</i>
-----------------------------	---

---

## Description

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

## Usage

```
linalg_eigvals(A)
```

## Arguments

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

## Details

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

where  $I$  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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

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

---

<code>linalg_eigvalsh</code>	<i>Computes the eigenvalues of a complex Hermitian or real symmetric matrix.</i>
------------------------------	--

---

**Description**

Letting  $\lambda$  be  $\cdot$  or  $\cdot$ , the **eigenvalues** of a complex Hermitian or real symmetric matrix  $A$  are defined as the roots (counted with multiplicity) of the polynomial  $p$  of degree  $n$  given by

**Usage**

```
linalg_eigvalsh(A, UPL0 = "L")
```

**Arguments**

<code>A</code>	(Tensor): tensor of shape $(*, n, n)$ where $*$ is zero or more batch dimensions consisting of symmetric or Hermitian matrices.
<code>UPL0</code>	('L', 'U', optional): controls whether to use the upper or lower triangular part of $A$ in the computations. Default: 'L'.

**Details**

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

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 `UPL0 = 'L'` (default), only the lower triangular part of the matrix is used in the computation.
- If `UPL0 = '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**

- `linalg_eigh()` computes the full eigenvalue decomposition.

Other linalg: `linalg_cholesky()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

```
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  $\cdot$  or  $\cdot^*$ , for a matrix  $\cdot$  with columns  $\cdot$  with  $\cdot$  and a vector  $\cdot$  with  $\cdot$ , this function computes the first  $\cdot$  columns of the matrix

**Usage**

```
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. Please find it in '<https://torch.mlverse.org/docs>'

where  $\cdot$  is the  $m$ -dimensional identity matrix and  $\cdot$  is the conjugate transpose when  $\cdot$  is complex, and the transpose when  $\cdot$  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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

```
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  $\mathbf{A}$  be  $\mathbf{A}$  or  $\mathbf{A}$ , for a matrix  $\mathbf{A}$ , its **inverse matrix**  $\mathbf{A}^{-1}$  (if it exists) is defined as

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

where  $\mathbf{I}$  is the  $n$ -dimensional identity matrix.

The inverse matrix exists if and only if  $\mathbf{A}$  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  $\mathbf{A}$  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

`linalg_pinv()` computes the pseudoinverse (Moore-Penrose inverse) of matrices of any shape. `linalg_solve()` computes `A$inv() %*% B` with a numerically stable algorithm.

Other linalg: `linalg_cholesky()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

## Examples

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

---

`linalg_inv_ex`

*Computes the inverse of a square matrix if it is invertible.*

---

## Description

Returns a namedtuple (`inverse`, `info`). `inverse` contains the result of inverting  $\mathbf{A}$  and `info` stores the LAPACK error codes. If  $\mathbf{A}$  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 `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. `info` filled with zeros indicates that the inversion was successful. If `check_errors=TRUE` and `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  $\mathbf{A}$  is a batch of matrices then the output has the same batch dimensions.

## Usage

```
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 info. Default: 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\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

**Examples**

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

---

linalg_lstsq	<i>Computes a solution to the least squares problem of a system of linear equations.</i>
--------------	--

---

**Description**

Letting  $.$  be  $.$  or  $.$ , the **least squares problem** for a linear system  $.$  with  $.$  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.

... currently 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. Please find it in '<https://torch.mlverse.org/docs>'

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. driver chooses the LAPACK/MAGMA function that will be used.

For CPU inputs the valid values are 'gels', 'gelsy', 'gelsd', 'gelss'. For CUDA input, the only valid driver is '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: 'gelsy' (QR with pivoting) (default)
- If A is full-rank: 'gels' (QR)
- If A is not well-conditioned.
- 'gelsd' (tridiagonal reduction and SVD)
- But if you run into memory issues: 'gelss' (full SVD).

See also the **full description of these drivers**

rcond is used to determine the effective rank of the matrices in A when driver is one of ('gelsy', 'gelsd', 'gelss'). In this case, if  $\cdot$  are the singular values of A in decreasing order,  $\cdot$  will be rounded down to zero if  $\cdot$ . If rcond = NULL (default), 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 (solution, residuals, rank, singular\_values). For inputs A, B of shape (\*, m, n), (\*, m, k) respectively, it contains

- solution: the least squares solution. It has shape (\*, n, k).
- 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.
- rank: tensor of ranks of the matrices in A. It has shape equal to the batch dimensions of A. It is computed when driver is one of ('gelsy', 'gelsd', 'gelss'), otherwise it is an empty tensor.
- singular\_values: tensor of singular values of the matrices in A. It has shape (\*, min(m, n)). It is computed when driver is one of ('gelsd', 'gelss'), otherwise it is an empty tensor.

### Value

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

**Warning**

The default value of `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  $X = A \text{pinverse}() \%*\% B$  in a faster and more numerically stable way than performing the computations separately.

**See Also**

Other linalg: `linalg_cholesky()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

```
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**

```
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.
ord	(int, inf, -inf, 'fro', 'nuc', optional): order of norm. Default: 'fro'
dim	(int, Tuple[int], 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:

ord	norm for matrices	norm for vectors
NULL (default)	Frobenius norm	2-norm (see below)
"fro"	Frobenius norm	– not supported –
"nuc"	nuclear norm	– not supported –
Inf	max(sum(abs(x), dim=2))	max(abs(x))
-Inf	min(sum(abs(x), dim=2))	min(abs(x))
0	– not supported –	sum(x != 0)
1	max(sum(abs(x), dim=1))	as below
-1	min(sum(abs(x), dim=1))	as below
2	largest singular value	as below
-2	smallest singular value	as below
other int or float	– not supported –	sum(abs(x)^{ord})^{(1 / ord)}

**See Also**

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_power\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_norm\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_qr\(\)](#), [linalg\\_slogdet\(\)](#), [linalg\\_solve\(\)](#), [linalg\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [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  $A^{-1}B$  with a numerically stable algorithm.

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_norm\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_norm\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_qr\(\)](#), [linalg\\_slogdet\(\)](#), [linalg\\_solve\(\)](#), [linalg\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [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

<code>A</code>	(Tensor): tensor of shape $(*, m, n)$ where $*$ is zero or more batch dimensions.
<code>...</code>	Not currently used.
<code>atol</code>	the absolute tolerance value. When NULL it's considered to be zero.
<code>rtol</code>	the relative tolerance value. See above for the value it takes when NULL.
<code>tol</code>	(float, Tensor, optional): the tolerance value. See above for the value it takes when NULL. Default: NULL.
<code>hermitian</code>	(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

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

where  $\cdot$  is the largest singular value (or eigenvalue in absolute value when `hermitian = TRUE`), and  $\epsilon$  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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `linalg_lstsq()`, `linalg_matrix_norm()`, `linalg_matrix_power()`, `linalg_multi_dot()`, `linalg_norm()`, `linalg_pinv()`, `linalg_qr()`, `linalg_slogdet()`, `linalg_solve()`, `linalg_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

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

---

<code>linalg_multi_dot</code>	<i>Efficiently multiplies two or more matrices</i>
-------------------------------	--

---

**Description**

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

**Usage**

```
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  $a * b * c$ . 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. Please find it in '<https://torch.mlverse.org/docs>'

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

**See Also**

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

**Examples**

```
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
ord	(int, float, inf, -inf, 'fro', 'nuc', optional): order of norm. Default: NULL
dim	(int, Tuple[int], 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:

ord	norm for matrices	norm for vectors
NULL (default)	Frobenius norm	2-norm (see below)
"fro"	Frobenius norm	– not supported –
"nuc"	nuclear norm	– not supported –
Inf	max(sum(abs(x), dim=2))	max(abs(x))
-Inf	min(sum(abs(x), dim=2))	min(abs(x))
0	– not supported –	sum(x != 0)
1	max(sum(abs(x), dim=1))	as below
-1	min(sum(abs(x), dim=1))	as below
2	largest singular value	as below
-2	smallest singular value	as below
other int or float	– not supported –	sum(abs(x)^{ord})^{(1 / ord)}

### See Also

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_norm\(\)](#), [linalg\\_matrix\\_power\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_qr\(\)](#), [linalg\\_slogdet\(\)](#), [linalg\\_solve\(\)](#), [linalg\\_solve\\_triangular](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

### Examples

```
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

```
linalg_pinv(A, rcond = NULL, hermitian = FALSE, atol = NULL, rtol = NULL)
```

### Arguments

A	(Tensor): tensor of shape (*, m, n) where * is zero or more batch dimensions.
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 <a href="#">linalg_svd()</a> . Alias for rtol. Default: 0.
hermitian	(bool, optional): indicates whether A is Hermitian if complex or symmetric if real. Default: FALSE.
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.

### Details

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. The singular values (or the norm of the eigenvalues when hermitian= TRUE) that are below the specified rcond threshold are treated as zero and discarded in the computation.

### Note

This function uses [linalg\\_svd\(\)](#) if hermitian= FALSE and [linalg\\_eigh\(\)](#) if hermitian= TRUE. For CUDA inputs, this function synchronizes that device with the CPU.

Consider using [linalg\\_lstsq\(\)](#) if possible for multiplying a matrix on the left by the pseudoinverse, as `linalg_lstsq(A, B)$solution == A$pinv() %*% B`

It is always preferred to use [linalg\\_lstsq\(\)](#) when possible, as it is faster and more numerically stable than computing the pseudoinverse explicitly.

**See Also**

- `linalg_inv()` computes the inverse of a square matrix.
- `linalg_lstsq()` computes  $A \text{pinv}() \%*\% B$  with a numerically stable algorithm.

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

**Examples**

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

---

`linalg_qr`

*Computes the QR decomposition of a matrix.*

---

**Description**

Letting  $Q$  be  $m \times m$  or  $n \times n$ , the **full QR decomposition** of a matrix  $A$  is defined as

**Usage**

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

**Arguments**

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

`mode` (str, optional): one of 'reduced', 'complete', 'r'. Controls the shape of the returned tensors. Default: 'reduced'.

**Details**

Equation not displayed. Please find it in '<https://torch.mlverse.org/docs>'

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. Please find it in '<https://torch.mlverse.org/docs>'

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 `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\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_norm\(\)](#), [linalg\\_matrix\\_power\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_norm\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_slogdet\(\)](#), [linalg\\_solve\(\)](#), [linalg\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [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	<i>Computes the sign and natural logarithm of the absolute value of the determinant of a square matrix.</i>
----------------	---

---

### 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  $\text{sign} * \exp(\text{logabsdet})$ .
- When a matrix has a determinant of zero, it returns  $(0, -\text{Inf})$ .

**See Also**

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_norm\(\)](#), [linalg\\_matrix\\_power\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_norm\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_qr\(\)](#), [linalg\\_solve\(\)](#), [linalg\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [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  $\cdot$  be  $\cdot$  or  $\cdot$ , this function computes the solution  $\cdot$  of the **linear system** associated to  $\cdot$ , 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\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [linalg\\_lstsq\(\)](#), [linalg\\_matrix\\_norm\(\)](#), [linalg\\_matrix\\_power\(\)](#), [linalg\\_matrix\\_rank\(\)](#), [linalg\\_multi\\_dot\(\)](#), [linalg\\_norm\(\)](#), [linalg\\_pinv\(\)](#), [linalg\\_qr\(\)](#), [linalg\\_slogdet\(\)](#), [linalg\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

**Examples**

```
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_solve_triangular`

*Triangular solve*

---

**Description**

Triangular solve



**Usage**

```
linalg_solve_triangular(A, B, ..., upper, left = TRUE, unitriangular = FALSE)
```

**Arguments**

A tensor of shape  $(*, n, n)$  or  $(*, k, k)$  if `left=TRUE` where  $*$  is zero or more batch dimensions.

B right-hand side tensor of shape  $(*, n, k)$

... Currently ignored.

upper whether A is an upper or lower triangular matrix.

left wheter to solve the system  $AX=B$  or  $XA=B$

unitriangular if TRUE, the diagonal elements of A are assumed to be all equal to 1.

**See Also**

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

---

linalg\_svd

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


---

**Description**

Letting  $.$  be  $.$  or  $.$ , the **full SVD** of a matrix  $.$ , 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. Please find it in '<https://torch.mlverse.org/docs>'

where  $\cdot$ ,  $\cdot$  is the conjugate transpose when  $\cdot$  is complex, and the transpose when  $\cdot$  is real-valued.

The matrices  $\cdot$ ,  $\cdot$  (and thus  $\cdot$ ) 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. Please find it in '<https://torch.mlverse.org/docs>'

where  $\cdot$ ,  $\cdot$

In this case,  $\cdot$  and  $\cdot$  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  $\cdot, \cdot, \cdot$  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  $\cdot, \cdot, \cdot$  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  $\cdot$  by  $-1$  in the real case or by  $\cdot$  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  $\cdot$  through the computation of  $\cdot$ . The gradient will also be numerically unstable when  $A$  has small singular values, as it also depends on the computation of  $\cdot$ .

**Note**

When `full_matrices=TRUE`, the gradients with respect to  $U[\dots, :, \min(m, n):]$  and  $V[\dots, \min(m, n):, :]$  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 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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

## Examples

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

---

<code>linalg_svdvals</code>	<i>Computes the singular values of a matrix.</i>
-----------------------------	--

---

## 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  $(*, m, n)$  where  $*$  is zero or more batch dimensions.

## Value

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

**See Also**

[linalg\\_svd\(\)](#) computes the full singular value decomposition.

Other linalg: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#), [linalg\\_vector\\_norm\(\)](#)

**Examples**

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

---

<code>linalg_tensorinv</code>	<i>Computes the multiplicative inverse of <code>torch_tensordot()</code></i>
-------------------------------	--

---

**Description**

If  $m$  is the product of the first `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 `torch_tensordot(A, X, ind)` is the identity matrix in dimension  $m$ .

**Usage**

```
linalg_tensorinv(A, ind = 3L)
```

**Arguments**

<code>A</code>	(Tensor): tensor to invert.
<code>ind</code>	(int): index at which to compute the inverse of <code>torch_tensordot()</code> . Default: 3.

**Details**

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

**Note**

Consider using [linalg\\_tensorsolve\(\)](#) if possible for multiplying a tensor on the left by the tensor inverse as `linalg_tensorsolve(A, B) == torch_tensordot(linalg_tensorinv(A), B)`

It is always preferred to use [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()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorsolve()`, `linalg_vector_norm()`

**Examples**

```
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 `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 `tensor_dot(A, x, dims=x$.ndim) == B`.

**Usage**

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

**Arguments**

<code>A</code>	(Tensor): tensor to solve for.
<code>B</code>	(Tensor): the solution
<code>dims</code>	(Tuple[int], optional): dimensions of $A$ to be moved. If <code>NULL</code> , no dimensions are moved. Default: <code>NULL</code> .

**Details**

If `dims` is specified,  $A$  will be reshaped as `A = movedim(A, dims, seq(len(dims) - A$.ndim + 1, 0))`

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

**See Also**

- `linalg_tensorinv()` computes the multiplicative inverse of `torch_tensordot()`.

Other linalg: `linalg_cholesky()`, `linalg_cholesky_ex()`, `linalg_det()`, `linalg_eig()`, `linalg_eigh()`, `linalg_eigvals()`, `linalg_eigvalsh()`, `linalg_householder_product()`, `linalg_inv()`, `linalg_inv_ex()`, `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_solve_triangular()`, `linalg_svd()`, `linalg_svdvals()`, `linalg_tensorinv()`, `linalg_vector_norm()`

**Examples**

```
if (torch_is_installed()) {
A <- torch_eye(2 * 3 * 4)$reshape(c(2 * 3, 4, 2, 3, 4))
B <- torch_randn(2 * 3, 4)
X <- linalg_tensorsolve(A, B)
X$shape
torch_allclose(torch_tensordot(A, X, dims = X$ndim), B)

A <- torch_randn(6, 4, 4, 3, 2)
B <- torch_randn(4, 3, 2)
X <- linalg_tensorsolve(A, B, dims = c(1, 3))
A <- A$permute(c(2, 4, 5, 1, 3))
torch_allclose(torch_tensordot(A, X, dims = X$ndim), B, atol = 1e-6)
}
```

---

`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 <code>dim</code> .
ord	(int, float, inf, -inf, 'fro', 'nuc', optional): order of norm. Default: 2
dim	(int, Tuple[int], optional): dimensions over which to compute the vector or matrix norm. See above for the behavior when <code>dim=NULL</code> . 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=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:

<code>ord</code>	norm for matrices	norm for vectors
NULL (default)	Frobenius norm	2-norm (see below)
"fro"	Frobenius norm	– not supported –
"nuc"	nuclear norm	– not supported –
Inf	$\max(\text{sum}(\text{abs}(x), \text{dim}=2))$	$\max(\text{abs}(x))$
-Inf	$\min(\text{sum}(\text{abs}(x), \text{dim}=2))$	$\min(\text{abs}(x))$
0	– not supported –	$\text{sum}(x \neq 0)$
1	$\max(\text{sum}(\text{abs}(x), \text{dim}=1))$	as below
-1	$\min(\text{sum}(\text{abs}(x), \text{dim}=1))$	as below
2	largest singular value	as below
-2	smallest singular value	as below
other int or float	– not supported –	$\text{sum}(\text{abs}(x)^{\{\text{ord}\}})^{\{(1 / \text{ord})\}}$

### See Also

Other `linalg`: [linalg\\_cholesky\(\)](#), [linalg\\_cholesky\\_ex\(\)](#), [linalg\\_det\(\)](#), [linalg\\_eig\(\)](#), [linalg\\_eigh\(\)](#), [linalg\\_eigvals\(\)](#), [linalg\\_eigvalsh\(\)](#), [linalg\\_householder\\_product\(\)](#), [linalg\\_inv\(\)](#), [linalg\\_inv\\_ex\(\)](#), [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\\_solve\\_triangular\(\)](#), [linalg\\_svd\(\)](#), [linalg\\_svdvals\(\)](#), [linalg\\_tensorinv\(\)](#), [linalg\\_tensorsolve\(\)](#)

### Examples

```
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	<i>Load a state dict file</i>
-----------------	-------------------------------

---

### 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	<i>Autocast context manager</i>
----------------	---------------------------------

---

### 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
)

set_autocast(device_type, dtype = NULL, enabled = TRUE, cache_enabled = NULL)

unset_autocast(context)

```

### Arguments

<code>device_type</code>	a character string indicating whether to use 'cuda' or 'cpu' device
<code>dtype</code>	a torch data type indicating whether to use <code>torch_float16()</code> or <code>torch_bfloat16()</code> .
<code>enabled</code>	a logical value indicating whether autocasting should be enabled in the region. Default: TRUE
<code>cache_enabled</code>	a logical value indicating whether the weight cache inside autocast should be enabled.
<code>...</code>	currently unused.
<code>.env</code>	The environment to use for scoping.
<code>code</code>	code to be executed with no gradient recording.
<code>context</code>	Returned by <code>set_autocast</code> and should be passed when unsetting it.

### 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.
- `set_autocast()`: Set the autocast context. For advanced users only.
- `unset_autocast()`: Unset the autocast context.

### See Also

[cuda\\_amp\\_grad\\_scaler\(\)](#) to perform dynamic gradient scaling.

## Examples

```
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)  
    w  
  }  
  
  out <- foo(x, y)  
}
```

---

local_device	<i>Device contexts</i>
--------------	------------------------

---

## Description

Device contexts

## Usage

```
local_device(device, ..., .env = parent.frame())
```

```
with_device(code, ..., device)
```

## Arguments

device	A torch device to be used by default when creating new tensors.
...	currently unused.
.env	The environment to use for scoping.
code	The code to be evaluated in the modified environment.

## Functions

- `with_device()`: Modifies the default device for the selected context.

---

lr_cosine_annealing	<i>Set the learning rate of each parameter group using a cosine annealing schedule</i>
---------------------	--

---

**Description**

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

**Usage**

```
lr_cosine_annealing(
    optimizer,
    T_max,
    eta_min = 0,
    last_epoch = -1,
    verbose = FALSE
)
```

**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	<i>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.</i>
-----------	--

---

**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	<i>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.</i>
-------------------	---

---

**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**

```
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**

```

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	<i>Once cycle learning rate</i>
--------------	---------------------------------

---

### 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 $\text{initial\_lr} = \text{max\_lr} / \text{div\_factor}$ Default: 25
final_div_factor	(float): Determines the minimum learning rate via $\text{min\_lr} = \text{initial\_lr} / \text{final\_div\_factor}$ 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 <i>batches</i> 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  $\text{total\_steps} = \text{epochs} * \text{steps\_per\_epoch}$

You must either provide a value for total\_steps or provide a value for both epochs and steps\_per\_epoch.

## Examples

```
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)  
}
```

---

lr\_reduce\_on\_plateau *Reduce learning rate on plateau*

---

## 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

```
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. $\text{new\_lr} <- \text{lr} * \text{factor}$ . 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, $\text{dynamic\_threshold} <- \text{best} * (1 + \text{threshold})$ in 'max' mode or $\text{best} * (1 - \text{threshold})$ in min mode. In abs mode, $\text{dynamic\_threshold} <- \text{best} + \text{threshold}$ in max mode or $\text{best} - \text{threshold}$ in min mode. Default: 'rel'.
cooldown	(int): Number of epochs to wait before resuming normal operation after lr has been reduced. Default: 0.
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	<i>Creates learning rate schedulers</i>
--------------	---

---

**Description**

Creates learning rate schedulers

**Usage**

```
lr_scheduler(
  classname = NULL,
  inherit = LRScheduler,
  ...,
  parent_env = parent.frame()
)
```

**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 <code>get_lr()</code> method that doesn't take any argument and returns learning rates for each <code>param_group</code> in the optimizer.
parent_env	passed to <code>R6::R6Class()</code> .

---

lr_step	<i>Step learning rate decay</i>
---------	---------------------------------

---

**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	<i>Affine_grid</i>
-----------------	--------------------

---

**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 <a href="#">nnf_grid_sample()</a> for a more complete description. A grid generated by <a href="#">nnf_affine_grid()</a> should be passed to <a href="#">nnf_grid_sample()</a> 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	<i>Alpha_dropout</i>
-------------------	----------------------

---

**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	<i>Avg_pool1d</i>
----------------	-------------------

---

**Description**

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

**Usage**

```
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	<i>Avg_pool2d</i>
----------------	-------------------

---

**Description**

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

**Usage**

```
nnf_avg_pool2d(
    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	<i>Avg_pool3d</i>
----------------	-------------------

---

**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  $\lfloor \frac{\text{input planes}}{sT} \rfloor$ .

**Usage**

```
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

---

nnf_batch_norm	<i>Batch_norm</i>
----------------	-------------------

---

**Description**

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

**Usage**

```
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



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")
)
```

**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'

---

```
nnf_binary_cross_entropy_with_logits
    Binary_cross_entropy_with_logits
```

---

**Description**

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

**Usage**

```
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_celu	<i>Celu</i>
----------	-------------

---

**Description**

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

**Usage**

```
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	<i>Sparsemax</i>
-----------------------	------------------

---

**Description**

Applies the SparseMax activation.

**Usage**

```
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

*Conv1d*


---

### Description

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

### Usage

```
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

*Conv2d*


---

### Description

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

**Usage**

```
nnf_conv2d(
    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**

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

**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

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

### Arguments

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

---

 nnf\_conv\_transpose2d *Conv\_transpose2d*


---

### Description

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

### Usage

```
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 *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

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
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 (dT, dH, dW). Default: 1

---

```
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**

```
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**

```
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**

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

---

nnf_cross_entropy	<i>Cross_entropy</i>
-------------------	----------------------

---

**Description**

This criterion combines `log_softmax` and `nll_loss` in a single function.

**Usage**

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

**Arguments**

input	(Tensor) $(N, C)$ where $C$ = number of classes or $(N, C, H, W)$ in case of 2D Loss, or $(N, C, d_1, d_2, \dots, d_K)$ 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 $(N, d_1, d_2, \dots, d_K)$ 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	<i>Ctc_loss</i>
--------------	-----------------

---

### 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 <a href="#">nnf_log_softmax</a> ).
targets	( $N, S$ ) or ( $\text{sum}(\text{target\_lengths})$ ). 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 $\leq T$ )
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'   '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'
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	<i>Dropout</i>
-------------	----------------

---

**Description**

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

**Usage**

```
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	<i>Dropout2d</i>
---------------	------------------

---

**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**

```
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	<i>Dropout3d</i>
---------------	------------------

---

### 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	<i>Elu</i>
---------	------------

---

### Description

Applies element-wise,

$$ELU(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$$

### 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**

```

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

*Embedding***Description**

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

**Usage**

```

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.

---

nnf_embedding_bag	<i>Embedding_bag</i>
-------------------	----------------------

---

**Description**

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

**Usage**

```
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".



mode	(string, optional) "sum", "mean" or "max". Specifies the way to reduce the bag. Default: 'mean'
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. Note: this option is not supported when mode="max".
per_sample_weights	(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.
include_last_offset	(bool, optional) if TRUE, the size of offsets is equal to the number of bags + 1.
padding_idx	(int, optional) If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.

---

nnf\_fold

*Fold*


---

### Description

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

### Usage

```
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\$size()[ $-c(1,2)$ ])
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.

---

```
nnf_fractional_max_pool2d
    Fractional_max_pool2d
```

---

**Description**

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

**Usage**

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

**Arguments**

<code>input</code>	the input tensor
<code>kernel_size</code>	the size of the window to take a max over. Can be a single number $k$ (for a square kernel of $k * k$ ) or a tuple ( $kH$ , $kW$ )
<code>output_size</code>	the target output size of the image of the form $oH * oW$ . Can be a tuple ( $oH$ , $oW$ ) or a single number $oH$ for a square image $oH * oH$
<code>output_ratio</code>	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)
<code>return_indices</code>	if True, will return the indices along with the outputs.
<code>random_samples</code>	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  $kH * kW$  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**

```
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 $k * k * k$ ) or a tuple ( $kT$ , $kH$ , $kW$ )
output_size	the target output size of the form $oT * oH * oW$ . Can be a tuple ( $oT$ , $oH$ , $oW$ ) or a single number $oH$ for a cubic output $oH * oH * oH$
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	undocumented argument.

**Details**

Fractional MaxPooling is described in detail in the paper [Fractional MaxPooling\\_](#) by Ben Graham

The max-pooling operation is applied in  $kT * kH * kW$  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

*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  $x * \text{pnorm}(x)$ , if 'tanh', then GELU is estimated. See [GELU](#) for more info.

**gelu(input) -> Tensor**

Applies element-wise the function  $GELU(x) = x * \Phi(x)$   
 where  $\Phi(x)$  is the Cumulative Distribution Function for Gaussian Distribution.  
 See [Gaussian Error Linear Units \(GELUs\)](#).

nnf\_glu

*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**

$$GLU(a, b) = a \otimes \sigma(b)$$

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	<i>Grid_sample</i>
-----------------	--------------------

---

### Description

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

### Usage

```
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 <a href="#">nnf_interpolate()</a> , 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	<i>Group_norm</i>
----------------	-------------------

---

### Description

Applies Group Normalization for last certain number of dimensions.

### Usage

```
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	<i>Gumbel_softmax</i>
--------------------	-----------------------

---

**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	<i>Hardshrink</i>
----------------	-------------------

---

**Description**

Applies the hard shrinkage function element-wise

**Usage**

```
nnf_hardshrink(input, lambda = 0.5)
```

**Arguments**

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

---

nnf\_hardsigmoid      *Hardsigmoid*

---

### Description

Applies the element-wise function  $\text{Hardsigmoid}(x) = \frac{\text{ReLU6}(x+3)}{6}$

### 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          *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

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



---

nnf_hardtanh	<i>Hardtanh</i>
--------------	-----------------

---

**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	<i>Hinge_embedding_loss</i>
--------------------------	-----------------------------

---

**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

```
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**

```
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 <i>independent</i> 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 v

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**

```
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**

```
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	<i>Layer_norm</i>
----------------	-------------------

---

**Description**

Applies Layer Normalization for last certain number of dimensions.

**Usage**

```
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	<i>Leaky_relu</i>
----------------	-------------------

---

**Description**

Applies element-wise,  $LeakyReLU(x) = \max(0, x) + negative\_slope * \min(0, x)$

**Usage**

```
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	<i>Linear</i>
------------	---------------

---

**Description**

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

**Usage**

```
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	<i>Local_response_norm</i>
-------------------------	----------------------------

---

**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**

```
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_logsigmoid	<i>Logsigmoid</i>
----------------	-------------------

---

**Description**

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

**Usage**

```
nnf_logsigmoid(input)
```

**Arguments**

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

---

nnf_log_softmax	<i>Log_softmax</i>
-----------------	--------------------

---

**Description**

Applies a softmax followed by a logarithm.

**Usage**

```
nnf_log_softmax(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  $\log(\text{softmax}(x))$ , 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	<i>Lp_pool1d</i>
---------------	------------------

---

**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	<i>Lp_pool2d</i>
---------------	------------------

---

**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  $x_1$ ,  $x_2$ , two 1D mini-batch Tensors, and a label 1D mini-batch tensor  $y$  (containing 1 or -1).

**Usage**

```
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 00 .
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**

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

**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
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 to compute the output shape. Default: FALSE
return_indices	whether to return the indices where the max occurs.

---

nnf_max_pool2d	<i>Max_pool2d</i>
----------------	-------------------

---

**Description**

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

**Usage**

```
nnf_max_pool2d(
    input,
    kernel_size,
    stride = kernel_size,
    padding = 0,
    dilation = 1,
    ceil_mode = FALSE,
    return_indices = FALSE
)
```

**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
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. Default: FALSE
return_indices	whether to return the indices where the max occurs.

---

nnf_max_pool3d	<i>Max_pool3d</i>
----------------	-------------------

---

**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	<i>Max_unpool1d</i>
------------------	---------------------

---

**Description**

Computes a partial inverse of MaxPool1d.

**Usage**

```
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

---

nnf_max_unpool2d	<i>Max_unpool2d</i>
------------------	---------------------

---

**Description**

Computes a partial inverse of MaxPool2d.

**Usage**

```
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	<i>Max_unpool3d</i>
------------------	---------------------

---

**Description**

Computes a partial inverse of MaxPool3d.

**Usage**

```
nnf_max_unpool3d(
    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_mse_loss	<i>Mse_loss</i>
--------------	-----------------

---

**Description**

Measures the element-wise mean squared error.

**Usage**

```
nnf_mse_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\_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

```
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

```
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**

```
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 = None,  
    need_weights = True,  
    attn_mask = None,  
    avg_weights = True,  
    use_separate_proj_weight = False,  
    q_proj_weight = None,  
    k_proj_weight = None,
```

```

    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_heads	parallel attention heads.
in_proj_weight	input projection weight.
in_proj_bias	input projection bias.
bias_k	bias of the key and value sequences to be added at dim=0.
bias_v	currently undocumented.
add_zero_attn	add a new batch of zeros to the key and value sequences at dim=1.
dropout_p	probability of an element to be zeroed.
out_proj_weight	the output projection weight.
out_proj_bias	output projection bias.
training	apply 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_weights	output attn_output_weights.
attn_mask	2D 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.



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 accept 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,  $0 \leq y \leq x.size(2) - 1$ ).

### Usage

```
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.

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\_nll\_loss

*Nll\_loss*


---

### Description

The negative log likelihood loss.

### Usage

```
nnf_nll_loss(
    input,
    target,
    weight = NULL,
    ignore_index = -100,
    reduction = "mean"
)
```

### Arguments

input ( $N, C$ ) where  $C$  = number of classes or  $(N, C, H, W)$  in case of 2D Loss, or  $(N, C, d_1, d_2, \dots, d_K)$  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  $(N, d_1, d_2, \dots, d_K)$  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	<i>Normalize</i>
---------------	------------------

---

**Description**

Performs  $L_p$  normalization of inputs over specified dimension.

**Usage**

```
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  $(n_0, \dots, n_{dim}, \dots, n_k)$ , 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	<i>One_hot</i>
-------------	----------------

---

**Description**

Takes LongTensor with index values of shape  $(*)$  and returns a tensor of shape  $(*, \text{num\_classes})$  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**

```
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	<i>Pad</i>
---------	------------

---

**Description**

Pads tensor.

**Usage**

```
nnf_pad(input, pad, mode = "constant", value = NULL)
```

**Arguments**

input	(Tensor) N-dimensional tensor
pad	(tuple) m-elements tuple, where $\frac{m}{2} \leq$ 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.  $\lfloor \frac{\text{len}(\text{pad})}{2} \rfloor$  dimensions of input will be padded. For example, to pad only the last dimension of the input tensor, then pad has the form (padding\_left, padding\_right); to pad the last 2 dimensions of the input tensor, then use (padding\_left, padding\_right, padding\_top, padding\_bottom); to pad the last 3 dimensions, use (padding\_left, padding\_right, padding\_top, padding\_bottom, padding\_front, 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 *Pairwise\_distance*


---

**Description**

Computes the batchwise pairwise distance between vectors using the p-norm.

**Usage**

```
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 *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**

```
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	<i>Pixel_shuffle</i>
-------------------	----------------------

---

**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**

```
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	<i>Poisson_nll_loss</i>
----------------------	-------------------------

---

**Description**

Poisson negative log likelihood loss.

**Usage**

```
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  $\exp(\text{input}) - \text{target} * \text{input}$ , if FALSE then loss is  $\text{input} - \text{target} * \log(\text{input} + \text{eps})$ . 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  $\text{log\_input}=\text{FALSE}$ . 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. Default: 'mean'

---

 nnf\_prelu

*Prelu*


---

### Description

Applies element-wise the function  $PReLU(x) = \max(0, x) + weight * \min(0, x)$  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

---

 nnf\_relu6

*Relu6*


---

**Description**

Applies the element-wise function  $ReLU6(x) = \min(\max(0, x), 6)$ .

**Usage**

```
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

*Rrelu*


---

**Description**

Randomized leaky ReLU.

**Usage**

```
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	<i>Selu</i>
----------	-------------

---

**Description**

Applies element-wise,

$$SELU(x) = scale * (max(0, x) + min(0, \alpha * (exp(x) - 1)))$$

, with  $\alpha = 1.6732632423543772848170429916717$  and  $scale = 1.0507009873554804934193349852946$ .

**Usage**

```
nnf_selu(input, inplace = FALSE)
```

```
nnf_selu_(input)
```

**Arguments**

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

inplace can optionally do the operation in-place. Default: FALSE

**Examples**

```
if (torch_is_installed()) {
  x <- torch_randn(2, 2)
  y <- nnf_selu(x)
  nnf_selu_(x)
  torch_equal(x, y)
}
```

---

nnf_sigmoid	<i>Sigmoid</i>
-------------	----------------

---

**Description**

Applies element-wise  $Sigmoid(x_i) = \frac{1}{1+exp(-x_i)}$

**Usage**

```
nnf_sigmoid(input)
```

**Arguments**

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

---

nnf_silu	<i>Applies the Sigmoid Linear Unit (SiLU) function, element-wise. See <a href="#">nn_silu()</a> for more information.</i>
----------	---

---

**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	<i>Smooth_l1_loss</i>
--------------------	-----------------------

---

**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	<i>Softmax</i>
-------------	----------------

---

**Description**

Applies a softmax function.

**Usage**

```
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:

$$\text{Softmax}(x_i) = \exp(x_i) / \sum_j \exp(x_j)$$

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	<i>Softmin</i>
-------------	----------------

---

**Description**

Applies a softmin function.

**Usage**

```
nnf_softmin(input, dim, dtype = NULL)
```

**Arguments**

input	(Tensor) input
dim	(int) A dimension along which softmin will be computed (so every slice along dim will sum to 1).
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**

Note that

$$\text{Softmin}(x) = \text{Softmax}(-x)$$

.

See [nnf\\_softmax](#) definition for mathematical formula.

---

 nnf\_softplus

*Softplus*


---

**Description**

Applies element-wise, the function  $\text{Softplus}(x) = 1/\beta * \log(1 + \exp(\beta * x))$ .

**Usage**

```
nnf_softplus(input, beta = 1, threshold = 20)
```

**Arguments**

input	(N,*) tensor, where * means, any number of additional dimensions
beta	the beta value for the Softplus formulation. Default: 1
threshold	values above this revert to a linear function. Default: 20

**Details**

For numerical stability the implementation reverts to the linear function when  $\text{input} * \beta > \text{threshold}$ .

---

nnf_softshrink	<i>Softshrink</i>
----------------	-------------------

---

**Description**

Applies the soft shrinkage function elementwise

**Usage**

```
nnf_softshrink(input, lambda = 0.5)
```

**Arguments**

input	(N,*) tensor, where * means, any number of additional dimensions
lambda	the lambda (must be no less than zero) value for the Softshrink formulation. Default: 0.5

---

nnf_softsign	<i>Softsign</i>
--------------	-----------------

---

**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

```
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,  $Tanhshrink(x) = x - Tanh(x)$

### Usage

```
nnf_tanhshrink(input)
```

### Arguments

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

---

nnf_threshold	<i>Threshold</i>
---------------	------------------

---

**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	<i>Triplet_margin_loss</i>
-------------------------	----------------------------

---

**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"
)
```

**Arguments**

anchor	the anchor input tensor
positive	the positive input tensor
negative	the negative input tensor
margin	Default: 1.
p	The norm degree for pairwise distance. Default: 2.
eps	(float, optional) Small value to avoid division by zero.
swap	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.
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\_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**

anchor	the anchor input tensor
positive	the positive input tensor
negative	the negative input tensor
distance_function	(callable, optional): A nonnegative, real-valued function that quantifies the closeness of two tensors. If not specified, <a href="#">nn_pairwise_distance()</a> will be used. Default: None



margin	Default: 1.
swap	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.
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\_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 a int, or NULL which means the size will be the same as that of the input.

### Examples

```
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  $D \times H \times W$ , 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  $D \times H \times W$ . Can be a tuple (D, H, W) or a single number D for a cube  $D \times D \times D$ . 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

```
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

```
nn_adaptive_log_softmax_with_loss(  
    in_features,  
    n_classes,  
    cutoffs,  
    div_value = 4,  
    head_bias = FALSE  
)
```

## Arguments

<code>in_features</code>	(int): Number of features in the input tensor
<code>n_classes</code>	(int): Number of classes in the dataset
<code>cutoffs</code>	(Sequence): Cutoffs used to assign targets to their buckets
<code>div_value</code>	(float, optional): value used as an exponent to compute sizes of the clusters. Default: 4.0
<code>head_bias</code>	(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.

- `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:  $(N, \text{in\_features})$
- 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)
```

**Arguments**

- `output_size` the target output size of the image of the form  $H \times W$ . Can be a tuple  $(H, W)$  or a single  $H$  for a square image  $H \times H$ .  $H$  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_unpool2d()`.  
Default: `FALSE`

**Examples**

```
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)
}
```

---

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  $D \times H \times W$ , 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  $D \times H \times W$ . Can be a tuple  $(D, H, W)$  or a single  $D$  for a cube  $D \times D \times D$ .  $D$ ,  $H$  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**

```

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	<i>Applies a 1D average pooling over an input signal composed of several input planes.</i>
---------------	--

---

**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:

**Usage**

```

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 ceil instead of floor to compute the output shape
count_include_pad	when TRUE, will include the zero-padding in the averaging calculation

**Details**

$$\text{out}(N_i, C_j, l) = \frac{1}{k} \sum_{m=0}^{k-1} \text{input}(N_i, C_j, \text{stride} \times l + m)$$



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:  $(N, C, L_{in})$
- Output:  $(N, C, L_{out})$ , where

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

### Examples

```
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	<i>Applies a 2D average pooling over an input signal composed of several input planes.</i>
---------------	--

---

### Description

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:

### Usage

```
nn_avg_pool2d(
  kernel_size,
  stride = NULL,
  padding = 0,
  ceil_mode = FALSE,
  count_include_pad = TRUE,
  divisor_override = NULL
)
```

### Arguments

<code>kernel_size</code>	the size of the window
<code>stride</code>	the stride of the window. Default value is <code>kernel_size</code>
<code>padding</code>	implicit zero padding to be added on both 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

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

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

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding[0] - kernel\_size[0]}{stride[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times padding[1] - kernel\_size[1]}{stride[1]} + 1 \right\rfloor$$

## Examples

```

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	<i>Applies a 3D average pooling over an input signal composed of several input planes.</i>
---------------	--

---

### 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_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

$$\text{out}(N_i, C_j, d, h, w) = \frac{\sum_{k=0}^{kD-1} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times d + k, \text{stride}[1] \times h + m, \text{stride}[2] \times w + n)}{kD \times kH \times kW}$$

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

**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**

```
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**

```
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 running_mean and running_var computation. Can be set to NULL 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 always uses batch statistics in both training and eval modes. Default: TRUE

**Details**

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

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.

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 :attr: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.

**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  $(N, L)$  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**

```
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)
```

```
output <- m(input)
}
```

---

nn_batch_norm2d	<i>BatchNorm2D</i>
-----------------	--------------------

---

## 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

```
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

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

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

```
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

```
nn_batch_norm3d(  
  num_features,  
  eps = 1e-05,  
  momentum = 0.1,  
  affine = TRUE,  
  track_running_stats = TRUE  
)
```

**Arguments**

num_features	$C$ from an expected input of size $(N, C, D, H, W)$
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**

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

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:  $(N, C, D, H, W)$
- 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**

```

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**

```
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 nbatch.
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:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n [y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

where  $N$  is the batch size. If reduction is not 'none' (default 'mean'), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets  $y$  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  $\lim_{x \rightarrow 0} \log(x) = -\infty$ .

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  $\lim_{x \rightarrow 0} \frac{d}{dx} \log(x) = \infty$ .

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

```
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 nbatch.
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:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n [y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))],$$

where  $N$  is the batch size. If reduction is not 'none' (default 'mean'), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

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:

$$\ell_c(x, y) = L_c = \{l_{1,c}, \dots, l_{N,c}\}^\top, \quad l_{n,c} = -w_{n,c} [p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))],$$

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**

```
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**

- Input1:  $(N, *, H_{in1})$   $H_{in1} = \text{in1\_features}$  and \* means any number of additional dimensions. All but the last dimension of the inputs should be the same.
- Input2:  $(N, *, H_{in2})$  where  $H_{in2} = \text{in2\_features}$ .
- Output:  $(N, *, H_{out})$  where  $H_{out} = \text{out\_features}$  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**

```
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	<i>Creates a nn_buffer</i>
-----------	----------------------------

---

**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	<i>CELU module</i>
---------	--------------------

---

**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**

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

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**

```

if (torch_is_installed()) {
  m <- nn_celu()
  input <- torch_randn(2)
  output <- m(input)
}

```

---

nn\_contrib\_sparsemax    *Sparsemax activation*

---

**Description**

Sparsemax activation module.

**Usage**

```
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**

```

nn_conv1d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1,
  bias = TRUE,
  padding_mode = "zeros"
)

```

**Arguments**

<code>in_channels</code>	(int): Number of channels in the input image
<code>out_channels</code>	(int): Number of channels produced by the convolution
<code>kernel_size</code>	(int or tuple): Size of the convolving kernel
<code>stride</code>	(int or tuple, optional): Stride of the convolution. Default: 1
<code>padding</code>	(int, tuple or str, optional) – Padding added to both sides of the input. Default: 0
<code>dilation</code>	(int or tuple, optional): Spacing between kernel elements. Default: 1
<code>groups</code>	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
<code>bias</code>	(bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
<code>padding_mode</code>	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

**Details**

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where  $\star$  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  $\left\lfloor \frac{\text{out\_channels}}{\text{in\_channels}} \right\rfloor$ .

**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  $(N, C_{\text{in}}, L_{\text{in}})$ , a depthwise convolution with a depthwise multiplier  $K$ , can be constructed by arguments  $(C_{\text{in}} = C_{\text{in}}, C_{\text{out}} = C_{\text{in}} \times K, \dots, \text{groups} = C_{\text{in}})$ .

**Shape**

- Input:  $(N, C_{in}, L_{in})$
- Output:  $(N, C_{out}, L_{out})$  where

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

**Attributes**

- weight (Tensor): the learnable weights of the module of shape  $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size})$ . The values of these weights are sampled from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{\text{groups}}{C_{in} * \text{kernel\_size}}$
- 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  $k = \frac{\text{groups}}{C_{in} * \text{kernel\_size}}$

**Examples**

```
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**

```
nn_conv2d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
  padding = 0,
  dilation = 1,
  groups = 1,
  bias = TRUE,
  padding_mode = "zeros"
)
```



**Arguments**

<code>in_channels</code>	(int): Number of channels in the input image
<code>out_channels</code>	(int): Number of channels produced by the convolution
<code>kernel_size</code>	(int or tuple): Size of the convolving kernel
<code>stride</code>	(int or tuple, optional): Stride of the convolution. Default: 1
<code>padding</code>	(int or tuple or string, optional): Zero-padding added to both sides of the input. controls the amount of padding applied to 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
<code>dilation</code>	(int or tuple, optional): Spacing between kernel elements. Default: 1
<code>groups</code>	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
<code>bias</code>	(bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
<code>padding_mode</code>	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

**Details**

In the simplest case, the output value of the layer with input size  $(N, C_{\text{in}}, H, W)$  and output  $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$  can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

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:  $\left\lfloor \frac{\text{out\_channels}}{\text{in\_channels}} \right\rfloor$ .

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 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  $(N, C_{in}, H_{in}, W_{in})$ , a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments  $(in\_channels = C_{in}, out\_channels = C_{in} \times K, \dots, groups = C_{in})$ .

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 `backends_cudnn_deterministic = 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  $(out\_channels, \frac{in\_channels}{groups}, kernel\_size[0], kernel\_size[1])$ . The values of these weights are sampled from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{groups}{C_{in} * \prod_{i=0}^1 kernel\_size[i]}$
- `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  $k = \frac{groups}{C_{in} * \prod_{i=0}^1 kernel\_size[i]}$

**Examples**

```
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

*Conv3D module***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:

**Usage**

```
nn_conv3d(
    in_channels,
    out_channels,
    kernel_size,
    stride = 1,
    padding = 0,
    dilation = 1,
    groups = 1,
    bias = TRUE,
    padding_mode = "zeros"
)
```

**Arguments**

<code>in_channels</code>	(int): Number of channels in the input image
<code>out_channels</code>	(int): Number of channels produced by the convolution
<code>kernel_size</code>	(int or tuple): Size of the convolving kernel
<code>stride</code>	(int or tuple, optional): Stride of the convolution. Default: 1
<code>padding</code>	(int, tuple or str, optional): padding added to all six sides of the input. Default: 0
<code>dilation</code>	(int or tuple, optional): Spacing between kernel elements. Default: 1
<code>groups</code>	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
<code>bias</code>	(bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
<code>padding_mode</code>	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

**Details**

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

where  $\star$  is the valid 3D cross-correlation operator

- `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  $\left\lfloor \frac{\text{out\_channels}}{\text{in\_channels}} \right\rfloor$ .

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single int – in which case the same value is used for the 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

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

### Attributes

- `weight` (Tensor): the learnable weights of the module of shape  $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$ . The values of these weights are sampled from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{\text{groups}}{C_{in} * \prod_{i=0}^2 \text{kernel\_size}[i]}$
- `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  $k = \frac{\text{groups}}{C_{in} * \prod_{i=0}^2 \text{kernel\_size}[i]}$

**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  $(N, C_{in}, D_{in}, H_{in}, W_{in})$ , a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments  $(in\_channels = C_{in}, out\_channels = C_{in} \times K, \dots, groups = C_{in})$ .

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**

```
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**

```
nn_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**

<code>in_channels</code>	(int): Number of channels in the input image
<code>out_channels</code>	(int): Number of channels produced by the convolution
<code>kernel_size</code>	(int or tuple): Size of the convolving kernel
<code>stride</code>	(int or tuple, optional): Stride of the convolution. Default: 1
<code>padding</code>	(int or tuple, optional): $\text{dilation} * (\text{kernel\_size} - 1) - \text{padding}$ zero-padding will be added to both sides of the input. Default: 0
<code>output_padding</code>	(int or tuple, optional): Additional size added to one side of the output shape. Default: 0
<code>groups</code>	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
<code>bias</code>	(bool, optional): If True, adds a learnable bias to the output. Default: True
<code>dilation</code>	(int or tuple, optional): Spacing between kernel elements. Default: 1
<code>padding_mode</code>	(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  $\text{dilation} * (\text{kernel\_size} - 1) - \text{padding}$  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  $\left\lfloor \frac{\text{out\_channels}}{\text{in\_channels}} \right\rfloor$ ).

**Shape**

- Input:  $(N, C_{in}, L_{in})$
- Output:  $(N, C_{out}, L_{out})$  where

$$L_{out} = (L_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{dilation} \times (\text{kernel\_size} - 1) + \text{output\_padding} + 1$$

**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  $k = \frac{\text{groups}}{C_{\text{out}} * \text{kernel\_size}}$
- 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  $k = \frac{\text{groups}}{C_{\text{out}} * \text{kernel\_size}}$

**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  $\text{dilation} * (\text{kernel\_size} - 1) - \text{padding}$  amount of zero padding to both sizes of the input. This is set so that when a `~torch.nn.Conv1d` and a `~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`, `~torch.nn.Conv1d` maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `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 `torch.backends.cudnn.deterministic = TRUE`.

**Examples**

```
if (torch_is_installed()) {
  m <- nn_conv_transpose1d(32, 16, 2)
  input <- torch_randn(10, 32, 2)
  output <- m(input)
}
```

---

nn\_conv\_transpose2d    *ConvTranpose2D module*

---

**Description**

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

**Usage**

```
nn_conv_transpose2d(
  in_channels,
  out_channels,
  kernel_size,
  stride = 1,
```

```

padding = 0,
output_padding = 0,
groups = 1,
bias = TRUE,
dilation = 1,
padding_mode = "zeros"
)

```

### Arguments

<code>in_channels</code>	(int): Number of channels in the input image
<code>out_channels</code>	(int): Number of channels produced by the convolution
<code>kernel_size</code>	(int or tuple): Size of the convolving kernel
<code>stride</code>	(int or tuple, optional): Stride of the convolution. Default: 1
<code>padding</code>	(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
<code>output_padding</code>	(int or tuple, optional): Additional size added to one side of each dimension in the output shape. Default: 0
<code>groups</code>	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
<code>bias</code>	(bool, optional): If True, adds a learnable bias to the output. Default: True
<code>dilation</code>	(int or tuple, optional): Spacing between kernel elements. Default: 1
<code>padding_mode</code>	(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 groups= in\_channels, each input channel is convolved with its own set of filters (of size  $\left\lceil \frac{\text{out\_channels}}{\text{in\_channels}} \right\rceil$ ).

The parameters kernel\_size, stride, padding, output\_padding can either be:

- a single int – in which case the same value is used for the height and width dimensions
- 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:  $(N, C_{in}, H_{in}, W_{in})$
- Output:  $(N, C_{out}, H_{out}, W_{out})$  where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

### Attributes

- 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  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{\text{groups}}{C_{out} * \prod_{i=0}^1 \text{kernel\_size}[i]}$
- 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  $k = \frac{\text{groups}}{C_{out} * \prod_{i=0}^1 \text{kernel\_size}[i]}$

### 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  $\text{dilation} * (\text{kernel\_size} - 1) - \text{padding}$  amount of zero padding to both sizes of the input. This is set so that when a `nn_conv2d` and a `nn_conv_transpose2d` are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when  $\text{stride} > 1$ , `nn_conv2d` maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `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 `torch.backends.cudnn.deterministic = TRUE`.

**Examples**

```

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    *ConvTranpose3D module*

---

**Description**

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

**Usage**

```

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

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
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:  $(N, C_{in}, D_{in}, H_{in}, W_{in})$
- Output:  $(N, C_{out}, D_{out}, H_{out}, W_{out})$  where

$$D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{padding}[2] + \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) + \text{output\_padding}[2] + 1$$

**Attributes**

- weight (Tensor): the learnable weights of the module of shape  $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$ . The values of these weights are sampled from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{\text{groups}}{C_{out} * \prod_{i=0}^2 \text{kernel\_size}[i]}$
- 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  $k = \frac{\text{groups}}{C_{out} * \prod_{i=0}^2 \text{kernel\_size}[i]}$

**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  $\text{dilation} * (\text{kernel\_size} - 1) - \text{padding}$  amount of zero padding to both sizes of the input. This is set so that when a `~torch.nn.Conv3d` and a `~torch.nn.ConvTranspose3d` are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when `stride > 1`, `~torch.nn.Conv3d` maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `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 `torch.backends.cudnn.deterministic = True`.

**Examples**

```
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

```
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

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

---

`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

```
nn_cross_entropy_loss(weight = NULL, ignore_index = -100, reduction = "mean")
```

**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 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  $(minibatch, C)$  or  $(minibatch, C, d_1, d_2, \dots, d_K)$  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  $(minibatch, C, d_1, d_2, \dots, d_K)$  with  $K \geq 1$ , where  $K$  is the number of dimensions, and a target of appropriate shape (see below).

**Shape**

- Input:  $(N, C)$  where  $C$  = number of classes, or  $(N, C, d_1, d_2, \dots, d_K)$  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  $(N, d_1, d_2, \dots, d_K)$  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  $(N, d_1, d_2, \dots, d_K)$  with  $K \geq 1$  in the case of K-dimensional loss.

**Examples**

```

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**

```
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  $T$  = input length,  $N$  = batch size, and  $C$  = number of classes (including blank). The logarithmized probabilities of the outputs (e.g. obtained with `[nnf]log_softmax()`).
- **Targets:** Tensor of size  $(N, S)$  or  $(\text{sum}(\text{target\_lengths}))$ , where  $N$  = batch size and  $S$  = 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  $(\text{sum}(\text{target\_lengths}))$  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 represent 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 represent 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 `target_n = 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  $N = \text{batch size}$ .

[nnf]log\_softmax(): R:nnf)log\_softmax() [n,0:s\_n]: 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

A. Graves et al.: Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks: [https://www.cs.toronto.edu/~graves/icml\\_2006.pdf](https://www.cs.toronto.edu/~graves/icml_2006.pdf)

### Examples

```
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)
```



```

loss$backward()

# 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

<code>p</code>	probability of an element to be zeroed. Default: 0.5
<code>inplace</code>	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  $\frac{1}{1-p}$  during training. This means that during evaluation the module simply computes an identity function.

### Shape

- Input: (\*). Input can be of any shape
- Output: (\*). Output is of the same shape as input

### Examples

```
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

```
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**

```

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**

```
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**

```

if (torch_is_installed()) {
  m <- nn_dropout3d(p = 0.2)
  input <- torch_randn(20, 16, 4, 32, 32)
  output <- m(input)
}

```

---

nn_elu	<i>ELU module</i>
--------	-------------------

---

**Description**

Applies the element-wise function:

**Usage**

```
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**

$$\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$$

**Shape**

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
if (torch_is_installed()) {  
  m <- nn_elu()  
  input <- torch_randn(2)  
  output <- m(input)  
}
```

---

nn_embedding	<i>Embedding module</i>
--------------	-------------------------

---

**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  $(\text{num\_embeddings}, \text{embedding\_dim})$  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.

**Examples**

```

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)
}

```

---

nn_embedding_bag	<i>Embedding bag module</i>
------------------	-----------------------------

---

**Description**

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

**Usage**

```

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`.

mode	(string, optional): "sum", "mean" or "max". Specifies the way to reduce the bag. "sum" computes the weighted sum, taking per_sample_weights into consideration. "mean" computes the average of the values in the bag, "max" computes the max value over each bag.
sparse	(bool, optional): If True, gradient w.r.t. weight matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.
include_last_offset	(bool, optional): if True, offsets has one additional element, where the last element is equivalent to the size of indices. This matches the CSR format.
padding_idx	(int, optional): If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.
.weight	(Tensor, optional) embeddings weights (in case you want to set it manually)

### Attributes

- weight (Tensor): the learnable weights of the module of shape (num\_embeddings, embedding\_dim) initialized from  $\mathcal{N}(0, 1)$

### Examples

```

if (torch_is_installed()) {
# an EmbeddingBag module containing 10 tensors of size 3
embedding_sum <- nn_embedding_bag(10, 3, mode = 'sum')
# a batch of 2 samples of 4 indices each
input <- torch_tensor(c(1, 2, 4, 5, 4, 3, 2, 9), dtype = torch_long())
offsets <- torch_tensor(c(0, 4), dtype = torch_long())
embedding_sum(input, offsets)
# example with padding_idx
embedding_sum <- nn_embedding_bag(10, 3, mode = 'sum', padding_idx = 1)
input <- torch_tensor(c(2, 2, 2, 2, 4, 3, 2, 9), dtype = torch_long())
offsets <- torch_tensor(c(0, 4), dtype = torch_long())
embedding_sum(input, offsets)
# An EmbeddingBag can be loaded from an Embedding like so
embedding <- nn_embedding(10, 3, padding_idx = 2)
embedding_sum <- nn_embedding_bag$from_pretrained(embedding$weight,
                                                padding_idx = embedding$padding_idx,
                                                mode='sum')
}

```

---

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)
```

**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**

```
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**

```
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



- 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_unpool2d()`.  
Default: FALSE

### 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

```
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

```
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 ( $k_t \times k_h \times k_w$ )
- 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

**Details**

The max-pooling operation is applied in  $kTxkHxkW$  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**

```
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)
}
```

---

 nn\_gelu

*GELU module*


---

**Description**

Applies the Gaussian Error Linear Units function:

$$\text{GELU}(x) = x * \Phi(x)$$

**Usage**

```
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:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
if (torch_is_installed()) {
  m <- nn_gelu()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

`nn_glu`*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**

```
if (torch_is_installed()) {  
  m <- nn_glu()  
  input <- torch_randn(4, 2)  
  output <- m(input)  
}
```

---

`nn_group_norm`*Group normalization*

---

**Description**

Applies Group Normalization over a mini-batch of inputs as described in the paper [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**

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

The input channels are separated into num\_groups groups, each containing num\_channels / 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**

```
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_norm])
  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	<i>Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.</i>
--------	---

---

### Description

For each element in the input sequence, each layer computes the following function:

### Usage

```
nn_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

$$\begin{aligned}
 r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
 z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
 n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\
 h_t &= (1 - z_t)n_t + z_th_{(t-1)}
 \end{aligned}$$

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.

**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$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  $t = \text{seq\_len}$ . Like *output*, the layers can be separated using `h_n$view(num_layers, num_directions, batch, hidden_size)`.

**Attributes**

- `weight_ih_l[k]`: the learnable input-hidden weights of the  $k^{th}$  layer ( $W_{ih}^l$ ), of shape  $(3 * \text{hidden\_size} \times \text{input\_size})$
- `weight_hh_l[k]`: the learnable hidden-hidden weights of the  $k^{th}$  layer ( $W_{hh}^l$ ), of shape  $(3 * \text{hidden\_size} \times \text{hidden\_size})$
- `bias_ih_l[k]`: the learnable input-hidden bias of the  $k^{th}$  layer ( $b_{ih}^l$ ), of shape  $(3 * \text{hidden\_size})$
- `bias_hh_l[k]`: the learnable hidden-hidden bias of the  $k^{th}$  layer ( $b_{hh}^l$ ), of shape  $(3 * \text{hidden\_size})$

**Note**

All the weights and biases are initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{1}{\text{hidden\_size}}$

**Examples**

```
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\_hardshrink      *Hardshrink module*

---

**Description**

Applies the hard shrinkage function element-wise:

**Usage**

```
nn_hardshrink(lambda = 0.5)
```

**Arguments**

lambda            the  $\lambda$  value for the Hardshrink formulation. Default: 0.5

**Details**

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

**Shape**

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
if (torch_is_installed()) {  
  m <- nn_hardshrink()  
  input <- torch_randn(2)  
  output <- m(input)  
}
```

---

nn\_hardsigmoid      *Hardsigmoid module*

---

**Description**

Applies the element-wise function:

**Usage**

```
nn_hardsigmoid()
```

**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**

```
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**

```

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 HardTanh function element-wise. HardTanh is defined as:

**Usage**

```
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**

```

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

```
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

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

where  $L = \{l_1, \dots, l_N\}^\top$ .

### 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	<i>Identity module</i>
-------------	------------------------

---

**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	<i>Calculate gain</i>
------------------------	-----------------------

---

**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

```
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**

```
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**

```
nn_init_eye_(tensor)
```

**Arguments**

tensor                      a 2-dimensional torch tensor.

**Examples**

```
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.

**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 1 - 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**

```
nn_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**

```
nn_init_trunc_normal_(tensor, mean = 0, std = 1, a = -2, b = 2)
```

**Arguments**

tensor	an n-dimensional Tensor
mean	the mean of the normal distribution
std	the standard deviation of the normal distribution
a	the minimum cutoff value
b	the maximum cutoff value

**Examples**

```
if (torch_is_installed()) {  
  w <- torch_empty(3, 5)  
  nn_init_trunc_normal_(w)  
}
```

---

nn\_init\_uniform\_      *Uniform initialization*

---

**Description**

Fills the input Tensor with values drawn from the uniform distribution

**Usage**

```
nn_init_uniform_(tensor, a = 0, b = 1)
```

**Arguments**

tensor	an n-dimensional Tensor
a	the lower bound of the uniform distribution
b	the upper bound of the uniform distribution

**Examples**

```
if (torch_is_installed()) {  
  w <- torch_empty(3, 5)  
  nn_init_uniform_(w)  
}
```

---

`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  
- Glorot, X. & Bengio, Y. (2010), using a normal distribution.

**Usage**

```
nn_init_xavier_normal_(tensor, gain = 1)
```

**Arguments**

tensor	an n-dimensional Tensor
gain	an optional scaling factor

**Examples**

```
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  
- Glorot, X. & Bengio, Y. (2010), using a uniform distribution.

**Usage**

```
nn_init_xavier_uniform_(tensor, gain = 1)
```

**Arguments**

tensor	an n-dimensional Tensor
gain	an optional scaling factor

**Examples**

```
if (torch_is_installed()) {
  w <- torch_empty(3, 5)
  nn_init_xavier_uniform_(w)
}
```

---

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_zeros_(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:

$$l(x, y) = L = \{l_1, \dots, l_N\}, \quad l_n = y_n \cdot (\log y_n - x_n)$$

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

```
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:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where  $N$  is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

$x$  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:  $(N, *)$  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

**Examples**

```
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()
}
```

---

 nn\_layer\_norm

*Layer normalization*


---

**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  $[* \times \text{normalized\_shape}[0] \times \text{normalized\_shape}[1] \times \dots \times \text{normalized\_shape}[-1]]$ . 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**

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

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:  $(N, *)$
- Output:  $(N, *)$  (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**

```
if (torch_is_installed()) {
  input <- torch_randn(20, 5, 10, 10)
  # With Learnable Parameters
  m <- nn_layer_norm(input$size()[-1])
}
```

```

# Without Learnable Parameters
m <- nn_layer_norm(input$size()[-1], elementwise_affine = FALSE)
# Normalize over last two dimensions
m <- nn_layer_norm(c(10, 10))
# Normalize over last dimension of size 10
m <- nn_layer_norm(10)
# Activating the module
output <- m(input)
}

```

---

nn\_leaky\_relu

*LeakyReLU module*


---

### Description

Applies the element-wise function:

### Usage

```
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

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x)$$

or

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

### Shape

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

### Examples

```

if (torch_is_installed()) {
m <- nn_leaky_relu(0.1)
input <- torch_randn(2)
output <- m(input)
}

```



---

nn_linear	<i>Linear module</i>
-----------	----------------------

---

### Description

Applies a linear transformation to the incoming data:  $y = xA^T + b$

### Usage

```
nn_linear(in_features, out_features, bias = 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 = in\_features.
- Output: (N, \*, H\_out) where all but the last dimension are the same shape as the input and :math:H\_{out} = out\\_features.

### Attributes

- weight: the learnable weights of the module of shape (out\_features, in\_features). The values are initialized from  $U(-\sqrt{k}, \sqrt{k})$ s, where  $k = \frac{1}{in\_features}$
- bias: the learnable bias of the module of shape (out\_features). If bias is TRUE, the values are initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{1}{in\_features}$

### Examples

```
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

```
nn_log_sigmoid()
```

### Shape

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

### Examples

```
if (torch_is_installed()) {  
  m <- nn_log_sigmoid()  
  input <- torch_randn(2)  
  output <- m(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:

### Usage

```
nn_log_softmax(dim)
```

### Arguments

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**

```
if (torch_is_installed()) {
  m <- nn_log_softmax(1)
  input <- torch_randn(2, 3)
  output <- m(input)
}
```

---

nn_lp_pool1d	<i>Applies a 1D power-average pooling over an input signal composed of several input planes.</i>
--------------	--

---

**Description**

On each window, the function computed is:

**Usage**

```
nn_lp_pool1d(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	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**

```
if (torch_is_installed()) {
# power-2 pool of window of length 3, with stride 2.
m <- nn_lp_pool1d(2, 3, stride = 2)
input <- torch_randn(20, 16, 50)
output <- m(input)
}
```

---

nn_lp_pool2d	<i>Applies a 2D power-average pooling over an input signal composed of several input planes.</i>
--------------	--

---

**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 `kernel_size`, `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:  $(N, C, H_{in}, W_{in})$
- Output:  $(N, C, H_{out}, W_{out})$ , where

$$H_{out} = \left\lfloor \frac{H_{in} - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

**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**

```
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	<i>Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.</i>
---------	--

---

### 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

$$\begin{aligned}
 i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
 f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
 g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
 o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
 c_t &= f_t c_{(t-1)} + i_t g_t \\
 h_t &= o_t \tanh(c_t)
 \end{aligned}$$

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  $t = \text{seq\_len}$ . 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  $t = \text{seq\_len}$

### Attributes

- `weight_ih_l[k]`: the learnable input-hidden weights of the  $k^{\text{th}}$  layer ( $W_{ii}|W_{if}|W_{ig}|W_{io}$ ), of shape ( $4*\text{hidden\_size} \times \text{input\_size}$ )
- `weight_hh_l[k]`: the learnable hidden-hidden weights of the  $k^{\text{th}}$  layer ( $W_{hi}|W_{hf}|W_{hg}|W_{ho}$ ), of shape ( $4*\text{hidden\_size} \times \text{hidden\_size}$ )
- `bias_ih_l[k]`: the learnable input-hidden bias of the  $k^{\text{th}}$  layer ( $b_{ii}|b_{if}|b_{ig}|b_{io}$ ), of shape ( $4*\text{hidden\_size}$ )
- `bias_hh_l[k]`: the learnable hidden-hidden bias of the  $k^{\text{th}}$  layer ( $b_{hi}|b_{hf}|b_{hg}|b_{ho}$ ), of shape ( $4*\text{hidden\_size}$ )

### Note

All the weights and biases are initialized from  $\mathcal{U}(-\sqrt{k}, \sqrt{k})$  where  $k = \frac{1}{\text{hidden\_size}}$

**Examples**

```

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))
}

```

---

nn\_margin\_ranking\_loss

*Margin ranking loss*


---

**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**

```
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:

$$\text{loss}(x_1, x_2, y) = \max(0, -y * (x_1 - x_2) + \text{margin})$$

**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**

```

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

*MaxPool1D module***Description**

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

**Usage**

```

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:

$$out(N_i, C_j, k) = \max_{m=0, \dots, \text{kernel\_size}-1} input(N_i, C_j, \text{stride} \times k + m)$$

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**

```
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**

```
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:

$$out(N_i, C_j, h, w) = \max_{m=0, \dots, kH-1} \max_{n=0, \dots, kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

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

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

**Examples**

```
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	<i>Applies a 3D max pooling over an input signal composed of several input planes.</i>
---------------	--

---

**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**

$$\text{out}(N_i, C_j, d, h, w) = \max_{k=0, \dots, kD-1} \max_{m=0, \dots, kH-1} \max_{n=0, \dots, kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times d + k, \text{stride}[1] \times h + m, \text{stride}[2] \times w + n)$$

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**

```

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	<i>Computes a partial inverse of MaxPool1d.</i>
-----------------	---

---

**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**

```
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

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0]$$

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**

```
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**

```
nn_max_unpool2d(kernel_size, stride = NULL, padding = 0)
```

**Arguments**

<code>kernel_size</code>	(int or tuple): Size of the max pooling window.
<code>stride</code>	(int or tuple): Stride of the max pooling window. It is set to <code>kernel_size</code> by default.
<code>padding</code>	(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:  $(N, C, H_{out}, W_{out})$ , where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0]$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel\_size}[1]$$

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**

```
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	<i>Computes a partial inverse of MaxPool3d.</i>
-----------------	---

---

**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**

```
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
...	methods implementation
private	passed to <code>R6::R6Class()</code> .
active	passed to <code>R6::R6Class()</code> .
parent_env	passed to <code>R6::R6Class()</code> .

**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 and 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.

### Cloning

To finalize the cloning of a module, you can define a private `finalize_deep_clone()` method. This method is called on the cloned object when deep-cloning a module, after all the modules, parameters and buffers were already cloned.

### 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**

```
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	<i>Holds submodules in a list.</i>
----------------	------------------------------------

---

**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**

```
nn_module_list(modules = list())
```

**Arguments**

modules            a list of modules to add

**See Also**[nn\\_module\\_dict\(\)](#)**Examples**

```
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

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 reduction set to 'none') loss can be described as:

### Usage

```
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

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where  $N$  is the batch size. If reduction is not 'none' (default 'mean'), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

$x$  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 reduction = 'sum'.

### Shape

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Target:  $(N, *)$ , same shape as the input

**Examples**

```

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()
}

```

---

nn\_multihead\_attention

*MultiHead attention*


---

**Description**

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

**Usage**

```

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**

embed_dim	total dimension of the model.
num_heads	parallel attention heads. Note that embed_dim will be split across num_heads (i.e. each head will have dimension embed_dim %% num_heads).
dropout	a Dropout layer on attn_output_weights. Default: 0.0.
bias	add bias as module parameter. Default: True.
add_bias_kv	add bias to the key and value sequences at dim=0.
add_zero_attn	add a new batch of zeros to the key and value sequences at dim=1.
kdim	total number of features in key. Default: NULL
vdim	total number of features in value. Default: NULL. Note: if kdim and vdim are NULL, they will be set to embed_dim such that query, key, and value have the same number of features.

`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

$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$  where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

## 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  $(N * \text{num\_heads}, L, S)$  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 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

```
if (torch_is_installed()) {
  ## Not run:
  multihead_attn <- nn_multihead_attention(embed_dim, num_heads)
  out <- multihead_attn(query, key, value)
```

```

attn_output <- out[[1]]
attn_output_weights <- out[[2]]

## End(Not run)

}

```

---

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

```
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

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{x.size(0)}$$

where  $x \in \{0, \dots, x.size(0) - 1\}$ ,  $y \in \{0, \dots, y.size(0) - 1\}$ ,  $0 \leq y[j] \leq x.size(0) - 1$ , and  $i \neq y[j]$  for all  $i$  and  $j$ .  $y$  and  $x$  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**

```

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**

```
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  $i \in \{0, \dots, \text{x.nElement()} - 1\}$ ,  $y[i] \in \{0, 1\}$ .

**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,  $0 \leq y \leq x.size(1) - 1$ ):

### Usage

```
nn_multi_margin_loss(p = 1, margin = 1, weight = NULL, reduction = "mean")
```

### Arguments

<code>p</code>	(int, optional): Has a default value of 1. 1 and 2 are the only supported values.
<code>margin</code>	(float, optional): Has a default value of 1.
<code>weight</code>	(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.
<code>reduction</code>	(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:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, \text{margin} - x[y] + x[i])^p}{x.size(0)}$$

where  $x \in \{0, \dots, x.size(0) - 1\}$  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:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, w[y] * (\text{margin} - x[y] + x[i]))^p}{x.size(0)}$$

---

nn_nll_loss	<i>Nll loss</i>
-------------	-----------------

---

### Description

The negative log likelihood loss. It is useful to train a classification problem with C classes.

### Usage

```
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  $(minibatch, C, d_1, d_2, \dots, d_K)$  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  $C$  = 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:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n, y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

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

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n}} l_n, & \text{if reduction = 'mean'}; \\ \sum_{n=1}^N l_n, & \text{if reduction = 'sum'}. \end{cases}$$

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size  $(minibatch, C, d_1, d_2, \dots, d_K)$  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  $(N, C, d_1, d_2, \dots, d_K)$  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  $(N, d_1, d_2, \dots, d_K)$  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  $(N, d_1, d_2, \dots, d_K)$  with  $K \geq 1$  in the case of K-dimensional loss.

### Examples

```
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()
}
```

---

nn\_pairwise\_distance    *Pairwise distance*

---

### Description

Computes the batchwise pairwise distance between vectors  $v_1, v_2$  using the p-norm:

**Usage**

```
nn_pairwise_distance(p = 2, eps = 1e-06, keepdim = 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  $D$  = vector dimension
- Input2:  $(N, D)$ , same shape as the Input1
- Output:  $(N)$ . If `keepdim` is TRUE, then  $(N, 1)$ .

**Examples**

```
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 `nn_module` that `x` is a parameter

**Usage**

```
nn_parameter(x, requires_grad = TRUE)
```

**Arguments**

`x` the tensor that you want to indicate as parameter  
`requires_grad` whether this parameter should have `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

```
nn_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 $\exp(\text{input}) - \text{target} * \text{input}$ , if FALSE the loss is $\text{input} - \text{target} * \log(\text{input} + \text{eps})$ .
full	(bool, optional): whether to compute full loss, i. e. to add the Stirling approximation term $\text{target} * \log(\text{target}) - \text{target} + 0.5 * \log(2\pi\text{target})$ .
eps	(float, optional): Small value to avoid evaluation of $\log(0)$ when <code>log_input = FALSE</code> . 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

$$\text{target} \sim \text{Poisson}(\text{input}) \text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} * \log(\text{input}) + \log(\text{target}!)$$

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**

```

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:

$$\text{PReLU}(x) = \max(0, x) + a * \min(0, x)$$

or

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases}$$

**Usage**

```
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:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Attributes**

- `weight` (Tensor): the learnable weights of shape  $(\text{num\_parameters})$ .

**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**

```
if (torch_is_installed()) {  
  m <- nn_prelu()  
  input <- torch_randn(2)  
  output <- m(input)  
}
```

---

nn_prune_head	<i>Prune top layer(s) of a network</i>
---------------	--

---

**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**

```
if (torch_is_installed()) {  
  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)  
    )  
  }  
}
```

```

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

```
nn_relu(inplace = FALSE)
```

### Arguments

`inplace` can optionally do the operation in-place. Default: FALSE

### Shape

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

### Examples

```

if (torch_is_installed()) {
m <- nn_relu()
input <- torch_randn(2)
m(input)
}

```

---

nn\_relu6

*ReLU6 module*


---

### Description

Applies the element-wise function:

### Usage

```
nn_relu6(inplace = FALSE)
```



**Arguments**

`inplace` can optionally do the operation in-place. Default: FALSE

**Details**

$$\text{ReLU6}(x) = \min(\max(0, x), 6)$$

**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**

<code>input_size</code>	The number of expected features in the input $x$
<code>hidden_size</code>	The number of features in the hidden state $h$
<code>num_layers</code>	Number of recurrent layers. E.g., setting <code>num_layers=2</code> 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
<code>nonlinearity</code>	The non-linearity to use. Can be either <code>'tanh'</code> or <code>'relu'</code> . Default: <code>'tanh'</code>
<code>bias</code>	If <code>FALSE</code> , then the layer does not use bias weights <code>b_ih</code> and <code>b_hh</code> . Default: <code>TRUE</code>
<code>batch_first</code>	If <code>TRUE</code> , then the input and output tensors are provided as <code>(batch, seq, feature)</code> . Default: <code>FALSE</code>
<code>dropout</code>	If non-zero, introduces a Dropout layer on the outputs of each RNN layer except the last layer, with dropout probability equal to <code>dropout</code> . Default: 0
<code>bidirectional</code>	If <code>TRUE</code> , becomes a bidirectional RNN. Default: <code>FALSE</code>
<code>...</code>	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)`.

**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  $S = \text{num\_layers} * \text{num\_directions}$  If the RNN is bidirectional,  $\text{num\_directions}$  should be 2, else it should be 1.
- Output1:  $(L, N, H_{all})$  where  $H_{all} = \text{num\_directions} * \text{hidden\_size}$
- Output2:  $(S, N, H_{out})$  tensor containing the next hidden state for each element in the batch

**Attributes**

- `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  $(\text{hidden\_size}, \text{num\_directions} * \text{hidden\_size})$
- `weight_hh_l[k]`: the learnable hidden-hidden weights of the k-th layer, of shape  $(\text{hidden\_size}, \text{hidden\_size})$
- `bias_ih_l[k]`: the learnable input-hidden bias of the k-th layer, of shape  $(\text{hidden\_size})$
- `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**

```
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**

```
nn_rrelu(lower = 1/8, upper = 1/3, inplace = FALSE)
```

**Arguments**

<code>lower</code>	lower bound of the uniform distribution. Default: $\frac{1}{8}$
<code>upper</code>	upper bound of the uniform distribution. Default: $\frac{1}{3}$
<code>inplace</code>	can optionally do the operation in-place. Default: FALSE

**Details**

Empirical Evaluation of Rectified Activations in Convolutional Network.

The function is defined as:

$$\text{RReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases}$$

where  $a$  is randomly sampled from uniform distribution  $\mathcal{U}(\text{lower}, \text{upper})$ . 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**

```
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**

```
nn_selu(inplace = FALSE)
```

**Arguments**

`inplace` (bool, optional): can optionally do the operation in-place. Default: FALSE

**Details**

$$\text{SELU}(x) = \text{scale} * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1)))$$

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:  $(N, *)$ , same shape as the input

**Examples**

```
if (torch_is_installed()) {  
  m <- nn_selu()  
  input <- torch_randn(2)  
  output <- m(input)  
}
```

---

nn_sequential	<i>A sequential container</i>
---------------	-------------------------------

---

**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**

```
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`*Sigmoid module*

---

**Description**

Applies the element-wise function:

**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**

```
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

$$\text{loss}(x, y) = \frac{1}{n} \sum_i z_i$$

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:

### Usage

```
nn_softmax(dim)
```

### Arguments

`dim` (int): A dimension along which Softmax will be computed (so every slice along `dim` will sum to 1).

### Details

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

When the input Tensor is a sparse tensor then the unspecified values are treated as `-Inf`.

### 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

```
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**

```
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:

$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x))$$

**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  $input \times \beta > threshold$ .

**Shape**

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
if (torch_is_installed()) {
  m <- nn_softplus()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

nn_softshrink	<i>Softshrink module</i>
---------------	--------------------------

---

**Description**

Applies the soft shrinkage function elementwise:

**Usage**

```
nn_softshrink(lambd = 0.5)
```

**Arguments**

lambd	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:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
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

```
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**

$$\text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{x.\text{nelement}()}$$

**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**

```
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**

```
if (torch_is_installed()) {
  m <- nn_tanh()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

nn_tanhshrink	<i>Tanhshrink module</i>
---------------	--------------------------

---

**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**

```
if (torch_is_installed()) {
  m <- nn_tanhshrink()
  input <- torch_randn(2)
  output <- m(input)
}
```

---

nn_threshold	<i>Threshold module</i>
--------------	-------------------------

---

**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:

$$y = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

**Shape**

- Input:  $(N, *)$  where  $*$  means, any number of additional dimensions
- Output:  $(N, *)$ , same shape as the input

**Examples**

```
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**

```
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 <a href="#">Learning shallow convolutional feature descriptors with triplet losses</a> by V. Balntas, E. Riba et al. 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.

### 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

$$d(x_i, y_i) = |\mathbf{x}_i - \mathbf{y}_i|_p$$

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

```
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()
}
```

---

nn\_triplet\_margin\_with\_distance\_loss

*Triplet margin with distance loss*

---

### 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, <code>nn_pairwise_distance()</code> 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 <a href="#">Learning shallow convolutional feature descriptors with triplet losses</a> 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:

$$\ell(a, p, n) = L = \{l_1, \dots, l_N\}^\top, \quad l_i = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$$

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 `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:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction = 'mean'}; \\ \text{sum}(L), & \text{if reduction = 'sum'}. \end{cases}$$

See also `nn_triplet_margin_loss()`, which computes the triplet loss for input tensors using the  $l_p$  distance as the distance function.

**Shape**

- Input:  $(N, *)$  where  $*$  represents any number of additional dimensions as supported by the distance function.
- Output: A Tensor of shape  $(N)$  if reduction is 'none', or a scalar otherwise.

**Examples**

```

if (torch_is_installed()) {
# Initialize embeddings
embedding <- nn_embedding(1000, 128)
anchor_ids <- torch_randint(1, 1000, 1, dtype = torch_long())
positive_ids <- torch_randint(1, 1000, 1, dtype = torch_long())
negative_ids <- torch_randint(1, 1000, 1, dtype = torch_long())
anchor <- embedding(anchor_ids)
positive <- embedding(positive_ids)
negative <- embedding(negative_ids)

# Built-in Distance Function
triplet_loss <- nn_triplet_margin_with_distance_loss(
  distance_function = nn_pairwise_distance()
)
output <- triplet_loss(anchor, positive, negative)

# Custom Distance Function
l_infinity <- function(x1, x2) {
  torch_max(torch_abs(x1 - x2), dim = 1)[[1]]
}

triplet_loss <- nn_triplet_margin_with_distance_loss(
  distance_function = l_infinity, margin = 1.5
)
output <- triplet_loss(anchor, positive, negative)

# Custom Distance Function (Lambda)
triplet_loss <- nn_triplet_margin_with_distance_loss(
  distance_function = function(x, y) {
    1 - nnf_cosine_similarity(x, y)
  }
)
output <- triplet_loss(anchor, positive, negative)
}

```

nn\_unflatten

*Unflattens a tensor dim expanding it to a desired shape. For use with [\[nn\\_sequential\]](#).*

**Description**

Unflattens a tensor dim expanding it to a desired shape. For use with [\[nn\\_sequential\]](#).

**Usage**

```
nn_unflatten(dim, unflattened_size)
```

**Arguments**

```
dim          Dimension to be unflattened
unflattened_size  New shape of the unflattened dimension
```

**Examples**

```
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)
}
```

---

nn\_utils\_clip\_grad\_norm\_

*Clips gradient norm of an iterable of parameters.*

---

### 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

<code>parameters</code>	(IterableTensor or Tensor): an iterable of Tensors or a single Tensor that will have gradients normalized
<code>max_norm</code>	(float or int): max norm of the gradients
<code>norm_type</code>	(float or int): type of the used p-norm. Can be Inf 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**

<code>parameters</code>	(Iterable(Tensor) or Tensor): an iterable of Tensors or a single Tensor that will have gradients normalized
<code>clip_value</code>	(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 $B \times T \times *$ 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]` should be the longest sequence, and `input[, B]` 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.

---

nn\_utils\_rnn\_pack\_sequence

*Packs a list of variable length Tensors*

---

**Description**

sequences should be a list of Tensors of size  $L \times *$ , where  $L$  is the length of a sequence and  $*$  is any number of trailing dimensions, including zero.

**Usage**

```
nn_utils_rnn_pack_sequence(sequences, enforce_sorted = TRUE)
```

**Arguments**

sequences	(list[Tensor]): A list of sequences of decreasing length.
enforce_sorted	(bool, optional): if TRUE, checks that the input contains sequences sorted by length in a decreasing order. If FALSE, this condition is not checked. Default: TRUE.

**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**

```
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**

```
nn_utils_rnn_pad_packed_sequence(
  sequence,
  batch_first = FALSE,
  padding_value = 0,
  total_length = NULL
)
```

**Arguments**

<code>sequence</code>	( <code>PackedSequence</code> ): batch to pad
<code>batch_first</code>	(bool, optional): if <code>True</code> , the output will be in “ <code>B x T x *</code> ” format.
<code>padding_value</code>	(float, optional): values for padded elements.
<code>total_length</code>	(int, optional): if not <code>NULL</code> , the output will be padded to have length <code>total_length</code> . This method will throw <code>ValueError</code> if <code>total_length</code> is less than the max sequence length in <code>sequence</code> .

**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  $\rightarrow$  recurrent network  $\rightarrow$  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 x B x \* if batch\_first is FALSE. Tensor of size B x T x \* otherwise

**Note**

This function returns a Tensor of size T x B x \* or B x T x \* 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**

```
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    *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:*

```
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:*

```
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>

**Examples**

```
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	<i>Creates a custom optimizer</i>
-----------	-----------------------------------

---

### 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

```
optimizer(
  name = NULL,
  inherit = Optimizer,
  ...,
  private = NULL,
  active = NULL,
  parent_env = parent.frame()
)
```

### Arguments

<code>name</code>	(optional) name of the optimizer
<code>inherit</code>	(optional) you can inherit from other optimizers to re-use some methods.
<code>...</code>	Pass any number of fields or methods. You should at least define the <code>initialize</code> and <code>step</code> methods. See the examples section.
<code>private</code>	(optional) a list of private methods for the optimizer.
<code>active</code>	(optional) a list of active methods for the optimizer.
<code>parent_env</code>	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

```
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) {
```

```

    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)
}

```

---

 OptimizerIgnite

*Abstract Base Class for LibTorch Optimizers*


---

### Description

Abstract base class for wrapping LibTorch C++ optimizers.

### Super class

`torch::torch_optimizer -> OptimizerIgnite`

### Methods

#### Public methods:

- [OptimizerIgnite\\$new\(\)](#)

- `OptimizerIgnite$state_dict()`
- `OptimizerIgnite$load_state_dict()`
- `OptimizerIgnite$step()`
- `OptimizerIgnite$zero_grad()`
- `OptimizerIgnite$add_param_group()`
- `OptimizerIgnite$clone()`

**Method** `new()`: Initializes the optimizer with the specified parameters and defaults.

*Usage:*

`OptimizerIgnite$new(params, defaults)`

*Arguments:*

`params` (`list()`)

Either a list of tensors or a list of parameter groups, each containing the params to optimizer as well as the optimizer options such as the learning rate, weight decay, etc.

`defaults` (`list()`)

A list of default optimizer options.

**Method** `state_dict()`: Returns the state dictionary containing the current state of the optimizer. The returned `list()` contains two lists:

- `param_groups`: The parameter groups of the optimizer (`lr, ...`) as well as to which parameters they are applied (`params, integer indices`)
- `state`: The states of the optimizer. The names are the indices of the parameters to which they belong, converted to character.

*Usage:*

`OptimizerIgnite$state_dict()`

*Returns:* (`list()`)

**Method** `load_state_dict()`: Loads the state dictionary into the optimizer.

*Usage:*

`OptimizerIgnite$load_state_dict(state_dict)`

*Arguments:*

`state_dict` (`list()`)

The state dictionary to load into the optimizer.

**Method** `step()`: Performs a single optimization step.

*Usage:*

`OptimizerIgnite$step(closure = NULL)`

*Arguments:*

`closure` (`function()`)

A closure that conducts the forward pass and returns the loss.

*Returns:* (`numeric()`)

The loss.

**Method** `zero_grad()`: Zeros out the gradients of the parameters.

*Usage:*

```
OptimizerIgnite$zero_grad()
```

**Method** `add_param_group()`: Adds a new parameter group to the optimizer.

*Usage:*

```
OptimizerIgnite$add_param_group(param_group)
```

*Arguments:*

```
param_group (list())
```

A parameter group to add to the optimizer. This should contain the params to optimize as well as the optimizer options. For all options that are not specified, the defaults are used.

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

*Usage:*

```
OptimizerIgnite$clone(deep = FALSE)
```

*Arguments:*

```
deep Whether to make a deep clone.
```

---

optimizer\_ignite

*Abstract Base Class for LibTorch Optimizers*

---

## Description

Abstract base class for wrapping LibTorch C++ optimizers.

## Usage

```
optimizer_ignite(
  name = NULL,
  ...,
  private = NULL,
  active = NULL,
  parent_env = parent.frame()
)
```

## Arguments

<code>name</code>	(optional) name of the optimizer
<code>...</code>	Pass any number of fields or methods. You should at least define the <code>initialize</code> and <code>step</code> methods. See the examples section.
<code>private</code>	(optional) a list of private methods for the optimizer.
<code>active</code>	(optional) a list of active methods for the optimizer.
<code>parent_env</code>	used to capture the right environment to define the class. The default is <code>fine</code> for most situations.

---

optim_adadelta	<i>Adadelta optimizer</i>
----------------	---------------------------

---

### 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 `$cuda()`, please do so before constructing optimizers for it. Parameters of a model after `$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

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**

```

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)
}

```

---

optim_adagrad	<i>Adagrad optimizer</i>
---------------	--------------------------

---

**Description**

Proposed in [Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](#)

**Usage**

```

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

The equation above and some remarks quoted after *An overview of gradient descent optimization algorithms* by Sebastian Ruder.

---

optim\_adam

*Implements Adam algorithm.*

---

**Description**

It has been proposed in [Adam: A Method for Stochastic Optimization](#).

**Usage**

```
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[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 <a href="#">On the Convergence of Adam and Beyond</a> (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**

```
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()

  ## End(Not run)
}
```

---

 optim\_adamw

*Implements AdamW algorithm*


---

**Description**

For further details regarding the algorithm we refer to [Decoupled Weight Decay Regularization](#)

**Usage**

```
optim_adamw(
  params,
  lr = 0.001,
  betas = c(0.9, 0.999),
  eps = 1e-08,
  weight_decay = 0.01,
  amsgrad = FALSE
)
```

**Arguments**

<code>params</code>	(iterable): iterable of parameters to optimize or dicts defining parameter groups
<code>lr</code>	(float, optional): learning rate (default: 1e-3)
<code>betas</code>	(Tuple[float, float], optional): coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
<code>eps</code>	(float, optional): term added to the denominator to improve numerical stability (default: 1e-8)
<code>weight_decay</code>	(float, optional): weight decay (L2 penalty) (default: 0)
<code>amsgrad</code>	(boolean, optional): whether to use the AMSGrad variant of this algorithm from the paper <a href="#">On the Convergence of Adam and Beyond</a> (default: FALSE)

---

 optim\_asgd

*Averaged Stochastic Gradient Descent optimizer*


---

**Description**

Proposed in [Acceleration of stochastic approximation by averaging](#)

**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\_ignite\_adagrad *LibTorch implementation of Adagrad*

---

## Description

Proposed in [Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](#)

## Usage

```
optim_ignite_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)

## Fields and Methods

See [OptimizerIgnite](#).

## Examples

```
if (torch_is_installed()) {  
  ## Not run:  
  optimizer <- optim_ignite_adagrad(model$parameters(), lr = 0.1)  
  optimizer$zero_grad()  
  loss_fn(model(input), target)$backward()  
}
```

```
optimizer$step()

## End(Not run)
}
```

---

optim\_ignite\_adam      *LibTorch implementation of Adam*

---

### Description

It has been proposed in [Adam: A Method for Stochastic Optimization](#).

### Usage

```
optim_ignite_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[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 <a href="#">On the Convergence of Adam and Beyond</a> (default: FALSE)

### Fields and Methods

See [OptimizerIgnite](#).

### Examples

```
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_ignite_adam(model$parameters(), lr = 0.1)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()
}
```

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

---

```
optim_ignite_adamw    LibTorch implementation of AdamW
```

---

## Description

For further details regarding the algorithm we refer to [Decoupled Weight Decay Regularization](#)

## Usage

```
optim_ignite_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 <a href="#">On the Convergence of Adam and Beyond</a> (default: FALSE)

## Fields and Methods

See [OptimizerIgnite](#).

## Examples

```
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_ignite_adamw(model$parameters(), lr = 0.1)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()

  ## End(Not run)
}
```

---

optim\_ignite\_rmsprop *LibTorch implementation of RMSProp*

---

### Description

Proposed by G. Hinton in his course.

### Usage

```
optim_ignite_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)

### Fields and Methods

See [OptimizerIgnite](#).

### Examples

```
if (torch_is_installed()) {
  ## Not run:
  optimizer <- optim_ignite_rmsprop(model$parameters(), lr = 0.1)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()

  ## End(Not run)
}
```



---

optim\_ignite\_sgd      *LibTorch implementation of SGD*

---

### 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

```
optim_ignite_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)

### Fields and Methods

See [OptimizerIgnite](#).

### Examples

```
if (torch_is_installed()) {  
  ## Not run:  
  optimizer <- optim_ignite_sgd(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

```
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  $\text{param\_bytes} * (\text{history\_size} + 1)$  bytes). If it doesn't fit in memory try reducing the history size, or use a different algorithm.

### Examples

```
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	<i>Dummy value indicating a required value.</i>
----------------	---

---

### Description

Dummy value indicating a required value.

### Usage

```
optim_required()
```

---

optim_rmsprop	<i>RMSprop optimizer</i>
---------------	--------------------------

---

### Description

Proposed by G. Hinton in his course.

### Usage

```
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 `$cuda()`, please do so before constructing optimizers for it. Parameters of a model after `$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

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} * g_t$$

---

optim\_rprop

*Implements the resilient backpropagation algorithm.*

---

## Description

Proposed first in [RPROP - A Fast Adaptive Learning Algorithm](#)

## Usage

```
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, etaplis), 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

```
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

```
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

$$\begin{aligned} v_{t+1} &= \mu * v_t + g_{t+1}, \\ p_{t+1} &= p_t - lr * v_{t+1}, \end{aligned}$$

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

$$\begin{aligned} v_{t+1} &= \mu * v_t + lr * g_{t+1}, \\ p_{t+1} &= p_t - v_{t+1}. \end{aligned}$$

The Nesterov version is analogously modified.

**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_sgd(model$parameters(), lr = 0.1, momentum = 0.9)
  optimizer$zero_grad()
  loss_fn(model(input), target)$backward()
  optimizer$step()

  ## End(Not run)
}
```

---

sampler	<i>Creates a new Sampler</i>
---------	------------------------------

---

**Description**

Samplers can be used with `dataloader()` when creating batches from a torch `dataset()`.

**Usage**

```
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 <code>initialize</code> and <code>step</code> 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.

---

<code>tensor_dataset</code>	<i>Dataset wrapping tensors.</i>
-----------------------------	----------------------------------

---

**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.

---

<code>threads</code>	<i>Number of threads</i>
----------------------	--------------------------

---

**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.



**Note**

torch\_set\_threads do not work on macOS system as it must be 1.

---

torch_abs	<i>Abs</i>
-----------	------------

---

**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	<i>Absolute</i>
----------------	-----------------

---

**Description**

Absolute

**Usage**

```
torch_absolute(self)
```

**Arguments**

self (Tensor) the input tensor.

**absolute(input, \*, out=None) -> Tensor**

Alias for [torch\\_abs\(\)](#)

---

`torch_acos`*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**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_acos(a)  
}
```

---

`torch_acosh`*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  $[1, \infty)$  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**

```
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.

$$\text{out} = \text{input} + \text{other}$$

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 .

$$\text{out} = \text{input} + \text{alpha} \times \text{other}$$

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
```

```

b = torch_randn(c(4, 1))
b
torch_add(a, b)
}

```

---

torch_addbmm	<i>Addbmm</i>
--------------	---------------

---

## Description

Addbmm

## Usage

```
torch_addbmm(self, batch1, batch2, beta = 1L, alpha = 1L)
```

## Arguments

self	(Tensor) matrix to be added
batch1	(Tensor) the first batch of matrices to be multiplied
batch2	(Tensor) the second batch of matrices to be multiplied
beta	(Number, optional) multiplier for input ( $\beta$ )
alpha	(Number, optional) multiplier for batch1 @ batch2 ( $\alpha$ )

## 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.

$$out = \beta \text{input} + \alpha \left( \sum_{i=0}^{b-1} \text{batch1}_i @ \text{batch2}_i \right)$$

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	<i>Addcdiv</i>
---------------	----------------

---

**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 ( $\text{input} + \text{value} * \text{tensor1} // \text{tensor2}$ ) and [torch\\_div\(\)](#) for float inputs ( $\text{input} + \text{value} * \text{tensor1} / \text{tensor2}$ ). The new addcdiv behavior can be implemented with [torch\\_true\\_divide\(\)](#) ( $\text{input} + \text{value} * \text{torch.true\_divide}(\text{tensor1}, \text{tensor2})$ ).

$$\text{out}_i = \text{input}_i + \text{value} \times \frac{\text{tensor1}_i}{\text{tensor2}_i}$$

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	<i>Addcmul</i>
---------------	----------------

---

## 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 $tensor1 * tensor2$

## **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.

$$out_i = input_i + value \times tensor1_i \times tensor2_i$$

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

```
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

*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 $mat1 @ mat2$ ( $\alpha$ )

**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.

$$\text{out} = \beta \text{input} + \alpha (\text{mat1}_i @ \text{mat2}_i)$$

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	<i>Addmv</i>
-------------	--------------

---

**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 $mat@vec$ ( $\alpha$ )

**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.

$$\text{out} = \beta \text{input} + \alpha (\text{mat} @ \text{vec})$$

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

*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 $\text{vec1} \otimes \text{vec2}$ ( $\alpha$ )

**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**

```
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	<i>Allclose</i>
----------------	-----------------

---

**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 NaN s will be compared as equal. Default: FALSE

**allclose(input, other, rtol=1e-05, atol=1e-08, equal\_nan=False) -> bool**

This function checks if all input and other satisfy the condition:

$$|\text{input} - \text{other}| \leq \text{atol} + \text{rtol} \times |\text{other}|$$

elementwise, for all elements of input and other. The behaviour of this function is analogous to `numpy.allclose` <<https://docs.scipy.org/doc/numpy/reference/generated/numpy.allclose.html>>\_

**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)
}
```

---

`torch_amax`*Amax*

---

## 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 the `put`.

## Examples

```
if (torch_is_installed()) {  
  a <- torch_randn(c(4, 4))  
  a  
  torch_amax(a, 1)  
}
```

---

`torch_amin`*Amin*

---

## Description

Amin

## Usage

```
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

```
if (torch_is_installed()) {  
  a <- torch_randn(c(4, 4))  
  a  
  torch_amin(a, 1)  
}
```

torch\_angle      *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**

```
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      *Arange*

---

**Description**

Arange

**Usage**

```
torch_arange(
    start,
    end,
    step = 1L,
    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_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.

**arange(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  $\left\lceil \frac{\text{end} - \text{start}}{\text{step}} \right\rceil$  with values from the interval  $[\text{start}, \text{end})$  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.

$$\text{out}_{i+1} = \text{out}_i + \text{step}$$

**Examples**

```
if (torch_is_installed()) {
  torch_arange(start = 0, end = 5)
  torch_arange(1, 4)
```

```
torch_arange(1, 2.5, 0.5)
}
```

---

torch_arccos	<i>Arccos</i>
--------------	---------------

---

**Description**

Arccos

**Usage**

```
torch_arccos(self)
```

**Arguments**

`self` (Tensor) the input tensor.

**arccos(input, \*, out=None) -> Tensor**

Alias for [torch\\_acos\(\)](#).

---

torch_arccosh	<i>Arccosh</i>
---------------	----------------

---

**Description**

Arccosh

**Usage**

```
torch_arccosh(self)
```

**Arguments**

`self` (Tensor) the input tensor.

**arccosh(input, \*, out=None) -> Tensor**

Alias for [torch\\_acosh\(\)](#).



---

torch_arcsin	<i>Arcsin</i>
--------------	---------------

---

**Description**

Arcsin

**Usage**

```
torch_arcsin(self)
```

**Arguments**

self (Tensor) the input tensor.

**arcsin(input, \*, out=None) -> Tensor**

Alias for [torch\\_asin\(\)](#).

---

torch_arcsinh	<i>Arcsinh</i>
---------------	----------------

---

**Description**

Arcsinh

**Usage**

```
torch_arcsinh(self)
```

**Arguments**

self (Tensor) the input tensor.

**arcsinh(input, \*, out=None) -> Tensor**

Alias for [torch\\_asinh\(\)](#).

---

torch_arctan	<i>Arctan</i>
--------------	---------------

---

**Description**

Arctan

**Usage**

```
torch_arctan(self)
```

**Arguments**

self (Tensor) the input tensor.

**arctan(input, \*, out=None) -> Tensor**

Alias for [torch\\_atan\(\)](#).

---

torch_arctanh	<i>Arctanh</i>
---------------	----------------

---

**Description**

Arctanh

**Usage**

```
torch_arctanh(self)
```

**Arguments**

self (Tensor) the input tensor.

**arctanh(input, \*, out=None) -> Tensor**

Alias for [torch\\_atanh\(\)](#).

---

torch_argmax	<i>Argmax</i>
--------------	---------------

---

## Description

Argmax

## Arguments

self	(Tensor) the input tensor.
dim	(int) the dimension to reduce. If NULL, the argmax of the flattened input is returned.
keepdim	(bool) whether the output tensor has dim retained or not. Ignored if dim=NULL.

## 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

```
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	<i>Argmin</i>
--------------	---------------

---

## 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

```
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	<i>Argsort</i>
---------------	----------------

---

**Description**

Argsort

**Usage**

```
torch_argsort(self, dim = -1, 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**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4, 4))  
  a  
  torch_argsort(a, dim=1)  
}
```

---

torch_asin	<i>Asin</i>
------------	-------------

---

**Description**

Asin

**Usage**

```
torch_asin(self)
```

**Arguments**

`self` (Tensor) the input tensor.

**asin(input, out=NULL) -> Tensor**

Returns a new tensor with the arcsine of the elements of input.

$$\text{out}_i = \sin^{-1}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_asin(a)  
}
```

---

torch\_asinh

*Asinh*

---

**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**

```
if (torch_is_installed()) {  
  a <- torch_randn(c(4))  
  a  
  torch_asinh(a)  
}
```

---

torch_as_strided	<i>As_strided</i>
------------------	-------------------

---

### 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

```
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	<i>Atan</i>
------------	-------------

---

**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	<i>Atan2</i>
-------------	--------------

---

**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  $\text{input}_i/\text{other}_i$  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 input and other must be broadcastable .

**Examples**

```
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 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.

$$\text{out}_i = \tanh^{-1}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(4))$uniform_(-1, 1)  
  a  
  torch_atanh(a)  
}
```

---

torch_atleast_1d	<i>Atleast_1d</i>
------------------	-------------------

---

**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**

```
torch_atleast_1d(self)
```

**Arguments**

self (Tensor or list of Tensors)

**Examples**

```
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_2d	<i>Atleast_2d</i>
------------------	-------------------

---

**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**

```
torch_atleast_2d(self)
```

**Arguments**

`self` (Tensor or list of Tensors)

**Examples**

```
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	<i>Atleast_3d</i>
------------------	-------------------

---

**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**

```
torch_atleast_3d(self)
```

**Arguments**

`self` (Tensor or list of Tensors)

---

torch_avg_pool1d	<i>Avg_pool1d</i>
------------------	-------------------

---

**Description**

Avg\_pool1d

**Usage**

```
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(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	<i>Baddbmm</i>
---------------	----------------

---

**Description**

Baddbmm

**Usage**

```
torch_baddbmm(self, batch1, batch2, beta = 1L, alpha = 1L)
```

**Arguments**

self	(Tensor) the tensor to be added
batch1	(Tensor) the first batch of matrices to be multiplied
batch2	(Tensor) the second batch of matrices to be multiplied
beta	(Number, optional) multiplier for input ( $\beta$ )
alpha	(Number, optional) multiplier for batch1 @ batch2 ( $\alpha$ )

**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.

$$\text{out}_i = \beta \text{input}_i + \alpha (\text{batch1}_i @ \text{batch2}_i)$$

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(10, 3, 5))
  batch1 = torch_randn(c(10, 3, 4))
  batch2 = torch_randn(c(10, 4, 5))
  torch_baddbmm(M, batch1, batch2)
}
```

---

 torch\_bartlett\_window *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.

$$w[n] = 1 - \left| \frac{2n}{N-1} - 1 \right| = \begin{cases} \frac{2n}{N-1} & \text{if } 0 \leq n \leq \frac{N-1}{2} \\ 2 - \frac{2n}{N-1} & \text{if } \frac{N-1}{2} < n < N \end{cases} ,$$

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`  $\neq 1$ , the returned window contains a single value 1.

---

<code>torch_bernoulli</code>	<i>Bernoulli</i>
------------------------------	------------------

---

### Description

Bernoulli

### Usage

```
torch_bernoulli(self, p, generator = NULL)
```

### Arguments

<code>self</code>	(Tensor) the input tensor of probability values for the Bernoulli distribution
<code>p</code>	(Number) a probability value. If <code>p</code> is passed than it's used instead of the values in <code>self</code> tensor.
<code>generator</code>	( <code>torch.Generator</code> , 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.

**Examples**

```

if (torch_is_installed()) {

  a = torch_empty(c(3, 3))$uniform_(0, 1) # generate a uniform random matrix with range c(0, 1)
  a
  torch_bernoulli(a)
  a = torch_ones(c(3, 3)) # probability of drawing "1" is 1
  torch_bernoulli(a)
  a = torch_zeros(c(3, 3)) # probability of drawing "1" is 0
  torch_bernoulli(a)
}

```

---

torch_bincount	<i>Bincount</i>
----------------	-----------------

---

**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**

```

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	<i>Bitwise_and</i>
-------------------	--------------------

---

**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	<i>Bitwise_not</i>
-------------------	--------------------

---

**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	<i>Bitwise_or</i>
------------------	-------------------

---

**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	<i>Bitwise_xor</i>
-------------------	--------------------

---

**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.

---

 torch\_blackman\_window *Blackman\_window*


---

**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

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_blackman_window(L, periodic=TRUE)` equal to `torch_blackman_window(L + 1, periodic=False)[-1]`.

### Note

If `window_length`  $\neq 1$ , the returned window contains a single value 1.

---

`torch_block_diag`      *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(c(1, 2, 3))
  E <- torch_tensor(rbind(4, 5, 6))
  torch_block_diag(list(A, B, C, D, E))
}
```

---

`torch_bmm`            *Bmm*

---

### Description

*Bmm*

### Usage

```
torch_bmm(self, mat2)
```

**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**

```
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.

**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	<i>Bucketize</i>
-----------------	------------------

---

**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. <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> .
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))
  boundaries
  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	<i>Can_cast</i>
----------------	-----------------

---

**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	<i>Cartesian_prod</i>
----------------------	-----------------------

---

**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.

**Examples**

```

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**

```
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**

```

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	<i>Cdist</i>
-------------	--------------

---

**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	<i>Ceil</i>
------------	-------------

---

**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**

```
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_	<i>Celu_</i>
-------------	--------------

---

**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	<i>Chain_matmul</i>
--------------------	---------------------

---

**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.

### Examples

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(3, 4))  
  b = torch_randn(c(4, 5))  
  c = torch_randn(c(5, 6))  
  d = torch_randn(c(6, 7))  
  torch_chain_matmul(list(a, b, c, d))  
}
```

---

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  $(*, C/g, H, W)$ , 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	<i>Cholesky</i>
----------------	-----------------

---

**Description**

Cholesky

**Usage**

```
torch_cholesky(self, upper = FALSE)
```

**Arguments**

self	(Tensor) the input tensor $A$ 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:

$$A = L L^T$$

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**

```
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
```

```

l = torch_cholesky(a)
z = torch_matmul(l, l$transpose(-1, -2))
torch_max(torch_abs(z - a)) # Max non-zero

## End(Not run)
}

```

---

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

$$inv = (u^T u)^{-1}$$

### 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()
}

```

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

---

```
torch_cholesky_solve  Cholesky_solve
```

---

## Description

Cholesky\_solve

## Usage

```
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 or 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

```
if (torch_is_installed()) {
  a = torch_randn(c(3, 3))
  a = torch_mm(a, a$t()) # make symmetric positive definite
  u = torch_cholesky(a)
  a
  b = torch_randn(c(3, 2))
}
```

```

b
torch_cholesky_solve(b, u)
torch_mm(a$inverse(), b)
}

```

---

torch_chunk	<i>Chunk</i>
-------------	--------------

---

### Description

Chunk

### 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	<i>Clamp</i>
-------------	--------------

---

### Description

Clamp

### 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 [ min, max ] and return a resulting tensor:

$$y_i = \begin{cases} \min & \text{if } x_i < \min \\ x_i & \text{if } \min \leq x_i \leq \max \\ \max & \text{if } x_i > \max \end{cases}$$

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**

```
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**

```
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

**clip(input, min, max, \*, out=None) -> Tensor**

Alias for [torch\\_clamp\(\)](#).

---

torch_clone	<i>Clone</i>
-------------	--------------

---

**Description**

Clone

**Usage**

```
torch_clone(self, memory_format = NULL)
```

**Arguments**

`self` (Tensor) the input tensor.  
`memory_format` a torch memory format. see [torch\\_preserve\\_format\(\)](#).

**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_combinations	<i>Combinations</i>
--------------------	---------------------

---

**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	<i>Complex</i>
---------------	----------------

---

**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**

```
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**

```
if (torch_is_installed()) {
  ## Not run:
  torch_conj(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))

  ## End(Not run)
}
```

---

torch_conv1d	<i>Conv1d</i>
--------------	---------------

---

**Description**

Conv1d

**Usage**

```
torch_conv1d(
    input,
    weight,
    bias = list(),
    stride = 1L,
    padding = 0L,
    dilation = 1L,
    groups = 1L
)
```

**Arguments**

input	input tensor of shape (minibatch, in_channels, $iW$ )
weight	filters of shape (out_channels, $\frac{\text{in\_channels}}{\text{groups}}$ , $kW$ )
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

**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**

```
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**

```
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, $\frac{\text{in\_channels}}{\text{groups}}$ , $kH$ , $kW$ )
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

**conv2d(input, weight, bias=NULL, stride=1, padding=0, dilation=1, groups=1) -> Tensor**

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

See [nn\\_conv2d\(\)](#) for details and output shape.

**Examples**

```
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	<i>Conv3d</i>
--------------	---------------

---

**Description**

Conv3d

**Usage**

```
torch_conv3d(
    input,
    weight,
    bias = list(),
    stride = 1L,
    padding = 0L,
    dilation = 1L,
    groups = 1L
)
```

**Arguments**

input	input tensor of shape (minibatch, in_channels, $iT$ , $iH$ , $iW$ )
weight	filters of shape (out_channels, $\frac{\text{in\_channels}}{\text{groups}}$ , $kT$ , $kH$ , $kW$ )
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

**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**

```
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	<i>Conv_tbc</i>
----------------	-----------------

---

**Description**

Conv\_tbc

**Usage**

```
torch_conv_tbc(self, weight, bias, pad = 0L)
```

**Arguments**

self	NA input tensor of shape (sequence length $\times$ batch $\times$ in_channels)
weight	NA filter of shape (kernel width $\times$ in_channels $\times$ out_channels)
bias	NA bias of shape (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 (Time, Batch, Channels) - hence TBC.

---

torch_conv_transpose1d	<i>Conv_transpose1d</i>
------------------------	-------------------------

---

**Description**

Conv\_transpose1d

**Usage**

```
torch_conv_transpose1d(
  input,
  weight,
  bias = list(),
  stride = 1L,
  padding = 0L,
  output_padding = 0L,
  groups = 1L,
  dilation = 1L
)
```



**Arguments**

input	input tensor of shape (minibatch, in_channels, $iW$ )
weight	filters of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$ , $kW$ )
bias	optional bias of shape (out_channels). Default: NULL
stride	the stride of the convolving kernel. Can be a single number or a tuple (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 (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, 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**

```
if (torch_is_installed()) {
  inputs = torch_randn(c(20, 16, 50))
  weights = torch_randn(c(16, 33, 5))
  nnf_conv_transpose1d(inputs, weights)
}
```

---

torch\_conv\_transpose2d

*Conv\_transpose2d*

---

**Description**

Conv\_transpose2d

**Usage**

```
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, $\frac{\text{out\_channels}}{\text{groups}}$ , $kH$ , $kW$ )
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

**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**

```
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, $\frac{\text{out\_channels}}{\text{groups}}$ , $kT$ , $kH$ , $kW$ )
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

```
if (torch_is_installed()) {  
  ## Not run:  
  inputs = torch_randn(c(20, 16, 50, 10, 20))  
  weights = torch_randn(c(16, 33, 3, 3, 3))  
  nfn_conv_transpose3d(inputs, weights)  
  
  ## End(Not run)  
}
```

---

torch_cos
-----------

---

Cos
-----

---

### Description

Cos

### Usage

```
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.

$$\text{out}_i = \cos(\text{input}_i)$$

### Examples

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(4))  
  a  
  torch_cos(a)  
}
```

---

torch_cosh	<i>Cosh</i>
------------	-------------

---

**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.

$$\text{out}_i = \cosh(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_cosh(a)  
}
```

---

torch_cosine_similarity	<i>Cosine_similarity</i>
-------------------------	--------------------------

---

**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.

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

**Examples**

```
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**

```
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

*Cross*

---

**Description**

Cross

**Usage**

```
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**

```
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

*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.

$$y_i = \max(x_1, x_2, x_3, \dots, x_i)$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(10))  
  a  
  torch_cummax(a, dim=1)  
}
```

---

torch\_cummin*Cummin*

---

**Description**

Cummin

**Usage**

torch\_cummin(self, dim)



**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.

$$y_i = \min(x_1, x_2, x_3, \dots, x_i)$$

**Examples**

```
if (torch_is_installed()) {
  a = torch_randn(c(10))
  a
  torch_cummin(a, dim=1)
}
```

---

torch_cumprod	<i>Cumprod</i>
---------------	----------------

---

**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.

$$y_i = x_1 \times x_2 \times x_3 \times \dots \times x_i$$

**Examples**

```

if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cumprod(a, dim=1)
}

```

---

torch_cumsum	<i>Cumsum</i>
--------------	---------------

---

**Description**

Cumsum

**Usage**

```
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.

$$y_i = x_1 + x_2 + x_3 + \dots + x_i$$

**Examples**

```

if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cumsum(a, dim=1)
}

```

---

torch_deg2rad	<i>Deg2rad</i>
---------------	----------------

---

**Description**

Deg2rad

**Usage**

```
torch_deg2rad(self)
```

**Arguments**

self (Tensor) the input tensor.

**deg2rad(input, \*, out=None) -> Tensor**

Returns a new tensor with each of the elements of input converted from angles in degrees to radians.

**Examples**

```
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	<i>Dequantize</i>
------------------	-------------------

---

**Description**

Dequantize

**Usage**

```
torch_dequantize(tensor)
```

**Arguments**

tensor (Tensor) A quantized Tensor or a list of quantized tensors

**dequantize(tensor) -> Tensor**

Returns an fp32 Tensor by dequantizing a quantized Tensor

**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	<i>Create a Device object</i>
--------------	-------------------------------

---

**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**

```
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	<i>Diag</i>
------------	-------------

---

**Description**

Diag

**Usage**

```
torch_diag(self, diagonal = 0L)
```

**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	<i>Diagflat</i>
----------------	-----------------

---

**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.

**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	<i>Diagonal</i>
----------------	-----------------

---

**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**

```

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)
}

```

---

torch_diag_embed	<i>Diag_embed</i>
------------------	-------------------

---

**Description**

Diag\_embed

**Usage**

```
torch_diag_embed(self, offset = 0L, dim1 = -2L, dim2 = -1L)
```

**Arguments**

self	(Tensor) the input tensor. Must be at least 1-dimensional.
offset	(int, optional) which diagonal to consider. Default: 0 (main diagonal).
dim1	(int, optional) first dimension with respect to which to take diagonal. Default: -2.
dim2	(int, optional) second dimension with respect to which to take diagonal. Default: -1.

**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.



**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**

<code>self</code>	the tensor to compute the differences on
<code>n</code>	the number of times to recursively compute the difference
<code>dim</code>	the dimension to compute the difference along. Default is the last dimension.
<code>prepend</code>	values to prepend to input along <code>dim</code> before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input's shape except on <code>dim</code> .
<code>append</code>	values to append to input along <code>dim</code> before computing the difference. Their dimensions must be equivalent to that of input, and their shapes must match input's shape except on <code>dim</code> .

**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	<i>Digamma</i>
---------------	----------------

---

**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	<i>Dist</i>
------------	-------------

---

**Description**

Dist

**Usage**

```
torch_dist(self, other, p = 2L)
```

**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**

```
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**

```
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))
  a
  torch_div(a, 0.5)

  a = torch_randn(c(4, 4))
  a
  b = torch_randn(c(4))
  b
  torch_div(a, b)
}
```

---

 torch\_divide

 Divide
 

---

**Description**

Divide

**Usage**

```
torch_divide(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`.

**divide(input, other, \*, out=None) -> Tensor**

Alias for [torch\\_div\(\)](#).

---

torch_dot	<i>Dot</i>
-----------	------------

---

**Description**

Dot

**Usage**

```
torch_dot(self, tensor)
```

**Arguments**

`self` the input 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

```
if (torch_is_installed()) {  
  
  torch_dot(torch_tensor(c(2, 3)), torch_tensor(c(2, 1)))  
}
```

---

torch\_dstack

*Dstack*

---

### Description

Dstack

### Usage

```
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

```
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	<i>Torch data types</i>
-------------	-------------------------

---

**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                      *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.

---

torch\_einsum                      *Einsum*

---

### Description

Einsum

### Usage

torch\_einsum(equation, tensors, path = NULL)



**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 <code>-&gt;</code> and gives the indices for the output. If the <code>-&gt;</code> 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 <code>...</code> 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 <code>opt_einsum</code> 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 <i>the</i> optimal path is an NP-hard problem, thus, <code>opt_einsum</code> relies on different heuristics to achieve near-optimal results. If <code>opt_einsum</code> 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**

```

if (torch_is_installed()) {

x = torch_randn(c(5))
y = torch_randn(c(4))
torch_einsum('i,j->ij', list(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', list(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', list(As, Bs)) # batch matrix multiplication
A = torch_randn(c(3, 3))
torch_einsum('ii->i', list(A)) # diagonal
A = torch_randn(c(4, 3, 3))
torch_einsum('...ii->...i', list(A)) # batch diagonal
A = torch_randn(c(2, 3, 4, 5))
torch_einsum('...ij->...ji', list(A))$shape # batch permute

}

```

---

torch_empty	<i>Empty</i>
-------------	--------------

---

**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	<i>Empty_like</i>
------------------	-------------------

---

### 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.

**Examples**

```
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**

```
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**

```
if (torch_is_installed()) {  
  
  torch_eq(torch_tensor(c(1,2,3,4)), torch_tensor(c(1, 3, 2, 4)))  
}
```

---

torch_equal	<i>Equal</i>
-------------	--------------

---

**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	<i>Erf</i>
-----------	------------

---

**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:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

**Examples**

```
if (torch_is_installed()) {  
  torch_erf(torch_tensor(c(0, -1., 10.)))  
}
```

---

torch_erfc	<i>Erfc</i>
------------	-------------

---

**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:

$$\operatorname{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

**Examples**

```
if (torch_is_installed()) {  
  torch_erfc(torch_tensor(c(0, -1., 10.)))  
}
```

---

torch_erfinv	<i>Erfinv</i>
--------------	---------------

---

**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**

```
if (torch_is_installed()) {  
  torch_erfinv(torch_tensor(c(0, 0.5, -1.)))  
}
```

---

torch_exp	<i>Exp</i>
-----------	------------

---

**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**

```
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**

```
if (torch_is_installed()) {
  torch_exp2(torch_tensor(c(0, log2(2.)), 3, 4))
}
```

---

torch_expm1	<i>Expm1</i>
-------------	--------------

---

**Description**

Expm1

**Usage**

torch\_expm1(self)

**Arguments**

self (Tensor) the input tensor.

**expm1(input, out=NULL) -> Tensor**

Returns a new tensor with the exponential of the elements minus 1 of input.

$$y_i = e^{x_i} - 1$$

**Examples**

```
if (torch_is_installed()) {  
  torch_expm1(torch_tensor(c(0, log(2))))  
}
```

---

torch_eye	<i>Eye</i>
-----------	------------

---

**Description**

Eye

**Usage**

```
torch_eye(  
  n,  
  m = n,  
  dtype = NULL,  
  layout = NULL,  
  device = NULL,  
  requires_grad = FALSE  
)
```

**Arguments**

<code>n</code>	(int) the number of rows
<code>m</code>	(int, optional) the number of columns with default being <code>n</code>
<code>dtype</code>	(torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see <code>torch_set_default_tensor_type</code> ).
<code>layout</code>	(torch.layout, optional) the desired layout of returned Tensor. Default: <code>torch_strided</code> .
<code>device</code>	(torch.device, optional) the desired device of returned tensor. Default: if NULL, uses the current device for the default tensor type (see <code>torch_set_default_tensor_type</code> ). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
<code>requires_grad</code>	(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**

```
if (torch_is_installed()) {
  torch_eye(3)
}
```

---

<code>torch_fft_fft</code>	<i>Fft</i>
----------------------------	------------

---

**Description**

Computes the one dimensional discrete Fourier transform of input.

**Usage**

```
torch_fft_fft(self, n = NULL, dim = -1L, norm = NULL)
```

**Arguments**

<code>self</code>	(Tensor) the input tensor
<code>n</code>	(int) Signal length. If given, the input will either be zero-padded or trimmed to this length before computing the FFT.
<code>dim</code>	(int, optional) The dimension along which to take the one dimensional FFT.
<code>norm</code>	(str, optional) Normalization mode. For the forward transform, these correspond to: <ul style="list-style-type: none"> <li>"forward" - normalize by 1/n</li> </ul>

- "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

```
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

```
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.

layout	(default: <code>torch_strided()</code> ) the desired layout of returned tensor.
device	(default: <code>NULL</code> ) the desired device of returned tensor. Default: If <code>NULL</code> , uses the current device for the default tensor type.
requires_grad	(default: <code>FALSE</code> ) If autograd should record operations on the returned tensor.

**Note**

By convention, `torch_fft_fft()` returns positive frequency terms first, followed by the negative frequencies in reverse order, so that `f[-i]` for all  $0 < i \leq n/2$  gives the negative frequency terms. For an FFT of length  $n$  and with inputs spaced in length unit  $d$ , the frequencies are:  $f = [0, 1, \dots, (n - 1) // 2, -(n // 2), \dots, -1] / (d * n)$

For even lengths, the Nyquist frequency at `f[n/2]` can be thought of as either negative or positive. `fftfreq()` follows NumPy's convention of taking it to be negative.

**Examples**

```
if (torch_is_installed()) {
  torch_fft_fftfreq(5) # Nyquist frequency at f[3] is positive
  torch_fft_fftfreq(4) # Nyquist frequency at f[3] is given as negative
}
```

---

torch_fft_ifft	<i>Ifft</i>
----------------	-------------

---

**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: <ul style="list-style-type: none"> <li>• "forward" - no normalization</li> <li>• "backward" - normalize by 1/n</li> <li>• "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 <code>ifft()</code> the exact inverse. Default is "backward" (normalize by 1/n).</li> </ul>

**Examples**

```

if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 3)
  t
  x <- torch_fft_fft(t, norm = "backward")
  torch_fft_irfft(x)

}

```

---

torch_fft_irfft	<i>Irfft</i>
-----------------	--------------

---

**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**

```
torch_fft_irfft(self, n = NULL, dim = -1L, norm = NULL)
```

**Arguments**

<code>self</code>	(Tensor) the input tensor representing a half-Hermitian signal
<code>n</code>	(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: $n=2*(input.size(dim) - 1)$ .
<code>dim</code>	(int, optional) – The dimension along which to take the one dimensional real IFFT.
<code>norm</code>	(str, optional) – Normalization mode. For the backward transform, these correspond to: <ul style="list-style-type: none"> <li>• "forward" - no normalization</li> <li>• "backward" - normalize by 1/n</li> <li>• "ortho" - normalize by <math>1/\sqrt{n}</math> (making the real IFFT orthonormal) Calling the forward transform (<code>torch_fft_rfft()</code>) with the same normalization mode will apply an overall normalization of 1/n between the two transforms. This is required to make <code>irfft()</code> the exact inverse. Default is "backward" (normalize by 1/n).</li> </ul>

**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**

```
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())
}
```

---

torch_fft_rfft	<i>Rfft</i>
----------------	-------------

---

**Description**

Computes the one dimensional Fourier transform of real-valued input.

**Usage**

```
torch_fft_rfft(self, n = NULL, dim = -1L, norm = NULL)
```

**Arguments**

<code>self</code>	(Tensor) the real input tensor
<code>n</code>	(int) Signal length. If given, the input will either be zero-padded or trimmed to this length before computing the real FFT.
<code>dim</code>	(int, optional) – The dimension along which to take the one dimensional real FFT.
<code>norm</code>	norm (str, optional) – Normalization mode. For the forward transform, these correspond to: <ul style="list-style-type: none"> <li>• "forward" - normalize by 1/n</li> <li>• "backward" - no normalization</li> <li>• "ortho" - normalize by 1/sqrt(n) (making the FFT orthonormal) Calling the backward transform (<code>torch_fft_irfft()</code>) with the same normalization mode will apply an overall normalization of 1/n between the two transforms. This is required to make <code>irfft()</code> the exact inverse. Default is "backward" (no normalization).</li> </ul>

**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**

```
if (torch_is_installed()) {
  t <- torch_arange(start = 0, end = 3)
  torch_fft_rfft(t)
}
```

---

torch_finfo	<i>Floating point type info</i>
-------------	---------------------------------

---

**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	<i>Fix</i>
-----------	------------

---

**Description**

Fix

**Usage**

```
torch_fix(self)
```

**Arguments**

self	(Tensor) the input tensor.
------	----------------------------

**fix(input, \*, out=None) -> Tensor**

Alias for [torch\\_trunc\(\)](#)



---

torch_flatten	<i>Flatten</i>
---------------	----------------

---

**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	<i>Flip</i>
------------	-------------

---

**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**

```
if (torch_is_installed()) {
  x <- torch_arange(1, 8)$view(c(2, 2, 2))
  x
  torch_flip(x, c(1, 2))
}
```

---

torch_fliplr	<i>Fliplr</i>
--------------	---------------

---

**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.

**Examples**

```
if (torch_is_installed()) {  
  
  x <- torch_arange(start = 1, end = 4)$view(c(2, 2))  
  x  
  torch_fliplr(x)  
}
```

---

torch_flipud	<i>Flipud</i>
--------------	---------------

---

**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	<i>Floor</i>
-------------	--------------

---

**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.

$$\text{out}_i = \lfloor \text{input}_i \rfloor$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_floor(a)  
}
```

---

torch_floor_divide	<i>Floor_divide</i>
--------------------	---------------------

---

**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**

```
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	<i>Fmod</i>
------------	-------------

---

**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**

```
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, 5)), 1.5)
}
```

---

torch_frac	<i>Frac</i>
------------	-------------

---

**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.

$$\text{out}_i = \text{input}_i - \lfloor |\text{input}_i| \rfloor * \text{sgn}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  torch_frac(torch_tensor(c(1, 2.5, -3.2)))  
}
```

---

torch_full	<i>Full</i>
------------	-------------

---

**Description**

Full

**Usage**

```
torch_full(  
  size,  
  fill_value,  
  names = NULL,  
  dtype = NULL,  
  layout = NULL,  
  device = NULL,  
  requires_grad = FALSE  
)
```

**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)
}
```

---

torch_full_like	<i>Full_like</i>
-----------------	------------------

---

**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

*Gather*

---

**Description**

Gather

**Usage**

```
torch_gather(self, dim, index, sparse_grad = FALSE)
```



**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.

**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::

```

out[i][j][k] = input[index[i][j][k]][j][k] # if dim == 0
out[i][j][k] = input[i][index[i][j][k]][k] # if dim == 1
out[i][j][k] = input[i][j][index[i][j][k]] # if dim == 2
  
```

If input is an  $n$ -dimensional tensor with size  $(x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1})$  and  $\text{dim} = i$ , then index must be an  $n$ -dimensional tensor with size  $(x_0, x_1, \dots, x_{i-1}, y, x_{i+1}, \dots, x_{n-1})$  where  $y \geq 1$  and out will have the same size as index.

**Examples**

```

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

*Gcd*


---

**Description**

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  $\text{gcd}(0, 0) = 0$ .

**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	<i>Create a Generator object</i>
-----------------	----------------------------------

---

**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**

```
if (torch_is_installed()) {  
  
  # Via string  
  generator <- torch_generator()  
  generator$current_seed()  
  generator$set_current_seed(1234567L)  
  generator$current_seed()  
  
}
```

---

torch_geqrf	<i>Geqrf</i>
-------------	--------------

---

**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  $(n \times m)$ .

**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()`.

---

torch_greater_equal	<i>Greater_equal</i>
---------------------	----------------------

---

### Description

Greater\_equal

### Usage

`torch_greater_equal(self, other)`

### Arguments

self	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare

**greater\_equal(input, other, \*, out=None) -> Tensor**

Alias for `torch_ge()`.

---

torch_gt	<i>Gt</i>
----------	-----------

---

### Description

Gt

### Usage

`torch_gt(self, other)`

### Arguments

self	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare

**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.

### Examples

```
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

```
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(window\_length, periodic=TRUE, alpha=0.54, beta=0.46, dtype=NULL, layout=torch.strided, device=NULL, requires\_grad=False) -> Tensor**

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 `window_length`  $\neq 1$ , the returned window contains a single value 1.

This is a generalized version of `torch_hann_window`.

---

torch_hann_window	<i>Hann_window</i>
-------------------	--------------------

---

### Description

Hann\_window

### Usage

```
torch_hann_window(
  window_length,
  periodic = TRUE,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE
)
```

### Arguments

<code>window_length</code>	(int) the size of returned window
<code>periodic</code>	(bool, optional) If TRUE, returns a window to be used as periodic function. If False, return a symmetric window.
<code>dtype</code>	(torch.dtype, optional) the desired data type of returned tensor. Default: if NULL, uses a global default (see <code>torch_set_default_tensor_type</code> ). 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.

**hann\_window(window\_length, periodic=TRUE, dtype=NULL, layout=torch.strided, device=NULL, requires\_grad=False) -> Tensor**

Hann window function.

$$w[n] = \frac{1}{2} \left[ 1 - \cos \left( \frac{2\pi n}{N-1} \right) \right] = \sin^2 \left( \frac{\pi n}{N-1} \right),$$

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\_hann\_window(L, periodic=TRUE) equal to torch\_hann\_window(L + 1, periodic=False)[-1]).

#### Note

If `window\_length`  $\neq 1$ , the returned window contains a single value 1.

---

torch_heaviside	<i>Heaviside</i>
-----------------	------------------

---

#### Description

Heaviside

#### Usage

```
torch_heaviside(self, values)
```

#### Arguments

self	(Tensor) the input tensor.
values	(Tensor) The values to use where input is zero.

**heaviside(input, values, \*, out=None) -> Tensor**

Computes the Heaviside step function for each element in input. The Heaviside step function is defined as:

$$\text{heaviside}(\text{input}, \text{values}) = \begin{cases} 0, & \text{if input} < 0 \\ \text{values}, & \text{if input} == 0 \\ 1, & \text{if input} > 0 \end{cases}$$

**Examples**

```
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**

```
if (torch_is_installed()) {
  torch_histc(torch_tensor(c(1., 2, 1)), bins=4, min=0, max=3)
}
```

---

torch_hstack	<i>Hstack</i>
--------------	---------------

---

**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	<i>Hypot</i>
-------------	--------------

---

**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)))
}
```

---

torch_i0	<i>I0</i>
----------	-----------

---

**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	<i>Integer type info</i>
-------------	--------------------------

---

**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	<i>Imag</i>
------------	-------------

---

**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.

$$\text{out}_i = \text{imag}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  ## Not run:  
  torch_imag(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))  
  
  ## End(Not run)  
}
```

---

torch_index	<i>Index torch tensors</i>
-------------	----------------------------

---

**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	<i>Modify values selected by indices.</i>
-----------------	---

---

**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) Wether 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**

```
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	<i>A simple exported version of install_path Returns the torch installation path.</i>
--------------------	---

---

**Description**

A simple exported version of install\_path Returns the torch installation path.

**Usage**

```
torch_install_path()
```

---

torch_inverse	<i>Inverse</i>
---------------	----------------

---

**Description**

Inverse

**Usage**

```
torch_inverse(self)
```

**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**

```
if (torch_is_installed()) {
  ## Not run:
  x = torch_rand(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	<i>Is</i> close
---------------	-----------------

---

**Description**

Is

close
**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 NaN s 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:

$$|\text{input} - \text{other}| \leq \text{atol} + \text{rtol} \times |\text{other}|$$

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

```
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	<i>Isfinite</i>
----------------	-----------------

---

### 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

```
if (torch_is_installed()) {
  torch_isfinite(torch_tensor(c(1, Inf, 2, -Inf, NaN)))
}
```

---

torch_isinf	<i>Isinf</i>
-------------	--------------

---

**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	<i>Isnan</i>
-------------	--------------

---

**Description**

Isnan

**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**

```
if (torch_is_installed()) {
  torch_isnan(torch_tensor(c(1, NaN, 2)))
}
```

---

torch_isneginf	<i>Isneginf</i>
----------------	-----------------

---

**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**

```
if (torch_is_installed()) {
  a <- torch_tensor(c(-Inf, Inf, 1.2))
  torch_isneginf(a)
}
```

---

torch_isposinf	<i>Isposinf</i>
----------------	-----------------

---

**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**

```
if (torch_is_installed()) {  
  
  a <- torch_tensor(c(-Inf, Inf, 1.2))  
  torch_isposinf(a)  
}
```

---

torch_isreal	<i>Isreal</i>
--------------	---------------

---

**Description**

Isreal

**Usage**

```
torch_isreal(self)
```

**Arguments**

self (Tensor) the input tensor.

**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**

```
if (torch_is_installed()) {  
  if (FALSE) {  
    torch_isreal(torch_tensor(c(1, 1+1i, 2+0i)))  
  }  
}
```

---

torch_istft	<i>Istft</i>
-------------	--------------

---

### 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

<code>self</code>	(Tensor) The input tensor. Expected to be output of <code>torch_stft()</code> , can either be complex (channel, fft_size, n_frame), or real (channel, fft_size, n_frame, 2) where the channel dimension is optional.
<code>n_fft</code>	(int) Size of Fourier transform
<code>hop_length</code>	(Optional[int]) The distance between neighboring sliding window frames. (Default: <code>n_fft % 4</code> )
<code>win_length</code>	(Optional[int]) The size of window frame and STFT filter. (Default: <code>n_fft</code> )
<code>window</code>	(Optional(torch.Tensor)) The optional window function. (Default: <code>torch_ones(win_length)</code> )
<code>center</code>	(bool) Whether input was padded on both sides so that the $t$ -th frame is centered at time $t \times \text{hop\_length}$ . (Default: TRUE)
<code>normalized</code>	(bool) Whether the STFT was normalized. (Default: FALSE)
<code>onesided</code>	(Optional(bool)) Whether the STFT was onesided. (Default: TRUE if <code>n_fft != fft_size</code> in the input size)
<code>length</code>	(Optional(int)) The amount to trim the signal by (i.e. the original signal length). (Default: whole signal)
<code>return_complex</code>	(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 <code>onesided=TRUE</code> . (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).

D. W. Griffin and J. S. Lim, "Signal estimation from modified short-time Fourier transform," IEEE Trans. ASSP, vol.32, no.2, pp.236-243, Apr. 1984.

---

<code>torch_is_complex</code>	<i>Is_complex</i>
-------------------------------	-------------------

---

**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  
*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 *Verifies if torch is installed*

---

**Description**

Verifies if torch is installed

**Usage**

torch\_is\_installed()



---

torch_is_nonzero	<i>Is_nonzero</i>
------------------	-------------------

---

**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()`  $\neq$  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	<i>Kaiser_window</i>
---------------------	----------------------

---

**Description**

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	<i>Kronecker product</i>
------------	--------------------------

---

**Description**

Computes the Kronecker product of `self` and `other`.

**Usage**

```
torch_kron(self, other)
```

**Arguments**

<code>self</code>	(Tensor) input Tensor
<code>other</code>	(Tensor) other tensor.

---

torch_kthvalue	<i>Kthvalue</i>
----------------	-----------------

---

**Description**

*Kthvalue*

**Usage**

```
torch_kthvalue(self, k, dim = -1, keepdim = FALSE)
```

**Arguments**

<code>self</code>	(Tensor) the input tensor.
<code>k</code>	(int) k for the k-th smallest element
<code>dim</code>	(int, optional) the dimension to find the kth value along
<code>keepdim</code>	(bool) whether the output tensor has <code>dim</code> 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.

**Examples**

```

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	<i>Creates the corresponding layout</i>
--------------	---

---

**Description**

Creates the corresponding layout

**Usage**

```

torch_strided()

torch_sparse_coo()

```

---

torch_lcm	<i>Lcm</i>
-----------	------------

---

**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  $lcm(0, 0) = 0$  and  $lcm(0, a) = 0$ .

**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_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

**le(input, other, out=NULL) -> Tensor**

Computes  $\text{input} \leq \text{other}$  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_le(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
           torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

---

torch_lerp	<i>Lerp</i>
------------	-------------

---

### 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	<i>Less</i>
------------	-------------

---

**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	<i>Less_equal</i>
------------------	-------------------

---

**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	<i>Lgamma</i>
--------------	---------------

---

**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	<i>Linspace</i>
----------------	-----------------

---

**Description**

Linspace

**Usage**

```
torch_linspace(  
  start,  
  end,  
  steps = 100,  
  dtype = NULL,  
  layout = NULL,  
  device = NULL,  
  requires_grad = FALSE  
)
```



**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.
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.

**linspace(start, end, steps=100, out=NULL, dtype=NULL, layout=torch.strided, device=NULL, requires\_grad=False) -> Tensor**

Returns a one-dimensional tensor of steps equally spaced points between start and end.

The output tensor is 1-D of size steps.

**Examples**

```
if (torch_is_installed()) {
  torch_linspace(3, 10, steps=5)
  torch_linspace(-10, 10, steps=5)
  torch_linspace(start=-10, end=10, steps=5)
  torch_linspace(start=-10, end=10, steps=1)
}
```

---

torch_load	<i>Loads a saved object</i>
------------	-----------------------------

---

**Description**

Loads a saved object

**Usage**

```
torch_load(path, device = "cpu")
```

**Arguments**

path	a path to the saved object
device	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.

**See Also**

Other torch\_save: [torch\\_save\(\)](#), [torch\\_serialize\(\)](#)

---

torch_log	<i>Log</i>
-----------	------------

---

**Description**

Log

**Usage**

```
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**

```
if (torch_is_installed()) {  
  a = torch_randn(c(5))  
  a  
  torch_log(a)  
}
```

---

torch_log10	<i>Log10</i>
-------------	--------------

---

**Description**

Log10

**Usage**

```
torch_log10(self)
```

**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**

```
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 + input).

$$y_i = \log_e(x_i + 1)$$

**Note**

This function is more accurate than [torch\\_log](#) for small values of input

**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	<i>Logaddexp</i>
-----------------	------------------

---

**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**

```
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	<i>Logaddexp2</i>
------------------	-------------------

---

**Description**

Logaddexp2

**Usage**

```
torch_logaddexp2(self, other)
```

**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 *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

$$\text{logcumsumexp}(x)_{ij} = \log \sum_{j=0}^i \exp(x_{ij})$$

**Examples**

```
if (torch_is_installed()) {
  a <- torch_randn(c(10))
  torch_logcumsumexp(a, dim=1)
}
```

---

torch_logdet	<i>Logdet</i>
--------------	---------------

---

## Description

Logdet

## Usage

```
torch_logdet(self)
```

## Arguments

`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

```
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**

```
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	<i>Logical_or</i>
------------------	-------------------

---

**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**

```
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  $[\text{eps}, 1 - \text{eps}]$  when eps is not None. When eps is None and  $\text{input} < 0$  or  $\text{input} > 1$ , the function will yields NaN.

$$y_i = \ln\left(\frac{z_i}{1 - z_i}\right) z_i = \begin{cases} x_i & \text{if eps is None} \\ \text{eps} & \text{if } x_i < \text{eps} \\ x_i & \text{if } \text{eps} \leq x_i \leq 1 - \text{eps} \\ 1 - \text{eps} & \text{if } x_i > 1 - \text{eps} \end{cases}$$

**Examples**

```
if (torch_is_installed()) {
  a <- torch_rand(5)
  a
  torch_logit(a, eps=1e-6)
}
```

---

torch\_logspace      *Logspace*

---

**Description**

Logspace

**Usage**

```
torch_logspace(
  start,
  end,
  steps = 100,
  base = 10,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE
)
```

**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  $\text{base}^{\text{start}}$  and  $\text{base}^{\text{end}}$ .

The output tensor is 1-D of size steps.

**Examples**

```
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

*Logsumexp*

---

**Description**

Logsumexp

**Usage**

```
torch_logsumexp(self, dim, 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.

**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 `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) `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**

```
if (torch_is_installed()) {
  a = torch_randn(c(3, 3))
  torch_logsumexp(a, 1)
}
```

---

torch_lstsq	<i>Lstsq</i>
-------------	--------------

---

**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  $m < n$ , `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

```
torch.lt(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  $\text{input} < \text{other}$  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.lt(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
           torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

---

torch_lu	<i>LU</i>
----------	-----------

---

**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 $(, m, n)$ $(,m,n)$
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	<i>Lu_solve</i>
----------------	-----------------

---

**Description**

Lu\_solve

**Usage**

```
torch_lu_solve(self, LU_data, LU_pivots)
```

**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**

```

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	<i>Lu_unpack</i>
-----------------	------------------

---

**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



**lu\_unpack(LU\_data, LU\_pivots, unpack\_data = TRUE, unpack\_pivots=TRUE) -> Tensor**

Unpacks the data and pivots from a LU factorization of a tensor into tensors L and U and a permutation tensor P such that `LU_data_and_pivots <- torch_lu(P$matmul(L)$matmul(U))`. Returns a list of tensors as `list(the P tensor (permutation matrix), the L tensor, the U tensor)`

---

`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

<code>seed</code>	integer seed.
<code>.env</code>	environment that will take the modifications from <code>manual_seed</code> .
<code>code</code>	expression to run in the context of the seed
<code>...</code>	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

```
if (torch_is_installed()) {  
  x = torch_randn(c(3, 4))  
  x  
  mask = x$ge(0.5)  
  mask  
  torch_masked_select(x, mask)  
}
```

---

torch_matmul	<i>Matmul</i>
--------------	---------------

---

## 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 multiply 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**

```

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	<i>Matrix_exp</i>
------------------	-------------------

---

**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.

### 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)]]  
}
```

---

torch_matrix_power	<i>Matrix_power</i>
--------------------	---------------------

---

### 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	<i>Matrix_rank</i>
-------------------	--------------------

---

**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  $S.\max() * \max(S.\text{size}()) * \text{eps}$  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	<i>Max</i>
-----------	------------

---

**Description**

Max

**Arguments**

self	(Tensor) the input tensor.
dim	(int) the dimension to reduce.
keepdim	(bool) whether the output tensor has <code>dim</code> retained or not. Default: FALSE.
out	(tuple, optional) the result tuple of two output tensors ( <code>max</code> , <code>max_indices</code> )
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**

```
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	<i>Maximum</i>
---------------	----------------

---

**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**

```
if (torch_is_installed()) {  
  
  a <- torch_tensor(c(1, 2, -1))  
  b <- torch_tensor(c(3, 0, 4))  
  torch_maximum(a, b)  
}
```

---

torch_mean	<i>Mean</i>
------------	-------------

---

**Description**

Mean

**Usage**

```
torch_mean(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` 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**

```
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	<i>Median</i>
--------------	---------------

---

**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**

```
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`    *Memory format*

---

**Description**

Returns the correspondent memory format.

**Usage**

```
torch_contiguous_format()  
  
torch_preserve_format()  
  
torch_channels_last_format()
```

---

torch_meshgrid	<i>Meshgrid</i>
----------------	-----------------

---

**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 (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	<i>Min</i>
-----------	------------

---

**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 .

$$\text{out}_i = \min(\text{tensor}_i, \text{other}_i)$$

**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)
```

```
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	<i>Minimum</i>
---------------	----------------

---

## 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

```
if (torch_is_installed()) {
  a <- torch_tensor(c(1, 2, -1))
  b <- torch_tensor(c(3, 0, 4))
  torch_minimum(a, b)
}
```

---

`torch_mm`*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`*Mode*

---

**Description**

Mode

**Usage**`torch_mode(self, dim = -1L, 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.

**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	<i>Movedim</i>
---------------	----------------

---

**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**

```
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

*Mul*


---

**Description**

Mul

**Usage**

```
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**

```

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	<i>Multinomial</i>
-------------------	--------------------

---

**Description**

Multinomial

**Usage**

```
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  $(m \times \text{num\_samples})$ .

If replacement is 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, `num_samples` must be lower than 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

```
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	<i>Multiply</i>
----------------	-----------------

---

### Description

Multiply

### Usage

```
torch_multiply(self, other)
```

### Arguments

self	(Tensor) the first multiplicand tensor
other	(Tensor) the second multiplicand tensor

**multiply(input, other, \*, out=None)**

Alias for [torch\\_mul\(\)](#).

---

`torch_mv`*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`*Mvlgamma*

---

**Description**

Mvlgamma

**Usage**`torch_mvlgamma(self, p)`

**Arguments**

`self` (Tensor) the tensor to compute the multivariate log-gamma function  
`p` (int) the number of dimensions

**mvlgamma(input, p) -> Tensor**

Computes the multivariate log-gamma function <[https://en.wikipedia.org/wiki/Multivariate\\_gamma\\_function](https://en.wikipedia.org/wiki/Multivariate_gamma_function)> with dimension  $p$  element-wise, given by

$$\log(\Gamma_p(a)) = C + \sum_{i=1}^p \log\left(\Gamma\left(a - \frac{i-1}{2}\right)\right)$$

where  $C = \log(\pi) \times \frac{p(p-1)}{4}$  and  $\Gamma(\cdot)$  is the Gamma function.

All elements must be greater than  $\frac{p-1}{2}$ , otherwise an error would be thrown.

**Examples**

```
if (torch_is_installed()) {
  a = torch_empty(c(2, 3))$uniform_(1, 2)
  a
  torch_mvlgamma(a, 2)
}
```

---

torch_nanquantile	<i>Nanquantile</i>
-------------------	--------------------

---

**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**

```
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**

<code>self</code>	(Tensor) the input tensor.
<code>dim</code>	(int or tuple of ints) the dimension or dimensions to reduce.
<code>keepdim</code>	(bool) whether the output tensor has <code>dim</code> retained or not.
<code>dtype</code>	the desired data type of returned tensor. If specified, the input tensor is casted to <code>dtype</code> before the operation is performed. This is useful for preventing data type overflows. Default: <code>NULL</code> .

**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**

```
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	<i>Narrow</i>
--------------	---------------

---

**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.

**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

*Ne*

---

**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*  $\neq$  *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	<i>Neg</i>
-----------	------------

---

**Description**

Neg

**Usage**

torch\_neg(self)

**Arguments**

self (Tensor) the input tensor.

**neg(input, out=NULL) -> 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	<i>Negative</i>
----------------	-----------------

---

**Description**

Negative

**Usage**

torch\_negative(self)

**Arguments**

self (Tensor) the input tensor.

**negative(input, \*, out=None) -> Tensor**Alias for [torch\\_neg\(\)](#)



---

torch_nextafter	<i>Nextafter</i>
-----------------	------------------

---

**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	<i>Nonzero</i>
---------------	----------------

---

**Description**

Nonzero elements of tensors.

**Usage**

```
torch_nonzero(self, as_list = FALSE)
```

**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**

```
if (torch_is_installed()) {
  torch_nonzero(torch_tensor(c(1, 1, 1, 0, 1)))
}
```

---

torch\_norm

*Norm*

---

**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' Frobe-  
 nius norm – 'nuc' nuclear norm – Other as vec norm when dim is NULL sum(abs(x)**ord**)(1./ord)  
 =====

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**

```
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	<i>Normal</i>
--------------	---------------

---

**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 <code>torch_tensor()</code> then the output has the same size as <code>std</code> 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 <code>mean</code> and <code>std</code> are scalars.
generator	a random number generator created with <code>torch_generator()</code> . If NULL a default generator is used.
...	Tensor option parameters like <code>dtype</code> , <code>layout</code> , and <code>device</code> . Can only be used when <code>mean</code> and <code>std</code> are both scalar numerics.

**normal(mean, std, \*) -> Tensor**

Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.

The `mean` is a tensor with the mean of each output element's normal distribution

The `std` is a tensor with the standard deviation of each output element's normal distribution

The shapes of `mean` and `std` don't need to match, but the total number of elements in each tensor need to be the same.

**normal(mean=0.0, std) -> Tensor**

Similar to the function above, but the means are shared among all drawn elements.

**normal(mean, std=1.0) -> Tensor**

Similar to the function above, but the standard-deviations are shared among all drawn elements.

**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 `size`.

**Note**

When the shapes do not match, the shape of `mean` is used as the shape for the returned output tensor

**Examples**

```
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	<i>Not_equal</i>
-----------------	------------------

---

**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	<i>Ones</i>
------------	-------------

---

**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 <a href="#">torch_set_default_tensor_type()</a> ).

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	<i>Ones_like</i>
-----------------	------------------

---

### 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

```
if (torch_is_installed()) {
  input = torch_empty(c(2, 3))
  torch_ones_like(input)
}
```

---

torch_orgqr	<i>Orgqr</i>
-------------	--------------

---

### Description

Orgqr

### Usage

```
torch_orgqr(self, input2)
```

### Arguments

self (Tensor) the a from [torch\\_geqrf](#).

input2 (Tensor) the tau from [torch\\_geqrf](#).

**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 `?orgqr`. See LAPACK documentation for `orgqr_` for further details.

---

torch_ormqr	<i>Ormqr</i>
-------------	--------------

---

**Description**

Ormqr

**Usage**

```
torch_ormqr(self, input2, input3, left = TRUE, transpose = FALSE)
```

**Arguments**

self	(Tensor) the a from <a href="#">torch_geqrf</a> .
input2	(Tensor) the tau from <a href="#">torch_geqrf</a> .
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	<i>Outer</i>
-------------	--------------

---

**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  $(n \times m)$ .



**Note**

This function does not broadcast.

**Examples**

```
if (torch_is_installed()) {
  v1 <- torch_arange(1., 5.)
  v2 <- torch_arange(1., 4.)
  torch_outer(v1, v2)
}
```

---

 torch\_pdist

*Pdist*


---

**Description**

Pdist

**Usage**

```
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')` \*  $M$ . When  $p = \infty$ , the closest scipy function is `scipy.spatial.distance.pdist(xn, lambda x, y: np.abs(x - y).max()`

---

torch_pinverse	<i>Pinverse</i>
----------------	-----------------

---

### 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

```
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     *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**

$\text{math}:(*, C \times r^2, H, W)$  to a :

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**

```
if (torch_is_installed()) {  
  
  input = torch_randn(c(1, 9, 4, 4))  
  output = nnf_pixel_shuffle(input, 3)  
  print(output$size())  
}
```

---

torch\_poisson     *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**

```
if (torch_is_installed()) {
  rates = torch_rand(c(4, 4)) * 5 # rate parameter between 0 and 5
  torch_poisson(rates)
}
```

---

torch_polar	<i>Polar</i>
-------------	--------------

---

**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**

```
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	<i>Polygamma</i>
-----------------	------------------

---

## 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  $n^{\text{th}}$  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

```
if (torch_is_installed()) {  
  ## Not run:  
  a = torch_tensor(c(1, 0.5))  
  torch_polygamma(1, a)  
  
  ## End(Not run)  
}
```

---

 torch\_pow

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:

$$\text{out}_i = \text{self}^{\text{exponent}_i}$$

**Examples**

```

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.

**Examples**

```
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**

```
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**

```
if (torch_is_installed()) {  
  
  torch_promote_types(torch_int32(), torch_float32())  
  torch_promote_types(torch_uint8(), torch_long())  
}
```



---

torch_qr	$Qr$
----------	------

---

**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 $m \times n$ .
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  $\text{input} = QR$  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	<i>Creates the corresponding Scheme object</i>
---------------	--

---

**Description**

Creates the corresponding Scheme object

**Usage**

torch\_per\_channel\_affine()

torch\_per\_tensor\_affine()

torch\_per\_channel\_symmetric()

torch\_per\_tensor\_symmetric()

---

torch_quantile	<i>Quantile</i>
----------------	-----------------

---

**Description**

Quantile

**Usage**

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	The interpolation method.

**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**

```
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.qint8, 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**

```
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**

```
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**

```
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	<i>Rad2deg</i>
---------------	----------------

---

**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	<i>Rand</i>
------------	-------------

---

**Description**

Rand

**Usage**

```
torch_rand(  
  ...,  
  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 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**

```
if (torch_is_installed()) {
  torch_rand(4)
  torch_rand(c(2, 3))
}
```

---

torch_randint	<i>Randint</i>
---------------	----------------

---

**Description**

Randint

**Usage**

```
torch_randint(
  low,
  high,
  size,
  generator = NULL,
  dtype = NULL,
  layout = NULL,
  device = NULL,
  requires_grad = FALSE,
  memory_format = torch_preserve_format()
)
```

**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).

The shape of the tensor is defined by the variable argument size.

.. 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

*Randn*

---

**Description**

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).

$$\text{out}_i \sim \mathcal{N}(0, 1)$$

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	<i>Randn_like</i>
------------------	-------------------

---

**Description**

Randn\_like

**Usage**

```
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\_for  
-> 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	<i>Randperm</i>
----------------	-----------------

---

### Description

Randperm

### Usage

```
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

```
if (torch_is_installed()) {  
  
    torch_randperm(4)  
}
```

---

torch_rand_like	<i>Rand_like</i>
-----------------	------------------

---

**Description**

Rand\_like

**Usage**

```
torch_rand_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.

**rand\_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 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	<i>Range</i>
-------------	--------------

---

**Description**

Range

**Usage**

```
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**

```
if (torch_is_installed()) {  
  
  torch_range(1, 4)  
  torch_range(1, 4, 0.5)  
}
```

---

torch_real	<i>Real</i>
------------	-------------

---

**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.

$$\text{out}_i = \text{real}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  ## Not run:  
  torch_real(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))  
  
  ## End(Not run)  
}
```

---

torch_reciprocal	<i>Reciprocal</i>
------------------	-------------------

---

**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

$$\text{out}_i = \frac{1}{\text{input}_i}$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_reciprocal(a)  
}
```

---

torch_reduction	<i>Creates the reduction objet</i>
-----------------	------------------------------------

---

**Description**

Creates the reduction objet

**Usage**

```
torch_reduction_sum()
```

```
torch_reduction_mean()
```

```
torch_reduction_none()
```

---

torch\_relu

*Relu*

---

### Description

Relu

### Usage

```
torch_relu(self)
```

### Arguments

self            the input tensor

### relu(input) -> Tensor

Computes the relu transformation.

---

torch\_relu\_

*Relu\_*

---

### Description

Relu\_

### Usage

```
torch_relu_(self)
```

### Arguments

self            the input tensor

### relu\_(input) -> Tensor

In-place version of [torch\\_relu\(\)](#).



---

torch_remainder	<i>Remainder</i>
-----------------	------------------

---

**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	<i>Renorm</i>
--------------	---------------

---

**Description**

Renorm

**Usage**

```
torch_renorm(self, p, dim, maxnorm)
```

**Arguments**

self	(Tensor) the input tensor.
p	(float) the power for the norm computation
dim	(int) the dimension to slice over to get the sub-tensors
maxnorm	(float) the maximum norm to keep each sub-tensor under

**renorm(input, p, dim, maxnorm, out=NULL) -> Tensor**

Returns a tensor where each sub-tensor of input along dimension dim is normalized such that the p-norm of the sub-tensor is lower than the value maxnorm

**Note**

If the norm of a row is lower than maxnorm, the row is unchanged

**Examples**

```
if (torch_is_installed()) {
  x = torch_ones(c(3, 3))
  x[2,]$fill_(2)
  x[3,]$fill_(3)
  x
  torch_renorm(x, 1, 1, 5)
}
```

---

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**

```
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	<i>Reshape</i>
---------------	----------------

---

**Description**

Reshape

**Usage**

```
torch_reshape(self, shape)
```

**Arguments**

self	(Tensor) the tensor to be reshaped
shape	(tuple of ints) the new shape

**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**

```
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))
}
```

---

torch_result_type	<i>Result_type</i>
-------------------	--------------------

---

**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**

```
if (torch_is_installed()) {
  torch_result_type(tensor1 = torch_tensor(c(1, 2), dtype=torch_int()), tensor2 = 1)
}
```

---

torch_roll	<i>Roll</i>
------------	-------------

---

**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	<i>Rot90</i>
-------------	--------------

---

**Description**

Rot90

**Usage**

```
torch_rot90(self, k = 1L, dims = c(0, 1))
```

**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**

```
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	<i>Round</i>
-------------	--------------

---

**Description**

Round

**Usage**

```
torch_round(self, decimals)
```

**Arguments**

self (Tensor) the input tensor.  
 decimals 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.

**Examples**

```

if (torch_is_installed()) {

a = torch_randn(c(4))
a
torch_round(a)
}

```

---

torch_rrelu_	<i>Rrelu_</i>
--------------	---------------

---

**Description**

Rrelu\_

**Usage**

```

torch_rrelu_(
  self,
  lower = 0.125,
  upper = 0.3333333333333333,
  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_rsqr	<i>Rsqrt</i>
------------	--------------

---

**Description**

Rsqrt

**Usage**

```
torch_rsqr(self)
```

**Arguments**

self (Tensor) the input tensor.

**rsqr(input, out=NULL) -> Tensor**

Returns a new tensor with the reciprocal of the square-root of each of the elements of input.

$$\text{out}_i = \frac{1}{\sqrt{\text{input}_i}}$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_rsqr(a)  
}
```

---

torch_save	<i>Saves an object to a disk file.</i>
------------	--

---

**Description**

This function is experimental, don't use for long term storage.

**Usage**

```
torch_save(obj, path, ..., compress = TRUE)
```



**Arguments**

obj	the saved object
path	a connection or the name of the file to save.
...	not currently used.
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.

---

torch_searchsorted	<i>Searchsorted</i>
--------------------	---------------------

---

**Description**

Searchsorted

**Usage**

```
torch_searchsorted(
    sorted_sequence,
    self,
    out_int32 = FALSE,
    right = FALSE,
    side = NULL,
    sorter = list()
)
```

**Arguments**

sorted_sequence	(Tensor) N-D or 1-D tensor, containing monotonically increasing sequence on the <i>innermost</i> dimension.
self	(Tensor or Scalar) N-D tensor or a Scalar containing the search value(s).
out_int32	(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> .
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.
side	the same as right but preferred. “left” corresponds to FALSE for right and “right” corresponds to TRUE for right. It will error if this is set to “left” while right is TRUE.
sorter	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.

**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.

**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	<i>Serialize a torch object returning a raw object</i>
-----------------	--

---

**Description**

It's just a wrapper around [torch\\_save\(\)](#).

**Usage**

```
torch_serialize(obj, ...)
```

**Arguments**

obj	the saved object
...	Additional arguments passed to <a href="#">torch_save()</a> . obj and path are not accepted as they are set by <a href="#">torch_serialize()</a> .

**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	<i>Gets and sets the default floating point dtype.</i>
-------------------------	--

---

**Description**

Gets and sets the default floating point dtype.

**Usage**

```
torch_set_default_dtype(d)
```

```
torch_get_default_dtype()
```

**Arguments**

d	The default floating point dtype to set. Initially set to <a href="#">torch_float()</a> .
---	---

---

torch_sgn	<i>Sgn</i>
-----------	------------

---

**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](#)).

$$\text{out}_i = 0, \text{ if } |\text{input}_i| == 0 \text{ out}_i = \frac{\text{input}_i}{|\text{input}_i|}, \text{ otherwise}$$
**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	<i>Sigmoid</i>
---------------	----------------

---

**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.

$$\text{out}_i = \frac{1}{1 + e^{-\text{input}_i}}$$

**Examples**

```
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_sigmoid(a)
}
```

---

 torch\_sign

*Sign*


---

**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.

$$\text{out}_i = \text{sgn}(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {
  a = torch_tensor(c(0.7, -1.2, 0., 2.3))
  a
  torch_sign(a)
}
```

---

torch_signbit	<i>Signbit</i>
---------------	----------------

---

**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**

```
if (torch_is_installed()) {  
  a <- torch_tensor(c(0.7, -1.2, 0., 2.3))  
  torch_signbit(a)  
}
```

---

torch_sin	<i>Sin</i>
-----------	------------

---

**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.

$$\text{out}_i = \sin(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(4))  
  a  
  torch_sin(a)  
}
```

---

torch\_sinh

*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_sinh(a)  
}
```



---

torch_slogdet	<i>Slogdet</i>
---------------	----------------

---

## 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*


---

**Description**

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

**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\_.

**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	<i>Sqrt</i>
------------	-------------

---

**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	<i>Square</i>
--------------	---------------

---

**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**

```
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**

```

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	<i>Stack</i>
-------------	--------------

---

**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	<i>Std</i>
-----------	------------

---

**Description**

Std

**Usage**

```
torch_std(self, dim, unbiased = TRUE, keepdim = FALSE)
```

**Arguments**

<code>self</code>	(Tensor) the input tensor.
<code>dim</code>	(int or tuple of ints) the dimension or dimensions to reduce.
<code>unbiased</code>	(bool) whether to use the unbiased estimation or not
<code>keepdim</code>	(bool) whether the output tensor has <code>dim</code> 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**

```
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)
}
```

---

<code>torch_std_mean</code>	<i>Std_mean</i>
-----------------------------	-----------------

---

**Description**

`Std_mean`

**Usage**

```
torch_std_mean(self, dim, unbiased = TRUE, keepdim = FALSE)
```



**Arguments**

<code>self</code>	(Tensor) the input tensor.
<code>dim</code>	(int or tuple of ints) the dimension or dimensions to reduce.
<code>unbiased</code>	(bool) whether to use the unbiased estimation or not
<code>keepdim</code>	(bool) whether the output tensor has <code>dim</code> 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**

```
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**

<code>input</code>	(Tensor) the input tensor
<code>n_fft</code>	(int) size of Fourier transform
<code>hop_length</code>	(int, optional) the distance between neighboring sliding window frames. Default: NULL (treated as equal to $\text{floor}(n\_fft / 4)$ )
<code>win_length</code>	(int, optional) the size of window frame and STFT filter. Default: NULL (treated as equal to <code>n_fft</code> )
<code>window</code>	(Tensor, optional) the optional window function. Default: NULL (treated as window of all 1 s)
<code>center</code>	(bool, optional) whether to pad input on both sides so that the $t$ -th frame is centered at time $t \times \text{hop\_length}$ . Default: TRUE
<code>pad_mode</code>	(string, optional) controls the padding method used when center is TRUE. Default: "reflect"
<code>normalized</code>	(bool, optional) controls whether to return the normalized STFT results Default: FALSE
<code>onesided</code>	(bool, optional) controls whether to return half of results to avoid redundancy Default: TRUE
<code>return_complex</code>	(bool, optional) controls whether to return complex tensors or not.

**Short-time Fourier transform (STFT).**

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  $0 \leq \omega < n\_fft$ . When `onesided` is the default value `TRUE`,

- \* ``input`` must be either a 1-D time sequence or a 2-D batch of time sequences.
- \* If ``hop_length`` is ``NULL`` (default), it is treated as equal to ``floor(n_fft / 4)``.
- \* If ``win_length`` is ``NULL`` (default), it is treated as equal to ``n_fft``.
- \* ``window`` can be a 1-D tensor of size ``win_length``, e.g., from ``torch_hann_window``. If ``window`` is ``NULL`` (default), it is treated as if having  $\text{eqn}\{1\}$  everywhere in the window. If  $\text{eqn}\{\text{win\_length}\} < \text{eqn}\{n\_fft\}$ , ``window`` will be padded on both sides to length ``n_fft`` before being applied.
- \* If ``center`` is ``TRUE`` (default), ``input`` will be padded on both sides so that the  $\text{eqn}\{t\}$ -th frame is centered at time  $\text{eqn}\{t \times \text{hop\_length}\}$ . Otherwise, the  $\text{eqn}\{t\}$ -th frame begins at time  $\text{eqn}\{t \times \text{hop\_length}\}$ .
- \* ``pad_mode`` determines the padding method used on ``input`` when ``center`` is ``TRUE``. See ``torch.nn.functional.pad`` for all available options. Default is ``"reflect"``.
- \* If ``onesided`` is ``TRUE`` (default), only values for  $\text{eqn}\{\omega\}$  in  $\text{eqn}\{\left[0, 1, 2, \dots, \left\lfloor \frac{n\_fft}{2} \right\rfloor + 1\right]\}$  are returned because the real-to-complex Fourier transform satisfies the conjugate symmetry, i.e.,  $\text{eqn}\{X[m, \omega] = X[m, n\_fft - \omega]^*\}$ .
- \* If ``normalized`` is ``TRUE`` (default is ``FALSE``), the function returns the normalized STFT results, i.e., multiplied by  $\text{eqn}\{(\text{frame\_length})^{-0.5}\}$ .

Returns the real and the imaginary parts together as one tensor of size  $\text{eqn}\{(* \times N \times T \times 2)\}$ , where  $\text{eqn}\{*\}$  is the optional batch size of ``input``,  $\text{eqn}\{N\}$  is the number of frequencies where STFT is applied,  $\text{eqn}\{T\}$  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.

### 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

**sub(input, other, \*, alpha=1, out=None) -> Tensor**

Subtracts other, scaled by alpha, from input.

$$\text{out}_i = \text{input}_i - \text{alpha} \times \text{other}_i$$

Supports broadcasting to a common shape , type promotion , and integer, float, and complex inputs.

**Examples**

```
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)
```

**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	<i>Sum</i>
-----------	------------

---

**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).

**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**

```
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$t()))
  a_big = torch_randn(c(7, 5, 3))
  out = torch_svd(a_big)
  u = out[[1]]
  s = out[[2]]
  v = out[[3]]
  torch_dist(a_big, torch_matmul(torch_matmul(u, torch_diag_embed(s)), v$transpose(-2, -1)))
}
```

---

 torch\_t

*T*


---

**Description**

T

**Usage**

`torch_t(self)`

**Arguments**

`self` (Tensor) the input tensor.

**t(input) -> Tensor**

Expects input to be  $\leq$  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

*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.



**Examples**

```
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()))
}
```

---

`torch_take_along_dim` *Selects values from input at the 1-dimensional indices from indices along the given dim.*

---

**Description**

Selects values from input at the 1-dimensional indices from indices along the given dim.

**Usage**

```
torch_take_along_dim(self, indices, dim = NULL)
```

**Arguments**

<code>self</code>	the input tensor.
<code>indices</code>	the indices into input. Must have long dtype.
<code>dim</code>	the dimension to select along. Default is NULL.

**Note**

If `dim` is NULL, the input array is treated as if it has been flattened to 1d.

Functions that return indices along a dimension, like `torch_argmax()` and `torch_argsort()`, are designed to work with this function. See the examples below.

**Examples**

```
if (torch_is_installed()) {
  t <- torch_tensor(matrix(c(10, 30, 20, 60, 40, 50), nrow = 2))
  max_idx <- torch_argmax(t)
  torch_take_along_dim(t, max_idx)

  sorted_idx <- torch_argsort(t, dim=2)
  torch_take_along_dim(t, sorted_idx, dim=2)
}
```

---

`torch_tan`*Tan*

---

**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`.

$$\text{out}_i = \tan(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {  
  a = torch_randn(c(4))  
  a  
  torch_tan(a)  
}
```

---

`torch_tanh`*Tanh*

---

**Description**

Tanh

**Usage**`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.

$$\text{out}_i = \tanh(\text{input}_i)$$

**Examples**

```
if (torch_is_installed()) {
  a = torch_randn(c(4))
  a
  torch_tanh(a)
}
```

---

torch_tensor	<i>Converts R objects to a torch tensor</i>
--------------	---

---

**Description**

Converts R objects to a torch tensor

**Usage**

```
torch_tensor(
  data,
  dtype = NULL,
  device = NULL,
  requires_grad = FALSE,
  pin_memory = FALSE
)
```

**Arguments**

data	an R atomic vector, matrix or array
dtype	a <a href="#">torch_dtype</a> instance
device	a device created with <a href="#">torch_device()</a>
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**

```
if (torch_is_installed()) {
  torch_tensor(c(1, 2, 3, 4))
  torch_tensor(c(1, 2, 3, 4), dtype = torch_int())
}
```

---

torch_tensordot	<i>Tensordot</i>
-----------------	------------------

---

**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**

```
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**

```
torch_tensor_from_buffer(buffer, shape, dtype = "float")
```

```
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_	<i>Threshold_</i>
------------------	-------------------

---

**Description**

Threshold\_

**Usage**

```
torch_threshold_(self, threshold, value)
```

**Arguments**

self	input tensor
threshold	The value to threshold at
value	The value to replace with

**threshold\_(input, threshold, value) -> Tensor**

In-place version of torch\_threshold.

---

`torch_topk`*Topk*

---

### Description

Topk

### Usage

```
torch_topk(self, k, dim = -1, largest = TRUE, sorted = TRUE)
```

### Arguments

<code>self</code>	(Tensor) the input tensor.
<code>k</code>	(int) the k in "top-k"
<code>dim</code>	(int, optional) the dimension to sort along
<code>largest</code>	(bool, optional) controls whether to return largest or smallest elements
<code>sorted</code>	(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	<i>Trace</i>
-------------	--------------

---

**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	<i>Transpose</i>
-----------------	------------------

---

**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

**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 out tensor shares it's underlying storage with the input tensor, so changing the content of one would change the content of the other.

**Examples**

```
if (torch_is_installed()) {
  x = torch_randn(c(2, 3))
  x
  torch_transpose(x, 1, 2)
}
```

---

 torch\_trapz

*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 $\int_a^b f = -\int_b^a f$ 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 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.



**Examples**

```

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)

}

```

---

```

torch_triangular_solve
      Triangular_solve

```

---

**Description**

Triangular\_solve

**Usage**

```

torch_triangular_solve(
  self,
  A,
  upper = TRUE,
  transpose = FALSE,
  unittriangular = FALSE
)

```

**Arguments**

self	(Tensor) multiple right-hand sides of size $(*, m, k)$ where $*$ is zero or 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.
unittriangular	(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.

`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

```
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

```
torch_tril(self, 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.

**Examples**

```

if (torch_is_installed()) {

a = torch_randn(c(3, 3))
a
torch_tril(a)
b = torch_randn(c(4, 6))
b
torch_tril(b, diagonal=1)
torch_tril(b, diagonal=-1)
}

```

---

torch\_tril\_indices      *Tril\_indices*

---

**Description**

Tril\_indices

**Usage**

```

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.

**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

```
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

**triu(input, diagonal=0, out=NULL) -> Tensor**

Returns the upper triangular part of a matrix (2-D tensor) or batch of matrices input, the other elements of the result tensor out are set to 0.

The upper triangular part of the matrix is defined as the elements on and above the diagonal.

The argument `diagonal` controls which diagonal to consider. If `diagonal = 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.

**Examples**

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(3, 3))  
  a  
  torch_triu(a)  
  torch_triu(a, diagonal=1)  
  torch_triu(a, diagonal=-1)  
  b = torch_randn(c(4, 6))  
  b  
  torch_triu(b, diagonal=1)  
  torch_triu(b, diagonal=-1)  
}
```

---

torch\_triu\_indices      *Triu\_indices*

---

**Description**

Triu\_indices

**Usage**

```
torch_triu_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.

**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**

```
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	<i>TRUE_divide</i>
-------------------	--------------------

---

**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	<i>Trunc</i>
-------------	--------------

---

**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**

```
if (torch_is_installed()) {  
  
  a = torch_randn(c(4))  
  a  
  torch_trunc(a)  
}
```

---

torch_unbind	<i>Unbind</i>
--------------	---------------

---

**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**

```
if (torch_is_installed()) {  
  
  torch_unbind(torch_tensor(matrix(1:9, ncol = 3, byrow=TRUE)))  
}
```



---

`torch_unique_consecutive`*Unique\_consecutive*

---

## Description

`Unique_consecutive`

## Usage

```
torch_unique_consecutive(  
    self,  
    return_inverse = FALSE,  
    return_counts = FALSE,  
    dim = NULL  
)
```

## Arguments

<code>self</code>	(Tensor) the input tensor
<code>return_inverse</code>	(bool) Whether to also return the indices for where elements in the original input ended up in the returned unique list.
<code>return_counts</code>	(bool) Whether to also return the counts for each unique element.
<code>dim</code>	(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

```
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	<i>Unsqueeze</i>
-----------------	------------------

---

**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	<i>Vander</i>
--------------	---------------

---

**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  $x^{(N-1)}, x^{(N-2)}, \dots, x^0$ . If increasing is TRUE, the order of the columns is reversed  $x^0, x^1, \dots, x^{(N-1)}$ . 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	<i>Var</i>
-----------	------------

---

**Description**

Var

**Usage**

```
torch_var(self, dim, unbiased = TRUE, keepdim = FALSE)
```

**Arguments**

<code>self</code>	(Tensor) the input tensor.
<code>dim</code>	(int or tuple of ints) the dimension or dimensions to reduce.
<code>unbiased</code>	(bool) whether to use the unbiased estimation or not
<code>keepdim</code>	(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**

```
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	<i>Var_mean</i>
----------------	-----------------

---

**Description**

Var\_mean

**Usage**

```
torch_var_mean(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.

**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**

```

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)
```

**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)
  }
}
```

---

`torch_view_as_complex` *View\_as\_complex*

---

**Description**

`View_as_complex`

**Usage**

```
torch_view_as_complex(self)
```

**Arguments**

`self` (Tensor) the input tensor.

**view\_as\_complex(input) -> Tensor**

Returns a view of `input` as a complex tensor. For an input complex tensor of size  $m_1, m_2, \dots, m_i, 2$ , this function returns a new complex tensor of size  $m_1, m_2, \dots, m_i$  where the last dimension of the input tensor is expected to represent the real and imaginary components of complex numbers.

**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**

```
if (torch_is_installed()) {
  if (FALSE) {
    x=torch_randn(c(4, 2))
    x
    torch_view_as_complex(x)
  }
}
```

---

torch_view_as_real	<i>View_as_real</i>
--------------------	---------------------

---

**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  $m_1, m_2, \dots, m_i$ , this function returns a new real tensor of size  $m_1, m_2, \dots, m_i, 2$ , 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**

```
if (torch_is_installed()) {

  if (FALSE) {
    x <- torch_randn(4, dtype=torch_cfloat())
    x
    torch_view_as_real(x)
  }
}
```



```
}  
}
```

---

torch_vstack	<i>Vstack</i>
--------------	---------------

---

### 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	<i>Where</i>
-------------	--------------

---

### Description

Where

### Usage

```
torch_where(condition, self = NULL, other = NULL)
```

**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**

```
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)

}
```

---

 torch\_zeros

*Zeros*


---

**Description**

Zeros

**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

**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

```
with_detect_anomaly(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

```
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	<i>Enable grad</i>
------------------	--------------------

---

## Description

Context-manager that enables gradient calculation. Enables gradient calculation, if it has been disabled via [with\\_no\\_grad](#).

## Usage

```
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

```
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	<i>Temporarily modify gradient recording.</i>
--------------	---

---

**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, 56
- \* **distributions**
  - distr\_bernoulli, 40
  - distr\_chi2, 42
  - distr\_gamma, 42
  - distr\_multivariate\_normal, 44
  - distr\_normal, 45
  - distr\_poisson, 46
- \* **linalg**
  - linalg\_cholesky, 62
  - linalg\_cholesky\_ex, 63
  - linalg\_det, 66
  - linalg\_eig, 67
  - linalg\_eigh, 68
  - linalg\_eigvals, 70
  - linalg\_eigvalsh, 71
  - linalg\_householder\_product, 72
  - linalg\_inv, 73
  - linalg\_inv\_ex, 74
  - linalg\_lstsq, 75
  - linalg\_matrix\_norm, 77
  - linalg\_matrix\_power, 79
  - linalg\_matrix\_rank, 80
  - linalg\_multi\_dot, 81
  - linalg\_norm, 82
  - linalg\_pinv, 84
  - linalg\_qr, 85
  - linalg\_slogdet, 86
  - linalg\_solve, 87
  - linalg\_solve\_triangular, 88
  - linalg\_svd, 89
  - linalg\_svdvals, 91
  - linalg\_tensorinv, 92
  - linalg\_tensorsolve, 93
  - linalg\_vector\_norm, 94
- \* **serialization**
  - load\_state\_dict, 96
  - torch\_load, 457
  - torch\_save, 536
  - torch\_serialize, 540
- \* **tensor-attributes**
  - is\_torch\_device, 52
  - is\_torch\_dtype, 52
  - torch\_dtype, 399
  - torch\_finfo, 416
  - torch\_iinfo, 437
  - torch\_qscheme, 514
  - torch\_reduction, 527
  - torch\_set\_default\_dtype, 540
- \* **tensor-attributes**
  - torch\_device, 389
- \* **torch\_save**
  - torch\_load, 457
  - torch\_save, 536
  - torch\_serialize, 540
- as\_array, 17
- autograd\_backward, 19
- autograd\_function, 20
- autograd\_grad, 21
- autograd\_set\_grad\_mode, 23
- AutogradContext, 17, 21
- backends\_cudnn\_is\_available, 23
- backends\_cudnn\_version, 23
- backends\_mkl\_is\_available, 24
- backends\_mkldnn\_is\_available, 24
- backends\_mps\_is\_available, 24
- backends\_openmp\_is\_available, 25
- broadcast\_all, 25
- buffer\_from\_torch\_tensor
  - (torch\_tensor\_from\_buffer), 564
- clone\_module, 26
- Constraint, 26
- contrib\_sort\_vertices, 27
- cuda\_amp\_grad\_scaler, 28
- cuda\_amp\_grad\_scaler(), 97



- cuda\_current\_device, 29
- cuda\_current\_device(), 32
- cuda\_device\_count, 29
- cuda\_empty\_cache, 29
- cuda\_empty\_cache(), 29
- cuda\_get\_device\_capability, 30
- cuda\_get\_rng\_state
  - (torch\_get\_rng\_state), 429
- cuda\_is\_available, 30
- cuda\_memory\_stats, 30
- cuda\_memory\_summary
  - (cuda\_memory\_stats), 30
- cuda\_runtime\_version, 32
- cuda\_set\_rng\_state
  - (torch\_get\_rng\_state), 429
- cuda\_synchronize, 32
  
- dataloader, 32
- dataloader(), 35, 36, 319
- dataloader\_make\_iter, 34, 35
- dataloader\_next, 35
- dataset, 35
- dataset(), 34, 319, 320
- dataset\_subset, 36
- distr\_bernoulli, 40, 42, 43, 45, 46
- distr\_categorical, 41
- distr\_chi2, 40, 42, 43, 45, 46
- distr\_gamma, 40, 42, 44, 45, 46
- distr\_mixture\_same\_family, 43
- distr\_multivariate\_normal, 40, 42, 43, 44, 46
- distr\_normal, 40, 42, 43, 45, 45, 46
- distr\_poisson, 40, 42, 43, 45, 46, 46
- Distribution, 37, 40, 42, 43, 45, 46
  
- enumerate, 47
- enumerate.dataloader, 47
  
- get\_install\_libs\_url, 48
  
- install\_torch, 49
- install\_torch\_from\_file
  - (get\_install\_libs\_url), 48
- is\_dataloader, 50
- is\_nn\_buffer, 50
- is\_nn\_module, 51
- is\_nn\_parameter, 51
- is\_optimizer, 51
- is\_torch\_device, 52
- is\_torch\_dtype, 52
- is\_torch\_layout, 52
- is\_torch\_memory\_format, 53
- is\_torch\_qscheme, 53
- is\_undefined\_tensor, 53
- iterable\_dataset, 54
  
- jit\_compile, 55, 60
- jit\_load, 55
- jit\_ops, 56
- jit\_save, 57
- jit\_save(), 56
- jit\_save\_for\_mobile, 57
- jit\_scalar, 58
- jit\_serialize, 58, 62
- jit\_trace, 59, 61
- jit\_trace(), 60
- jit\_trace\_module, 60
- jit\_trace\_module(), 59, 60
- jit\_tuple, 61
- jit\_unserialize, 58, 62
  
- linalg\_cholesky, 62, 64, 66, 68, 70–75, 77–79, 81–83, 85–89, 91–95
- linalg\_cholesky(), 63, 64, 69
- linalg\_cholesky\_ex, 63, 63, 66, 68, 70–75, 77–79, 81–83, 85–89, 91–95
- linalg\_cholesky\_ex(), 63
- linalg\_cond, 64
- linalg\_det, 63, 64, 66, 68, 70–75, 77–79, 81–83, 85–89, 91–95
- linalg\_eig, 63, 64, 66, 67, 70–75, 77–79, 81–83, 85–89, 91–95
- linalg\_eig(), 68, 69, 71, 91
- linalg\_eigh, 63, 64, 66, 68, 68, 71–75, 77–79, 81–83, 85–89, 91–95
- linalg\_eigh(), 63, 68, 69, 72, 84, 91
- linalg\_eigvals, 63, 64, 66, 68, 70, 70, 72–75, 77–79, 81–83, 85–89, 91–95
- linalg\_eigvals(), 68
- linalg\_eigvalsh, 63, 64, 66, 68, 70, 71, 71, 73–75, 77–79, 81–83, 85–89, 91–95
- linalg\_eigvalsh(), 69
- linalg\_householder\_product, 63, 64, 66, 68, 70–72, 72, 74, 75, 77–79, 81–83, 85–89, 91–95
- linalg\_inv, 63, 64, 66, 68, 70–73, 73, 75, 77–79, 81–83, 85–89, 91–95
- linalg\_inv(), 65, 75, 85

- `linalg_inv_ex`, [63](#), [64](#), [66](#), [68](#), [70–74](#), [74](#),  
[77–79](#), [81–83](#), [85–89](#), [91–95](#)
- `linalg_lstsq`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [75](#), [78](#),  
[79](#), [81–83](#), [85–89](#), [91–95](#)
- `linalg_lstsq()`, [84](#), [85](#)
- `linalg_matrix_norm`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77](#), [77](#), [79](#), [81–83](#), [85–89](#), [91–95](#)
- `linalg_matrix_power`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77](#), [78](#), [79](#), [81–83](#), [85–89](#), [91–95](#)
- `linalg_matrix_rank`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [80](#), [82](#), [83](#), [85–89](#), [91–95](#)
- `linalg_multi_dot`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81](#), [81](#), [83](#), [85–89](#), [91–95](#)
- `linalg_norm`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [77–79](#),  
[81](#), [82](#), [82](#), [85–89](#), [91–95](#)
- `linalg_norm()`, [65](#), [95](#)
- `linalg_pinv`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [77–79](#),  
[81–83](#), [84](#), [86–89](#), [91–95](#)
- `linalg_pinv()`, [74](#)
- `linalg_qr`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [77–79](#),  
[81–83](#), [85](#), [85](#), [87–89](#), [91–95](#)
- `linalg_qr()`, [68](#), [70](#), [73](#), [91](#)
- `linalg_slogdet`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81–83](#), [85](#), [86](#), [86](#), [88](#), [89](#),  
[91–95](#)
- `linalg_solve`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [77–79](#),  
[81–83](#), [85–87](#), [87](#), [89](#), [91–95](#)
- `linalg_solve()`, [74](#), [79](#)
- `linalg_solve_triangular`, [63](#), [64](#), [66](#), [68](#),  
[70–75](#), [77–79](#), [81–83](#), [85–88](#), [88](#),  
[91–95](#)
- `linalg_svd`, [63](#), [64](#), [66](#), [68](#), [70–75](#), [77–79](#),  
[81–83](#), [85–89](#), [89](#), [92–95](#)
- `linalg_svd()`, [65](#), [68](#), [70](#), [84](#), [90](#), [92](#)
- `linalg_svdvals`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81–83](#), [85–89](#), [91](#), [91](#), [93–95](#)
- `linalg_svdvals()`, [90](#)
- `linalg_tensorinv`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81–83](#), [85–89](#), [91](#), [92](#), [92](#), [94](#),  
[95](#)
- `linalg_tensorinv()`, [94](#)
- `linalg_tensorsolve`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81–83](#), [85–89](#), [91–93](#), [93](#), [95](#)
- `linalg_tensorsolve()`, [92](#), [93](#)
- `linalg_vector_norm`, [63](#), [64](#), [66](#), [68](#), [70–75](#),  
[77–79](#), [81–83](#), [85–89](#), [91–94](#), [94](#)
- `load_state_dict`, [96](#)
- `local_autocast`, [96](#)
- `local_device`, [98](#)
- `local_enable_grad` (`with_enable_grad`),  
[590](#)
- `local_no_grad` (`with_no_grad`), [591](#)
- `local_torch_manual_seed`  
(`torch_manual_seed`), [473](#)
- `lr_cosine_annealing`, [99](#)
- `lr_lambda`, [99](#)
- `lr_multiplicative`, [100](#)
- `lr_one_cycle`, [101](#)
- `lr_reduce_on_plateau`, [103](#)
- `lr_scheduler`, [105](#)
- `lr_step`, [105](#)
- `nn_adaptive_avg_pool1d`, [170](#)
- `nn_adaptive_avg_pool1d()`, [323](#)
- `nn_adaptive_avg_pool2d`, [170](#)
- `nn_adaptive_avg_pool3d`, [171](#)
- `nn_adaptive_log_softmax_with_loss`, [172](#)
- `nn_adaptive_max_pool1d`, [174](#)
- `nn_adaptive_max_pool2d`, [174](#)
- `nn_adaptive_max_pool3d`, [175](#)
- `nn_avg_pool1d`, [176](#)
- `nn_avg_pool1d()`, [348](#)
- `nn_avg_pool2d`, [177](#)
- `nn_avg_pool3d`, [179](#)
- `nn_batch_norm1d`, [180](#)
- `nn_batch_norm2d`, [182](#)
- `nn_batch_norm3d`, [183](#)
- `nn_bce_loss`, [185](#)
- `nn_bce_with_logits_loss`, [186](#)
- `nn_bilinear`, [188](#)
- `nn_buffer`, [189](#)
- `nn_buffer()`, [257](#)
- `nn_celu`, [189](#)
- `nn_contrib_sparsemax`, [190](#)
- `nn_conv1d`, [190](#)
- `nn_conv1d()`, [373](#)
- `nn_conv2d`, [192](#), [201](#), [210](#), [211](#)
- `nn_conv2d()`, [374](#)
- `nn_conv3d`, [195](#)
- `nn_conv3d()`, [375](#)
- `nn_conv_transpose1d`, [197](#)
- `nn_conv_transpose1d()`, [377](#)
- `nn_conv_transpose2d`, [199](#), [201](#)
- `nn_conv_transpose2d()`, [378](#)
- `nn_conv_transpose3d`, [202](#)
- `nn_conv_transpose3d()`, [379](#)
- `nn_cosine_embedding_loss`, [205](#)

nn\_cross\_entropy\_loss, 205  
nn\_ctc\_loss, 207  
nn\_dropout, 209  
nn\_dropout2d, 210, 210  
nn\_dropout3d, 211, 211  
nn\_elu, 212  
nn\_embedding, 212, 213  
nn\_embedding\_bag, 214  
nn\_flatten, 215  
nn\_fractional\_max\_pool2d, 216  
nn\_fractional\_max\_pool3d, 217  
nn\_gelu, 218  
nn\_glu, 219  
nn\_group\_norm, 219  
nn\_gru, 221  
nn\_hardshrink, 223  
nn\_hardsigmoid, 223  
nn\_hardswish, 224  
nn\_hardtanh, 225  
nn\_hinge\_embedding\_loss, 226  
nn\_identity, 227  
nn\_init\_calculate\_gain, 227  
nn\_init\_constant\_, 228  
nn\_init\_dirac\_, 228  
nn\_init\_eye\_, 229  
nn\_init\_kaiming\_normal\_, 229  
nn\_init\_kaiming\_uniform\_, 230  
nn\_init\_normal\_, 231  
nn\_init\_ones\_, 232  
nn\_init\_orthogonal\_, 232  
nn\_init\_sparse\_, 233  
nn\_init\_trunc\_normal\_, 233  
nn\_init\_uniform\_, 234  
nn\_init\_xavier\_normal\_, 235  
nn\_init\_xavier\_uniform\_, 235  
nn\_init\_zeros\_, 236  
nn\_kl\_div\_loss, 236  
nn\_l1\_loss, 237  
nn\_layer\_norm, 238  
nn\_leaky\_relu, 240  
nn\_linear, 241  
nn\_log\_sigmoid, 242  
nn\_log\_softmax, 242  
nn\_log\_softmax(), 205  
nn\_lp\_pool1d, 243  
nn\_lp\_pool2d, 244  
nn\_lstm, 246  
nn\_margin\_ranking\_loss, 248  
nn\_max\_pool1d, 249  
nn\_max\_pool1d(), 253  
nn\_max\_pool2d, 250  
nn\_max\_pool2d(), 254  
nn\_max\_pool3d, 251  
nn\_max\_pool3d(), 256  
nn\_max\_unpool1d, 253  
nn\_max\_unpool1d(), 174  
nn\_max\_unpool2d, 254  
nn\_max\_unpool2d(), 175, 217  
nn\_max\_unpool3d, 255  
nn\_max\_unpool3d(), 175, 217  
nn\_module, 26, 256  
nn\_module(), 59  
nn\_module\_dict, 258  
nn\_module\_dict(), 259  
nn\_module\_list, 259, 259  
nn\_module\_list(), 259  
nn\_mse\_loss, 260  
nn\_multi\_margin\_loss, 265  
nn\_multihead\_attention, 261  
nn\_multilabel\_margin\_loss, 263  
nn\_multilabel\_soft\_margin\_loss, 264  
nn\_nll\_loss, 266  
nn\_nll\_loss(), 237  
nn\_pairwise\_distance, 267  
nn\_pairwise\_distance(), 168, 289  
nn\_parameter, 268  
nn\_parameter(), 257  
nn\_poisson\_nll\_loss, 269  
nn\_prelu, 270  
nn\_prune\_head, 271  
nn\_relu, 272  
nn\_relu6, 272  
nn\_rnn, 273  
nn\_rrelu, 275  
nn\_selu, 276  
nn\_sequential, 215, 277, 290  
nn\_sigmoid, 278  
nn\_silu, 278  
nn\_silu(), 162  
nn\_smooth\_l1\_loss, 279  
nn\_soft\_margin\_loss, 284  
nn\_softmax, 280  
nn\_softmax2d, 281  
nn\_softmin, 281  
nn\_softplus, 282  
nn\_softshrink, 283

- nn\_softsign, 284
- nn\_tanh, 285
- nn\_tanhshrink, 286
- nn\_threshold, 286
- nn\_triplet\_margin\_loss, 287
- nn\_triplet\_margin\_loss(), 289
- nn\_triplet\_margin\_with\_distance\_loss, 288
- nn\_triplet\_margin\_with\_distance\_loss(), 168, 288
- nn\_unflatten, 216, 290
- nn\_upsample, 291
- nn\_utils\_clip\_grad\_norm\_, 292
- nn\_utils\_clip\_grad\_value\_, 293
- nn\_utils\_rnn\_pack\_padded\_sequence, 293
- nn\_utils\_rnn\_pack\_padded\_sequence(), 222, 247, 295, 296
- nn\_utils\_rnn\_pack\_sequence, 294
- nn\_utils\_rnn\_pack\_sequence(), 247, 296
- nn\_utils\_rnn\_pad\_packed\_sequence, 295
- nn\_utils\_rnn\_pad\_sequence, 296
- nn\_utils\_weight\_norm, 297
- nnf\_adaptive\_avg\_pool1d, 106
- nnf\_adaptive\_avg\_pool2d, 106
- nnf\_adaptive\_avg\_pool3d, 107
- nnf\_adaptive\_max\_pool1d, 107
- nnf\_adaptive\_max\_pool2d, 108
- nnf\_adaptive\_max\_pool3d, 108
- nnf\_affine\_grid, 109
- nnf\_affine\_grid(), 109, 134
- nnf\_alpha\_dropout, 109
- nnf\_avg\_pool1d, 110
- nnf\_avg\_pool2d, 110
- nnf\_avg\_pool3d, 111
- nnf\_batch\_norm, 112
- nnf\_bilinear, 113
- nnf\_binary\_cross\_entropy, 113
- nnf\_binary\_cross\_entropy\_with\_logits, 114
- nnf\_celu, 115
- nnf\_celu(), 362
- nnf\_celu\_ (nnf\_celu), 115
- nnf\_contrib\_sparsemax, 115
- nnf\_conv1d, 116
- nnf\_conv2d, 116
- nnf\_conv3d, 117
- nnf\_conv\_tbc, 118
- nnf\_conv\_transpose1d, 119
- nnf\_conv\_transpose2d, 120
- nnf\_conv\_transpose3d, 121
- nnf\_cosine\_embedding\_loss, 122
- nnf\_cosine\_similarity, 122
- nnf\_cross\_entropy, 123
- nnf\_ctc\_loss, 124
- nnf\_dropout, 125
- nnf\_dropout2d, 125
- nnf\_dropout3d, 126
- nnf\_elu, 126
- nnf\_elu\_ (nnf\_elu), 126
- nnf\_embedding, 127
- nnf\_embedding\_bag, 128
- nnf\_fold, 129
- nnf\_fractional\_max\_pool2d, 130
- nnf\_fractional\_max\_pool3d, 131
- nnf\_gelu, 132
- nnf\_glu, 132
- nnf\_grid\_sample, 133
- nnf\_grid\_sample(), 109
- nnf\_group\_norm, 134
- nnf\_gumbel\_softmax, 135
- nnf\_hardshrink, 135
- nnf\_hardsigmoid, 136
- nnf\_hardswish, 136
- nnf\_hardtanh, 137
- nnf\_hardtanh\_ (nnf\_hardtanh), 137
- nnf\_hinge\_embedding\_loss, 137
- nnf\_instance\_norm, 138
- nnf\_interpolate, 138
- nnf\_interpolate(), 133
- nnf\_kl\_div, 140
- nnf\_l1\_loss, 140
- nnf\_layer\_norm, 141
- nnf\_leaky\_relu, 141
- nnf\_linear, 142
- nnf\_local\_response\_norm, 142
- nnf\_log\_softmax, 124, 143
- nnf\_logsigmoid, 143
- nnf\_lp\_pool1d, 144
- nnf\_lp\_pool2d, 144
- nnf\_margin\_ranking\_loss, 145
- nnf\_max\_pool1d, 145
- nnf\_max\_pool2d, 146
- nnf\_max\_pool3d, 147
- nnf\_max\_unpool1d, 147
- nnf\_max\_unpool2d, 148
- nnf\_max\_unpool3d, 149

- nnf\_mse\_loss, 149
- nnf\_multi\_head\_attention\_forward, 151
- nnf\_multi\_margin\_loss, 153
- nnf\_multilabel\_margin\_loss, 150
- nnf\_multilabel\_soft\_margin\_loss, 150
- nnf\_nll\_loss, 154
- nnf\_normalize, 155
- nnf\_one\_hot, 155
- nnf\_pad, 156
- nnf\_pairwise\_distance, 157
- nnf\_pdist, 157
- nnf\_pixel\_shuffle, 158
- nnf\_poisson\_nll\_loss, 158
- nnf\_prelu, 159
- nnf\_relu, 159
- nnf\_relu6, 160
- nnf\_relu\_(nnf\_relu), 159
- nnf\_rrelu, 160
- nnf\_rrelu\_(nnf\_rrelu), 160
- nnf\_selu, 161
- nnf\_selu\_(nnf\_selu), 161
- nnf\_sigmoid, 161
- nnf\_silu, 162
- nnf\_smooth\_l1\_loss, 162
- nnf\_soft\_margin\_loss, 166
- nnf\_softmax, 163, 164
- nnf\_softmin, 163
- nnf\_softplus, 164
- nnf\_softshrink, 165
- nnf\_softsign, 165
- nnf\_tanhshrink, 166
- nnf\_threshold, 167
- nnf\_threshold\_(nnf\_threshold), 167
- nnf\_triplet\_margin\_loss, 167
- nnf\_triplet\_margin\_with\_distance\_loss, 168
- nnf\_unfold, 169
  
- optim\_adadelta, 304
- optim\_adagrad, 305
- optim\_adam, 306
- optim\_adamw, 307
- optim\_asgd, 308
- optim\_ignite\_adagrad, 309
- optim\_ignite\_adam, 310
- optim\_ignite\_adamw, 311
- optim\_ignite\_rmsprop, 312
- optim\_ignite\_sgd, 313
- optim\_lbfgs, 314
  
- optim\_required, 315
- optim\_rmsprop, 316
- optim\_rprop, 317
- optim\_sgd, 318
- optimizer, 300
- optimizer\_ignite, 303
- OptimizerIgnite, 301, 309–313
  
- R6::R6Class(), 35, 54, 105, 257
  
- sampler, 319
- sampler(), 33, 34
- set\_autocast(local\_autocast), 96
  
- tensor\_dataset, 320
- threads, 320
- torch\_abs, 321
- torch\_abs(), 321
- torch\_absolute, 321
- torch\_acos, 322
- torch\_acos(), 336
- torch\_acosh, 322
- torch\_acosh(), 336
- torch\_adaptive\_avg\_pool1d, 323
- torch\_add, 324
- torch\_addbmm, 325
- torch\_addcdiv, 326
- torch\_addcmul, 327
- torch\_addmm, 328
- torch\_addmv, 329
- torch\_addr, 330
- torch\_allclose, 331
- torch\_amax, 332
- torch\_amin, 333
- torch\_angle, 334
- torch\_arange, 334, 526
- torch\_arccos, 336
- torch\_arccosh, 336
- torch\_arcsin, 337
- torch\_arcsinh, 337
- torch\_arctan, 338
- torch\_arctanh, 338
- torch\_argmax, 339
- torch\_argmax(), 561
- torch\_argmin, 340
- torch\_argsort, 341
- torch\_argsort(), 561
- torch\_as\_strided, 343
- torch\_asin, 341

`torch_asin()`, 337  
`torch_asinh`, 342  
`torch_asinh()`, 337  
`torch_atan`, 344  
`torch_atan()`, 338  
`torch_atan2`, 344  
`torch_atanh`, 345  
`torch_atanh()`, 338  
`torch_atleast_1d`, 346  
`torch_atleast_2d`, 347  
`torch_atleast_2d()`, 585  
`torch_atleast_3d`, 347  
`torch_atleast_3d()`, 398  
`torch_avg_pool1d`, 348  
`torch_baddbmm`, 349  
`torch_bartlett_window`, 350  
`torch_bernoulli`, 351  
`torch_bincount`, 352  
`torch_bitwise_and`, 353  
`torch_bitwise_not`, 353  
`torch_bitwise_or`, 354  
`torch_bitwise_xor`, 354  
`torch_blackman_window`, 355  
`torch_block_diag`, 356  
`torch_bmm`, 356  
`torch_bool (torch_dtype)`, 399  
`torch_broadcast_tensors`, 357  
`torch_bucketize`, 358  
`torch_can_cast`, 359  
`torch_cartesian_prod`, 359  
`torch_cat`, 360, 360  
`torch_cdist`, 361  
`torch_cdouble (torch_dtype)`, 399  
`torch_ceil`, 361  
`torch_celu`, 362  
`torch_celu()`, 363  
`torch_celu_`, 363  
`torch_cfloat (torch_dtype)`, 399  
`torch_cfloat128 (torch_dtype)`, 399  
`torch_cfloat32 (torch_dtype)`, 399  
`torch_cfloat64 (torch_dtype)`, 399  
`torch_chain_matmul`, 363  
`torch_chalf (torch_dtype)`, 399  
`torch_channel_shuffle`, 364  
`torch_channels_last_format (torch_memory_format)`, 482  
`torch_cholesky`, 365  
`torch_cholesky_inverse`, 366  
`torch_cholesky_solve`, 367  
`torch_chunk`, 360, 368  
`torch_chunk()`, 578  
`torch_clamp`, 368  
`torch_clamp()`, 370  
`torch_clip`, 369  
`torch_clone`, 370  
`torch_combinations`, 371  
`torch_complex`, 371  
`torch_conj`, 372  
`torch_contiguous_format (torch_memory_format)`, 482  
`torch_conv1d`, 373  
`torch_conv2d`, 374  
`torch_conv3d`, 375  
`torch_conv_tbc`, 376  
`torch_conv_transpose1d`, 376  
`torch_conv_transpose2d`, 377  
`torch_conv_transpose3d`, 379  
`torch_cos`, 380  
`torch_cosh`, 381  
`torch_cosine_similarity`, 381  
`torch_count_nonzero`, 382  
`torch_cross`, 383  
`torch_cummax`, 384  
`torch_cummin`, 384  
`torch_cumprod`, 385  
`torch_cumsum`, 386  
`torch_deg2rad`, 387  
`torch_dequantize`, 387  
`torch_det`, 388  
`torch_device`, 389  
`torch_device()`, 563  
`torch_diag`, 389  
`torch_diag_embed`, 392  
`torch_diagflat`, 390  
`torch_diagonal`, 391  
`torch_diff`, 393  
`torch_diff()`, 393  
`torch_digamma`, 394  
`torch_dist`, 394  
`torch_div`, 395, 421, 575  
`torch_div()`, 326, 397  
`torch_divide`, 396  
`torch_dot`, 397  
`torch_double (torch_dtype)`, 399  
`torch_dstack`, 398  
`torch_dtype`, 399, 563

torch\_eig, 400  
torch\_einsum, 400  
torch\_empty, 402  
torch\_empty\_like, 403  
torch\_empty\_strided, 404  
torch\_eq, 405  
torch\_equal, 406  
torch\_erf, 406  
torch\_erfc, 407  
torch\_erfinv, 408  
torch\_exp, 408  
torch\_exp2, 409  
torch\_expm1, 410  
torch\_eye, 410  
torch\_fft\_fft, 411  
torch\_fft\_fft(), 413, 416  
torch\_fft\_fftfreq, 412  
torch\_fft\_ifft, 413  
torch\_fft\_irfft, 414  
torch\_fft\_irfft(), 415  
torch\_fft\_rfft, 415  
torch\_fft\_rfft(), 414  
torch\_finfo, 416  
torch\_finfo(), 80  
torch\_fix, 416  
torch\_flatten, 417  
torch\_flip, 417  
torchfliplr, 418  
torch\_flipud, 419  
torch\_float(torch\_dtype), 399  
torch\_float16(torch\_dtype), 399  
torch\_float32(torch\_dtype), 399  
torch\_float64(torch\_dtype), 399  
torch\_floor, 420  
torch\_floor\_divide, 420  
torch\_floor\_divide(), 326, 396  
torch\_fmod, 421  
torch\_frac, 422  
torch\_full, 422  
torch\_full\_like, 423  
torch\_gather, 424  
torch\_gcd, 425  
torch\_ge, 426  
torch\_ge(), 430  
torch\_generator, 427  
torch\_generator(), 500  
torch\_geqrf, 427, 503, 504  
torch\_geqrf(), 73, 504  
torch\_ger, 428  
torch\_get\_default\_dtype  
(torch\_set\_default\_dtype), 540  
torch\_get\_default\_dtype(), 412  
torch\_get\_num\_interop\_threads  
(threads), 320  
torch\_get\_num\_threads(threads), 320  
torch\_get\_rng\_state, 429  
torch\_greater, 429  
torch\_greater\_equal, 430  
torch\_gt, 430  
torch\_gt(), 430  
torch\_half(torch\_dtype), 399  
torch\_hanning\_window, 431  
torch\_hann\_window, 432  
torch\_heaviside, 433  
torch\_histc, 434  
torch\_hstack, 435  
torch\_hypot, 435  
torch\_i0, 436  
torch\_i0(), 450  
torch\_iinfo, 437  
torch\_imag, 437  
torch\_index, 438  
torch\_index\_put, 438  
torch\_index\_put\_, 439  
torch\_index\_select, 439  
torch\_install\_path, 440  
torch\_int(torch\_dtype), 399  
torch\_int16(torch\_dtype), 399  
torch\_int32(torch\_dtype), 399  
torch\_int32(), 358, 538  
torch\_int64(torch\_dtype), 399  
torch\_int64(), 358, 538  
torch\_int8(torch\_dtype), 399  
torch\_inverse, 440  
torch\_is\_complex, 447  
torch\_is\_floating\_point, 448  
torch\_is\_installed, 448  
torch\_is\_nonzero, 449  
torch\_isclose, 441  
torch\_isfinite, 442  
torch\_isinf, 443  
torch\_isnan, 443  
torch\_isneginf, 444  
torch\_isposinf, 444  
torch\_isreal, 445  
torch\_istft, 446

torch\_kaiser\_window, 449  
torch\_kron, 451  
torch\_kthvalue, 451  
torch\_layout, 452  
torch\_lcm, 452  
torch\_le, 453  
torch\_le(), 455  
torch\_lerp, 454  
torch\_less, 455  
torch\_less\_equal, 455  
torch\_lgamma, 456  
torch\_linspace, 456  
torch\_load, 457, 537, 540  
torch\_load(), 540  
torch\_log, 458, 459  
torch\_log10, 458  
torch\_log1p, 459  
torch\_log2, 460  
torch\_logaddexp, 461  
torch\_logaddexp(), 462  
torch\_logaddexp2, 461  
torch\_logcumsumexp, 462  
torch\_logdet, 463  
torch\_logical\_and, 464  
torch\_logical\_not, 464  
torch\_logical\_or, 465  
torch\_logical\_xor, 466  
torch\_logit, 466  
torch\_logspace, 467  
torch\_logsumexp, 468  
torch\_logsumexp(), 461  
torch\_long (torch\_dtype), 399  
torch\_lstsq, 469  
torch\_lstsq(), 469, 470  
torch\_lt, 470  
torch\_lt(), 455  
torch\_lu, 471  
torch\_lu\_solve, 471  
torch\_lu\_unpack, 472  
torch\_manual\_seed, 473  
torch\_manual\_seed(), 429  
torch\_masked\_select, 474  
torch\_matmul, 357, 475, 486  
torch\_matrix\_exp, 476  
torch\_matrix\_power, 477  
torch\_matrix\_rank, 478  
torch\_max, 478  
torch\_maximum, 480  
torch\_maximum(), 480  
torch\_mean, 480  
torch\_median, 481  
torch\_memory\_format, 482  
torch\_meshgrid, 483  
torch\_min, 483  
torch\_minimum, 485  
torch\_minimum(), 485  
torch\_mm, 486  
torch\_mm(), 82  
torch\_mode, 486  
torch\_movedim, 487  
torch\_mul, 488  
torch\_mul(), 490  
torch\_multinomial, 489  
torch\_multinomial(), 41  
torch\_multiply, 490  
torch\_mv, 491  
torch\_mvlgamma, 491  
torch\_nanquantile, 492  
torch\_nansum, 493  
torch\_narrow, 494  
torch\_ne, 495  
torch\_ne(), 501  
torch\_neg, 496  
torch\_neg(), 496  
torch\_negative, 496  
torch\_nextafter, 497  
torch\_nonzero, 497  
torch\_nonzero(), 586  
torch\_norm, 498  
torch\_normal, 499  
torch\_not\_equal, 501  
torch\_ones, 501  
torch\_ones\_like, 502  
torch\_orgqr, 503  
torch\_ormqr, 504  
torch\_ormqr(), 73  
torch\_outer, 504  
torch\_pdist, 505  
torch\_per\_channel\_affine  
    (torch\_kscheme), 514  
torch\_per\_channel\_symmetric  
    (torch\_kscheme), 514  
torch\_per\_tensor\_affine  
    (torch\_kscheme), 514  
torch\_per\_tensor\_symmetric  
    (torch\_kscheme), 514



- torch\_pinverse, 506
- torch\_pixel\_shuffle, 507
- torch\_poisson, 507
- torch\_polar, 508
- torch\_polygamma, 509
- torch\_pow, 510
- torch\_preserve\_format
  - (torch\_memory\_format), 482
- torch\_preserve\_format(), 370
- torch\_prod, 511
- torch\_promote\_types, 512
- torch\_qint32 (torch\_dtype), 399
- torch\_qint8 (torch\_dtype), 399
- torch\_qr, 428, 513
- torch\_qscheme, 514
- torch\_quantile, 514
- torch\_quantile(), 493
- torch\_quantize\_per\_channel, 515
- torch\_quantize\_per\_tensor, 516
- torch\_quint8 (torch\_dtype), 399
- torch\_rad2deg, 517
- torch\_rand, 517
- torch\_rand\_like, 524
- torch\_randint, 518
- torch\_randint\_like, 519
- torch\_randn, 520
- torch\_randn\_like, 522
- torch\_randperm, 523
- torch\_range, 525
- torch\_real, 526
- torch\_reciprocal, 527
- torch\_reduction, 527
- torch\_reduction\_mean (torch\_reduction), 527
- torch\_reduction\_none (torch\_reduction), 527
- torch\_reduction\_sum (torch\_reduction), 527
- torch\_relu, 528
- torch\_relu(), 528
- torch\_relu\_, 528
- torch\_remainder, 529
- torch\_renorm, 529
- torch\_repeat\_interleave, 530
- torch\_reshape, 531
- torch\_result\_type, 532
- torch\_roll, 533
- torch\_rot90, 533
- torch\_round, 534
- torch\_rrelu\_, 535
- torch\_rsqr, 536
- torch\_save, 458, 536, 540
- torch\_save(), 540
- torch\_scalar\_tensor, 537
- torch\_searchsorted, 538
- torch\_selu, 539
- torch\_selu(), 540
- torch\_selu\_, 539
- torch\_serialize, 458, 537, 540
- torch\_serialize(), 540
- torch\_set\_default\_dtype, 540
- torch\_set\_num\_interop\_threads
  - (threads), 320
- torch\_set\_num\_threads (threads), 320
- torch\_set\_rng\_state
  - (torch\_get\_rng\_state), 429
- torch\_sgn, 541
- torch\_short (torch\_dtype), 399
- torch\_sigmoid, 541
- torch\_sign, 541, 542
- torch\_signbit, 543
- torch\_sin, 543
- torch\_sinh, 544
- torch\_slogdet, 545
- torch\_sort, 546
- torch\_sparse\_coo (torch\_layout), 452
- torch\_sparse\_coo\_tensor, 546
- torch\_split, 548
- torch\_split(), 360, 579
- torch\_sqrt, 549
- torch\_square, 549
- torch\_squeeze, 451, 469, 479, 481, 482, 484, 487, 494, 511, 515, 550, 552, 553, 557, 581, 582
- torch\_squeeze(), 333
- torch\_stack, 551
- torch\_std, 551
- torch\_std\_mean, 552
- torch\_stft, 553
- torch\_stft(), 446, 447, 450
- torch\_strided (torch\_layout), 452
- torch\_strided(), 413
- torch\_sub, 556
- torch\_sub(), 557
- torch\_subtract, 556
- torch\_sum, 557

`torch_svd`, 558  
`torch_t`, 559  
`torch_take`, 560  
`torch_take_along_dim`, 561  
`torch_tan`, 562  
`torch_tanh`, 562  
`torch_tensor`, 563  
`torch_tensor()`, 499, 500  
`torch_tensor_from_buffer`, 564  
`torch_tensordot`, 564  
`torch_tensordot()`, 92, 94  
`torch_threshold_`, 565  
`torch_topk`, 566  
`torch_trace`, 567  
`torch_transpose`, 567  
`torch_trapz`, 568  
`torch_triangular_solve`, 569  
`torch_tril`, 570  
`torch_tril_indices`, 571  
`torch_triu`, 572  
`torch_triu_indices`, 573  
`torch_true_divide`, 575  
`torch_true_divide()`, 326, 396  
`torch_trunc`, 575  
`torch_trunc()`, 416  
`torch_uint8(torch_dtype)`, 399  
`torch_unbind`, 576  
`torch_unique_consecutive`, 577  
`torch_unsafe_chunk`, 578  
`torch_unsafe_split`, 578  
`torch_unsqueeze`, 579  
`torch_vander`, 580  
`torch_var`, 580  
`torch_var_mean`, 581  
`torch_vdot`, 582  
`torch_view_as_complex`, 583, 584  
`torch_view_as_real`, 584  
`torch_view_as_real()`, 584  
`torch_vstack`, 585  
`torch_where`, 585  
`torch_zeros`, 586  
`torch_zeros_like`, 587  
  
`unset_autocast (local_autocast)`, 96  
  
`with_autocast (local_autocast)`, 96  
`with_detect_anomaly`, 589  
`with_device (local_device)`, 98  
`with_enable_grad`, 590  
  
`with_no_grad`, 590, 591  
`with_torch_manual_seed`  
    (`torch_manual_seed`), 473