Package 'torch'

August 6, 2020

Type Package

Title Tensors and Neural Networks with 'GPU' Acceleration

Version 0.0.1

Description Provides functionality to define and train neural networks similar to 'PyTorch' by Paszke et al (2019) <arXiv:1912.01703> but written entirely in R using the 'libtorch' library. Also supports low-level tensor operations and 'GPU' acceleration.

License MIT + file LICENSE

URL http://mlverse.github.io/torch, https://github.com/mlverse/torch

BugReports https://github.com/mlverse/torch/issues

Encoding UTF-8

LazyData true

SystemRequirements C++11, LibTorch (https://pytorch.org/)

LinkingTo Rcpp

Imports Rcpp, R6, withr, rlang, methods, utils, stats

RoxygenNote 7.1.1

Suggests testthat (>= 2.1.0), covr, knitr, rmarkdown, bit64, magrittr, glue

VignetteBuilder knitr

Collate 'R7.R' 'RcppExports.R' 'tensor.R' 'autograd.R' 'backends.R' 'codegen-utils.R' 'conditions.R' 'creation-ops.R' 'cuda.R' 'device.R' 'dimname_list.R' 'dtype.R' 'gen-method.R' 'gen-namespace-docs.R' 'gen-namespace-examples.R' 'gen-namespace.R' 'generator.R' 'help.R' 'indexing.R' 'install.R' 'lantern_load.R' 'lantern_sync.R' 'layout.R' 'memory_format.R' 'utils-data.R' 'nn.R' 'nn-activation.R' 'nn-batchnorm.R' 'nn-conv.R' 'nn-dropout.R' 'nn-init.R' 'nn-linear.R' 'nn-loss.R' 'nn-pooling.R' 'nn-rnn.R' 'nn-sparse.R' 'nn-utils-rnn.R' 'nn-utils.R' 'nn_adaptive.R' 'nnf-activation.R' 'nnf-batchnorm.R' 'nnf-conv.R' 'nnf-distance.R' 'nnf-dropout.R' 'nnf-embedding.R' 'nnf-fold.R'

'nnf-instancenorm.R' 'nnf-linear.R' 'nnf-loss.R'
'nnf-normalization.R' 'nnf-padding.R' 'nnf-pixelshuffle.R'
'nnf-pooling.R' 'nnf-upsampling.R' 'nnf-vision.R' 'operators.R'
'optim.R' 'optim-adam.R' 'optim-sgd.R' 'package.R' 'qscheme.R'
'quantization.R' 'reduction.R' 'save.R' 'scalar.R' 'storage.R'
'tensor_list.R' 'tensor_options.R' 'utils-data-collate.R'
'utils-data-dataloader.R' 'utils-data-enum.R'
'utils-data-fetcher.R' 'utils-data-sampler.R' 'utils.R'
'variable_list.R' 'with-indices.R' 'wrapers.R'

NeedsCompilation yes

Author Daniel Falbel [aut, cre, cph],
Javier Luraschi [aut, cph],
Dmitriy Selivanov [ctb],
Athos Damiani [ctb],
RStudio [cph]

Maintainer Daniel Falbel <daniel@rstudio.com>

Repository CRAN

Date/Publication 2020-08-06 14:40:02 UTC

R topics documented:

`as_array` *Converts to array*

Description

Converts to array

Usage

```
as_array(x)
```

Arguments

`x`	object to be converted into an array

`AutogradContext` *Class representing the context.*

Description

Class representing the context.

Class representing the context.

Public fields

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

Active bindings

`needs_input_grad` boolean listing arguments of `forward` and whether they require_grad.

`saved_variables` list of objects that were saved for backward via `save_for_backward`.

Methods

Public methods:

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

Method `new()`**:** (Dev related) Initializes the context. Not user related.

Usage:

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

Arguments:

`ptr` pointer to the c++ object

`env` environment that encloses both forward and backward

`argument_names` names of forward arguments

`argument_needs_grad` whether each argument in forward needs grad.

Method `save_for_backward()`**:** Saves given objects for a future call to backward().

This should be called at most once, and only from inside the `forward()` method.

Later, saved objects can be accessed through the `saved_variables` attribute. Before returning them to the user, a check is made to ensure they weren't used in any in-place operation that modified their content.

Arguments can also be any kind of R object.

Usage:

```
AutogradContext$save_for_backward(...)
```

Arguments:

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

Method `mark_non_differentiable()`**:** Marks outputs as non-differentiable.

This should be called at most once, only from inside the `forward()` method, and all arguments should be outputs.

This will mark outputs as not requiring gradients, increasing the efficiency of backward 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.r.t. each element of corresponding tensors. NULLvalues can be specified for scalar Tensors or ones that don't require grad. If aNULL' 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 `ctx` as the first argument, followed by any number of arguments (tensors or other types). The context can be used to store tensors that can be then retrieved during the backward pass. See AutogradContext for more information about context methods.
`backward`	Defines a formula for differentiating the operation. It must accept a context `ctx` as the first argument, followed by as many outputs did `forward()` return, and it should return a named list. Each argument is the gradient w.r.t the given output, and each element in the returned list should be the gradient w.r.t. the corresponding input. The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx$needs_input_grad` as a named list of booleans representing whether each input needs gradient. E.g., `backward()` will have `ctx$needs_input_grad$input = TRUE` if the `input` argument to `forward()` needs gradient computated w.r.t. the output. See AutogradContext 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 higher order derivative products. Default: FALSE'.
allow_unused	(bool, optional) – If FALSE, specifying inputs that were not used when computing outputs (and therefore their grad is always zero) is an error. 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.

---

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_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()
```

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 = 0,
  worker_init_fn = NULL
)
```

Arguments

dataset	(Dataset): dataset from which to load the data.
batch_size	(int, optional): how many samples per batch to load (default: 1).
shuffle	(bool, optional): set to TRUE to have the data reshuffled at every epoch (default: FALSE).
sampler	(Sampler, optional): defines the strategy to draw samples from the dataset. If specified, shuffle must be False.
batch_sampler	(Sampler, optional): like sampler, but returns a batch of indices at a time. Mutually exclusive with batch_size, shuffle, sampler, and drop_last.
num_workers	(int, optional): how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)
collate_fn	(callable, optional): merges a list of samples to form a mini-batch.
pin_memory	(bool, optional): If TRUE, the data loader will copy tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your collate_fn returns a batch that is a custom type see the example below.
drop_last	(bool, optional): set to TRUE to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If FALSE and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: FALSE)
timeout	(numeric, optional): if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: 0)
worker_init_fn	(callable, optional): If not NULL, this will be called on each worker subprocess with the worker id (an int in [0, num_workers - 1]) as input, after seeding and before data loading. (default: NULL)

---

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 *Get the next element of a dataloader iterator*

Description

Get the next element of a dataloader iterator

Usage

```
dataloader_next(iter)
```

Arguments

iter	a DataLoader iter created with dataloader_make_iter.

dataset *An abstract class representing a* Dataset.

Description

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `get_item`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `lenght`, which is expected to return the size of the dataset by many `~torch.utils.data.Sampler` implementations and the default options of `~torch.utils.data.DataLoader`.

Usage

```
dataset(name = NULL, inherit = Dataset, ..., 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
parent_env	An environment to use as the parent of newly-created objects.

Note

`~torch.utils.data.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.

enumerate *Enumerate an iterator*

Description

Enumerate an iterator

Usage

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

Arguments

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

enumerate.dataloader *Enumerate an iterator*

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.

install_torch *Install Torch*

Description

Installs Torch and its dependencies.

Usage

```
install_torch(
  version = "1.5.0",
  type = install_type(version = version),
  reinstall = FALSE,
  path = NULL,
  ...
)
```

Arguments

version	The Torch version to install.
type	The installation type for Torch. Valid values are "cpu" or the 'CUDA' version.
reinstall	Re-install Torch even if its already installed?
path	Optional path to install or check for an already existing installation.
...	other optional arguments (like load for manual installation.)

Details

When using path to install in a specific location, make sure the TORCH_HOME environment variable is set to this same path to reuse this installation. 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 troubleshootinng only.

is_dataloader *Checks if the object is a dataloader*

Description

Checks if the object is a dataloader

Usage

```
is_dataloader(x)
```

Arguments

x	object to check

is_torch_dtype *Check if object is a torch data type*

Description

Check if object is a torch data type

Usage

```
is_torch_dtype(x)
```

Arguments

x	object to check.

is_torch_layout *Check if an object is a torch layout.*

Description

Check if an object is a torch layout.

Usage

```
is_torch_layout(x)
```

Arguments

x	object to check

is_torch_memory_format *Check if an object is a memory format*

Description

Check if an object is a memory format

Usage

```
is_torch_memory_format(x)
```

Arguments

x	object to check

is_torch_qscheme *Checks if an object is a QScheme*

Description

Checks if an object is a QScheme

Usage

```
is_torch_qscheme(x)
```

Arguments

x	object to check

nnf_adaptive_avg_pool1d *Adaptive_avg_pool1d*

Description

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

Usage

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

Arguments

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

nnf_adaptive_avg_pool2d *Adaptive_avg_pool2d*

Description

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

Usage

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

Arguments

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

nnf_adaptive_avg_pool3d *Adaptive_avg_pool3d*

Description

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

Usage

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

Arguments

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

`nnf_adaptive_max_pool1d` *Adaptive_max_pool1d*

Description

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

Usage

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

Arguments

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

`nnf_adaptive_max_pool2d` *Adaptive_max_pool2d*

Description

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

Usage

```
nnf_adaptive_max_pool2d(input, output_size, return_indices = FALSE)
```

Arguments

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

nnf_adaptive_max_pool3d *Adaptive_max_pool3d*

Description

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

Usage

```
nnf_adaptive_max_pool3d(input, output_size, return_indices = FALSE)
```

Arguments

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

nnf_affine_grid *Affine_grid*

Description

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

Usage

```
nnf_affine_grid(theta, size, align_corners = FALSE)
```

Arguments

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

Note

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

nnf_alpha_dropout *Alpha_dropout*

Description

Applies alpha dropout to the input.

Usage

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

Arguments

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

nnf_avg_pool1d *Avg_pool1d*

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 *Avg_pool2d*

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 *Avg_pool3d*

Description

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

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 *Batch_norm*

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: $y = x_1 A x_2 + b$

Usage

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

Arguments

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

Value

output $(N, *, H_{out})$ where $H_{out} =$ 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 *Celu*

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_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 $(\text{sequence length} \times batch \times \text{in\_channels})$
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

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

`nnf_conv_transpose2d` *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 x1 and x2, computed along dim.

Usage

```
nnf_cosine_similarity(x1, x2, dim = 1, 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

Details

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

---

nnf_cross_entropy *Cross_entropy*

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

`nnf_ctc_loss` *Ctc_loss*

Description

The Connectionist Temporal Classification loss.

Usage

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

Arguments

`log_probs`	(T, N, C) where C = number of characters in alphabet including blank, T = input length, and N = batch size. The logarithmized probabilities of the outputs (e.g. obtained with nnf_log_softmax).
`targets`	(N, S) or `(sum(target_lengths))`. Targets cannot be blank. In the second form, the targets are assumed to be concatenated.
`input_lengths`	(N). Lengths of the inputs (must each be $\leq T$)
`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 *Dropout*

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 *Dropout2d*

Description

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

Usage

```
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 *Dropout3d*

Description

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

Usage

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

Arguments

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

nnf_elu *Elu*

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` *Embedding_bag*

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
)
```

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.

---

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

Warning

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

`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

`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 $k * k$) or a tuple (kH, kW)
`output_size`	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 $oH * oH$
`output_ratio`	If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range (0, 1)
`return_indices`	if `True`, will return the indices along with the outputs.
`random_samples`	optional random samples.

Details

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

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

Arguments

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

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, σ is the sigmoid function and $\otimes$ is the element-wise product between matrices.

See Language Modeling with Gated Convolutional Networks.

`nnf_grid_sample` *Grid_sample*

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

Details

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

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

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

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

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

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

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

Note

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

`nnf_group_norm` *Group_norm*

Description

Applies Group Normalization for last certain number of dimensions.

Usage

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

Arguments

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

nnf_gumbel_softmax *Gumbel_softmax*

Description

Samples from the Gumbel-Softmax distribution and optionally discretizes.

Usage

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

Arguments

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

nnf_hardshrink *Hardshrink*

Description

Applies the hard shrinkage function element-wise

Usage

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

Arguments

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

nnf_hardsigmoid *Hardsigmoid*

Description

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

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 *Hardtanh*

Description

Applies the HardTanh function element-wise.

Usage

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

nnf_hardtanh_(input, min_val = -1, max_val = 1)
```

Arguments

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

nnf_hinge_embedding_loss *Hinge_embedding_loss*

Description

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

Usage

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

Arguments

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

nnf_instance_norm *Instance_norm*

Description

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

Usage

```
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 *independent* of input size when scale_factor is kept the same. This only has an effect when mode is 'linear', 'bilinear', 'bicubic' or 'trilinear'. Default: False
recompute_scale_factor	(bool, optional) recompute the scale_factor for use in the interpolation calculation. When scale_factor is passed as a parameter, it is used to compute the output_size. If recompute_scale_factor is "'True" or not specified, a new scale_factor will be computed based on the output and input sizes for use in the interpolation computation (i.e. the computation will be identical to if the computed 'output_size' were passed-in explicitly). Otherwise, the passed-in 'scale_factor' will be used in the interpolation computation. Note that when 'scale_factor' is floating-point, the recomputed scale_factor may differ from the one passed in due to rounding and precision issues.

Details

The algorithm used for interpolation is determined by mode.

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

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

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

nnf_kl_div *Kl_div*

Description

The Kullback-Leibler divergence Loss.

Usage

```
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 *Layer_norm*

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 *Leaky_relu*

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 *Linear*

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 *Local_response_norm*

Description

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

Usage

```
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 *Logsigmoid*

Description

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

Usage

```
nnf_logsigmoid(input)
```

Arguments

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

nnf_log_softmax *Log_softmax*

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(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 *Lp_pool1d*

Description

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

Usage

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

Arguments

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

nnf_lp_pool2d *Lp_pool2d*

Description

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

Usage

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

Arguments

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

nnf_margin_ranking_loss *Margin_ranking_loss*

Description

Creates a criterion that measures the loss given inputs x1 , x2 , 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 *Max_pool2d*

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 *Max_pool3d*

Description

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

Usage

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

Arguments

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

nnf_max_unpool1d *Max_unpool1d*

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 *Max_unpool2d*

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 *Max_unpool3d*

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 *Mse_loss*

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, 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'

`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 = NULL,
  need_weights = TRUE,
  attn_mask = NULL,
  use_separate_proj_weight = FALSE,
  q_proj_weight = NULL,
  k_proj_weight = NULL,
  v_proj_weight = NULL,
  static_k = NULL,
  static_v = NULL
)
```

Arguments

`query`	(L, N, E) where L is the target sequence length, N is the batch size, E is the embedding dimension.
`key`	(S, N, E), where S is the source sequence length, N is the batch size, E is the embedding dimension.
`value`	(S, N, E) where S is the source sequence length, N is the batch size, E is the embedding dimension.

`embed_dim_to_check`
: total dimension of the model.

`num_heads`
: parallel attention heads.

`in_proj_weight`
: input projection weight and bias.

`in_proj_bias`
: currently undocumented.

`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 and bias.

`out_proj_bias`
: currently undocumented.

`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.

`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.

nnf_multi_margin_loss *Multi_margin_loss*

Description

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch Tensor) and output y (which is a 1D tensor of target class indices, 0 <= y <= x$size(2) - 1).

Usage

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

---

nnf_normalize *Normalize*

Description

Performs L_p normalization of inputs over specified dimension.

Usage

```
nnf_normalize(input, p = 2, dim = 1, eps = 1e-12, out = NULL)
```

Arguments

input	input tensor of any shape
p	(float) the exponent value in the norm formulation. Default: 2
dim	(int) the dimension to reduce. Default: 1
eps	(float) small value to avoid division by zero. Default: 1e-12
out	(Tensor, optional) the output tensor. If out is used, this operation won't be differentiable.

Details

For a tensor `input` of sizes $(n_0, ..., n_{dim}, ..., n_k)$, each n_{dim} -element vector v along dimension `dim` is transformed as

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}.$$

With the default arguments it uses the Euclidean norm over vectors along dimension 1 for normalization.

---

nnf_one_hot *One_hot*

Description

Takes LongTensor with index values of shape (*) and returns a tensor of shape (*, num_classes) that have zeros everywhere except where the index of last dimension matches the corresponding value of the input tensor, in which case it will be 1.

Usage

```
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 *Pad*

Description

Pads tensor.

Usage

```
nnf_pad(input, pad, mode = "constant", value = 0)
```

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. $\left\lfloor \frac{\text{len(pad)}}{2} \right\rfloor$ dimensions of input will be padded. For example, to pad only the last dimension of the input tensor, then pad has the form $(\text{padding\_left}, \text{padding\_right})$;
to pad the last 2 dimensions of the input tensor, then use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom})$;
to pad the last 3 dimensions, use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom padding\_front}, \text{padding\_back})$.

Padding mode

See nn_constant_pad_2d, nn_reflection_pad_2d, and nn_replication_pad_2d for concrete examples on how each of the padding modes works. Constant padding is implemented for arbitrary dimensions. tensor, or the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor. Reflect padding is only implemented for padding the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor.

---

nnf_pairwise_distance *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 *Pixel_shuffle*

Description

Rearranges elements in a tensor of shape $(*, C \times r^2, H, W)$ to a tensor of shape $(*, C, H \times r, W \times r)$.

Usage

```
nnf_pixel_shuffle(input, upscale_factor)
```

Arguments

input	(Tensor) the input tensor
upscale_factor	(int) factor to increase spatial resolution by

nnf_poisson_nll_loss *Poisson_nll_loss*

Description

Poisson negative log likelihood loss.

Usage

```
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 log_input=FALSE. Default: 1e-8
reduction	(string, optional) – Specifies the reduction to apply to the output: 'none' | 'mean' | 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. 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 wether it's a training pass. DEfault: FALSE
inplace	can optionally do the operation in-place. Default: FALSE

nnf_selu *Selu*

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_smooth_l1_loss` *Smooth_l1_loss*

Description

Function that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise.

Usage

```
nnf_smooth_l1_loss(input, target, reduction = "mean")
```

Arguments

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

`nnf_softmax` *Softmax*

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:

$$Softmax(x_i) = exp(x_i)/\sum_j exp(x_j)$$

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 *Softmin*

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

$$Softmin(x) = Softmax(-x)$$

.

See nnf_softmax definition for mathematical formula.

nnf_softplus *Softplus*

Description

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

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 $input*\beta > threshold$.

nnf_softshrink *Softshrink*

Description

Applies the soft shrinkage function elementwise

Usage

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

Arguments

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

nnf_softsign *Softsign*

Description

Applies element-wise, the function $SoftSign(x) = x/(1 + |x|$

Usage

```
nnf_softsign(input)
```

Arguments

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

nnf_soft_margin_loss *Soft_margin_loss*

Description

Creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1).

Usage

```
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 *Threshold*

Description

Thresholds each element of the input Tensor.

Usage

```
nnf_threshold(input, threshold, value, inplace = FALSE)

nnf_threshold_(input, threshold, value)
```

Arguments

input	(N,*) tensor, where * means, any number of additional dimensions
threshold	The value to threshold at
value	The value to replace with
inplace	can optionally do the operation in-place. Default: FALSE

`nnf_triplet_margin_loss` *Triplet_margin_loss*

Description

Creates a criterion that measures the triplet loss given an input tensors x1 , x2 , x3 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_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

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

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_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

in_features	(int): Number of features in the input tensor
n_classes	(int): Number of classes in the dataset
cutoffs	(Sequence): Cutoffs used to assign targets to their buckets
div_value	(float, optional): value used as an exponent to compute sizes of the clusters. Default: 4.0
head_bias	(bool, optional): If True, adds a bias term to the 'head' of the adaptive softmax. Default: False

Details

Adaptive softmax is an approximate strategy for training models with large output spaces. It is most effective when the label distribution is highly imbalanced, for example in natural language modelling, where the word frequency distribution approximately follows the Zipf's law.

Adaptive softmax partitions the labels into several clusters, according to their frequency. These clusters may contain different number of targets each.

Additionally, clusters containing less frequent labels assign lower dimensional embeddings to those labels, which speeds up the computation. For each minibatch, only clusters for which at least one target is present are evaluated.

The idea is that the clusters which are accessed frequently (like the first one, containing most frequent labels), should also be cheap to compute – that is, contain a small number of assigned labels. We highly recommend taking a look at the original paper for more details.

- 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 $0 <= \text{target[i]} <= \text{n\_classes}$
- 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_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

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

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size `C` (where `C` is the input size). By default, the elements of γ are set to 1 and the elements of β 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` *BatchNorm2D*

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

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

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size C (where C is the input size). By default, the elements of γ are set to 1 and the elements of β 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_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. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Details

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

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

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

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

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets y should be numbers between 0 and 1.

Notice that if x_n is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set $\log(0) = -\infty$, since $\lim_{x \to 0} \log(x) = -\infty$.

However, an infinite term in the loss equation is not desirable for several reasons. For one, if either $y_n = 0$ or $(1 - y_n) = 0$, then we would be multiplying 0 with infinity. Secondly, if we have an infinite loss value, then we would also have an infinite term in our gradient, since $\lim_{x \to 0} \frac{d}{dx} \log(x) = \infty$.

This would make BCELoss's backward method nonlinear with respect to x_n, and using it for things like linear regression would not be straight-forward. Our solution is that BCELoss clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.

Shape

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then $(N, *)$, same shape as input.

Examples

```
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_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: $(N, *, H_{in1})$ $H_{in1} =$ 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 $H_{in2} =$ in2_features.
- Output: $(N, *, H_{out})$ where $H_{out} =$ out_features and all but the last dimension are the same shape as the input.

Attributes

- weight: the learnable weights of the module of shape (out_features, in1_features, 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 (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_celu` *CELU module*

Description

Applies the element-wise function:

Usage

```
nn_celu(alpha = 1, inplace = FALSE)
```

Arguments

`alpha`	the α 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_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 $(N, C_{\text{out}}, L_{\text{out}})$ 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

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): Zero-padding added to both sides of the input. Default: 0
dilation	(int or tuple, optional): Spacing between kernel elements. Default: 1

`groups`	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
`bias`	(bool, optional): If `TRUE`, adds a learnable bias to the output. Default: `TRUE`
`padding_mode`	(string, optional): `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`

Details

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

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

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

Shape

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

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

Attributes

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

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

`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): Zero-padding added to both sides of the input. Default: 0

dilation	(int or tuple, optional): Spacing between kernel elements. Default: 1
groups	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
bias	(bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
padding_mode	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

In the simplest case, the output value of the layer with input size (N, C_{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: $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$.

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

- a single int – in which case the same value is used for the height and width dimension
- a tuple of two ints – in which case, the first int is used for the height dimension, and the second int for the width dimension

Note

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

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

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

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): Zero-padding added to all three sides of the input. Default: 0
dilation	(int or tuple, optional): Spacing between kernel elements. Default: 1
groups	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
bias	(bool, optional): If TRUE, adds a learnable bias to the output. Default: TRUE
padding_mode	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

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

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

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

- a single `int` – in which case the same value is used for the 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 $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$
- bias (Tensor): the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$

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 $(in\_channels = C_{in}, out\_channels = C_{in} \times K, ..., groups = C_{in})$.

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

`in_channels`	(int): Number of channels in the input image
`out_channels`	(int): Number of channels produced by the convolution
`kernel_size`	(int or tuple): Size of the convolving kernel
`stride`	(int or tuple, optional): Stride of the convolution. Default: 1
`padding`	(int or tuple, optional): `dilation * (kernel_size -1) -padding` zero-padding will be added to both sides of the input. Default: 0
`output_padding`	(int or tuple, optional): Additional size added to one side of the output shape. Default: 0
`groups`	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
`bias`	(bool, optional): If `True`, adds a learnable bias to the output. Default: `TRUE`
`dilation`	(int or tuple, optional): Spacing between kernel elements. Default: 1
`padding_mode`	(string, optional): `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`

Details

This module can be seen as the gradient of Conv1d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `dilation * (kernel_size -1) -padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

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 (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kernel_size). The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \text{kernel\_size}}$
- bias (Tensor): the learnable bias of the module of shape (out_channels). If `bias` is `TRUE`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \text{kernel\_size}}$

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 `dilation * (kernel_size - 1) - 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

in_channels	(int): Number of channels in the input image
out_channels	(int): Number of channels produced by the convolution
kernel_size	(int or tuple): Size of the convolving kernel
stride	(int or tuple, optional): Stride of the convolution. Default: 1
padding	(int or tuple, optional): dilation * (kernel_size -1) -padding zero-padding will be added to both sides of each dimension in the input. Default: 0
output_padding	(int or tuple, optional): Additional size added to one side of each dimension in the output shape. Default: 0
groups	(int, optional): Number of blocked connections from input channels to output channels. Default: 1
bias	(bool, optional): If True, adds a learnable bias to the output. Default: True
dilation	(int or tuple, optional): Spacing between kernel elements. Default: 1
padding_mode	(string, optional): 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Details

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- stride controls the stride for the cross-correlation.
- padding controls the amount of implicit zero-paddings on both sides for dilation * (kernel_size -1) -padding number of points. See note below for details.
- output_padding controls the additional size added to one side of the output shape. See note below for details.
- dilation controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link_ has a nice visualization of what dilation does.
- groups controls the connections between inputs and outputs. in_channels and out_channels must both be divisible by groups. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

- At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

The parameters `kernel_size`, `stride`, `padding`, `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 $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- bias (Tensor): the learnable bias of the module of shape (out_channels) If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

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 `dilation * (kernel_size - 1) - 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 `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 $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

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

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 `dilation * (kernel_size - 1) - 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_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. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

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 $(minibatch, C, d_1, d_2, ..., d_K)$ 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:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

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 $(minibatch, C, d_1, d_2, ..., d_K)$ 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 $(N, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N), or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

Examples

```
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_dropout *Dropout module*

Description

During training, randomly zeroes some of the elements of the input tensor with probability `p` using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

Usage

```
nn_dropout(p = 0.5, inplace = FALSE)
```

Arguments

`p`	probability of an element to be zeroed. Default: 0.5
`inplace`	If set to `TRUE`, will do this operation in-place. Default: `FALSE`.

Details

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors.

Furthermore, the outputs are scaled by a factor of :math:\frac{1}{1-p} during training. This means that during evaluation the module simply computes an identity function.

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 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 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 *ELU module*

Description

Applies the element-wise function:

Usage

```
nn_elu(alpha = 1, inplace = FALSE)
```

Arguments

alpha	the α 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 *Embedding module*

Description

A simple lookup table that stores embeddings of a fixed dictionary and size. This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Usage

```
nn_embedding(
  num_embeddings,
  embedding_dim,
  padding_idx = NULL,
  max_norm = NULL,
  norm_type = 2,
  scale_grad_by_freq = FALSE,
  sparse = FALSE,
  .weight = NULL
)
```

Arguments

num_embeddings	(int): size of the dictionary of embeddings
embedding_dim	(int): the size of each embedding vector
padding_idx	(int, optional): If given, pads the output with the embedding vector at padding_idx (initialized to zeros) whenever it encounters the index.
max_norm	(float, optional): If given, each embedding vector with norm larger than max_norm is renormalized to have norm max_norm.
norm_type	(float, optional): The p of the p-norm to compute for the max_norm option. Default 2.
scale_grad_by_freq	(boolean, optional): If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default False.
sparse	(bool, optional): If True, gradient w.r.t. weight matrix will be a sparse tensor.
.weight	(Tensor) embeddings weights (in case you want to set it manually)
	See Notes for more details regarding sparse gradients.

Attributes

- weight (Tensor): the learnable weights of the module of shape (num_embeddings, embedding_dim) initialized from $\mathcal{N}(0, 1)$

Shape

- Input: $(*)$, LongTensor of arbitrary shape containing the indices to extract
- Output: $(*, H)$, where $*$ is the input shape and $H =$ 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_gelu *GELU module*

Description

Applies the Gaussian Error Linear Units function:

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

Usage

```
nn_gelu()
```

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_hardshrink *Hardshwink module*

Description

Applies the hard shrinkage function element-wise:

Usage

```
nn_hardshrink(lambd = 0.5)
```

Arguments

lambd	the λ 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 :math:[-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_identity *Identity module*

Description

A placeholder identity operator that is argument-insensitive.

Usage

```
nn_identity(...)
```

Arguments

...	any arguments (unused)

Examples

```
if (torch_is_installed()) {
m <- nn_identity(54, unused_argument1 = 0.1, unused_argument2 = FALSE)
input <- torch_randn(128, 20)
output <- m(input)
print(output$size())

}
```

nn_init_calculate_gain *Calculate gain*

Description

Return the recommended gain value for the given nonlinearity function.

Usage

```
nn_init_calculate_gain(nonlinearity, param = NULL)
```

Arguments

nonlinearity	the non-linear function
param	optional parameter for the non-linear function

nn_init_constant_ *Constant initialization*

Description

Fills the input Tensor with the value `val`.

Usage

```
nn_init_constant_(tensor, val)
```

Arguments

`tensor`	an n-dimensional `Tensor`
`val`	the value to fill the tensor with

Examples

```
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 non-linear dynamics of learning in deep linear neural networks - Saxe, A. et al. (2013). The input tensor must have at least 2 dimensions, and for tensors with more than 2 dimensions the trailing dimensions are flattened.

Usage

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

Arguments

tensor	an n-dimensional Tensor
gain	optional scaling factor

Examples

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

}
```

`nn_init_sparse_` *Sparse initialization*

Description

Fills the 2D input `Tensor` as a sparse matrix, where the non-zero elements will be drawn from the normal distribution as described in Deep learning via Hessian-free optimization - Martens, J. (2010).

Usage

```
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 feedforward neural networks - 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 feedforward neural networks - 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_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` *Linear module*

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 $k = \frac{1}{\text{in\_features}}$
- 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 $k = \frac{1}{\text{in\_features}}$

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 [-inf, 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_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:

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

If padding is non-zero, then the input is implicitly zero-padded on both sides for padding number of points. dilation controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what dilation does.

Shape

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

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

Examples

```
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:

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

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_module *Base class for all neural network modules.*

Description

Your models should also subclass this class.

Usage

```
nn_module(classname = NULL, inherit = nn_Module, ...)
```

Arguments

classname	an optional name for the module
inherit	an optional module to inherit from
...	methods implementation

Details

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes.

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_list *Holds submodules in a list.*

Description

nn_module_list can be indexed like a regular R list, but modules it contains are properly registered, and will be visible by all nn_module methods.

Usage

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

Arguments

modules	a list of modules to add

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_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
)
```

Arguments

embed_dim	total dimension of the model.
num_heads	parallel attention 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.

Details

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, \ldots, head_h)W^O \text{where} head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Shape

Inputs:

- query: (L, N, E) where L is the target sequence length, N is the batch size, E is the embedding dimension.
- key: (S, N, E), where S is the source sequence length, N is the batch size, E is the embedding dimension.
- value: (S, N, E) where S is the source sequence length, N is the batch size, E is the embedding dimension.
- 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 $(N * num_heads, L, S)$ 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.

Outputs:

- attn_output: (L, N, E) where L is the target sequence length, N is the batch size, E is the embedding dimension.
- attn_output_weights: (N, L, S) where N is the batch size, L is the target sequence length, S is the source sequence length.

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_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 (`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_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

input_size	The number of expected features in the input x
hidden_size	The number of features in the hidden state h
num_layers	Number of recurrent layers. E.g., setting num_layers=2 would mean stacking two RNNs together to form a stacked RNN, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
nonlinearity	The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
bias	If FALSE, then the layer does not use bias weights b_ih and b_hh. Default: TRUE
batch_first	If TRUE, then the input and output tensors are provided as (batch, seq, feature). Default: FALSE
dropout	If non-zero, introduces a Dropout layer on the outputs of each RNN layer except the last layer, with dropout probability equal to dropout. Default: 0
bidirectional	If TRUE, becomes a bidirectional RNN. Default: FALSE
...	other arguments that can be passed to the super class.

Details

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

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time t, x_t is the input at time t, and $h_{(t-1)}$ is the hidden state of the previous layer at time t-1 or the initial hidden state at time 0. If nonlinearity is 'relu', then ReLU is used instead of $\tanh$.

Inputs

- **input** of shape (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence.
- **h_0** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

Outputs

- **output** of shape (seq_len, batch, num_directions * hidden_size): tensor containing the output features (`h_t`) from the last layer of the RNN, for each `t`. If a :class:`nn_packed_sequence` has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using `output$view(seq_len,batch,num_directions,hidden_size)`, with forward and backward being direction `0` and `1` respectively. Similarly, the directions can be separated in the packed case.
- **h_n** of shape (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for `t = seq_len`. Like *output*, the layers can be separated using `h_n$view(num_layers,num_directions,batch,`

Shape

- Input1: (L, N, H_{in}) tensor containing input features where H_{in} = 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} = hidden_size Defaults to zero if not provided. where S = num_layers$*$num_directions If the RNN is bidirectional, num_directions should be 2, else it should be 1.
- Output1: (L, N, H_{all}) where H_{all} = num_directions $*$ 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 (hidden_size, input_size) for `k = 0`. Otherwise, the shape is (hidden_size, num_directions * hidden_size)
- `weight_hh_l[k]`: the learnable hidden-hidden weights of the k-th layer, of shape (hidden_size, hidden_size)
- `bias_ih_l[k]`: the learnable input-hidden bias of the k-th layer, of shape `(hidden_size)`
- `bias_hh_l[k]`: the learnable hidden-hidden bias of the k-th layer, of shape `(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

`lower`	lower bound of the uniform distribution. Default: $\frac{1}{8}$
`upper`	upper bound of the uniform distribution. Default: $\frac{1}{3}$
`inplace`	can optionally do the operation in-place. Default: `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 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 *A sequential container*

Description

A sequential container. Modules will be added to it in the order they are passed in the constructor. See examples.

Usage

```
nn_sequential(..., name = NULL)
```

Arguments

...	sequence of modules to be added
name	optional name for the generated module.

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_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 unspecifed 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 β 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 *Softshrink module*

Description

Applies the soft shrinkage function elementwise:

Usage

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

Arguments

lambd	the λ (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_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 *Tanhshrink module*

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 *Threshoold module*

Description

Thresholds each element of the input Tensor.

Usage

```
nn_threshold(threshold, value, inplace = FALSE)
```

Arguments

threshold	The value to threshold at
value	The value to replace with
inplace	can optionally do the operation in-place. Default: FALSE

Details

Threshold is defined as:

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

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_utils_rnn_pack_padded_sequence

Packs a Tensor containing padded sequences of variable length.

Description

`input` can be of size T x B x * where `T` is the length of the longest sequence (equal to `lengths[1]`), `B` is the batch size, and * is any number of dimensions (including 0). If `batch_first` is `TRUE`, B x T x * `input` is expected.

Usage

```
nn_utils_rnn_pack_padded_sequence(
  input,
  lengths,
  batch_first = FALSE,
  enforce_sorted = TRUE
)
```

Arguments

`input`	(Tensor): padded batch of variable length sequences.
`lengths`	(Tensor): list of sequences lengths of each batch element.
`batch_first`	(bool, optional): if `TRUE`, the input is expected in B x T x * format.
`enforce_sorted`	(bool, optional): if `TRUE`, the input is expected to contain sequences sorted by length in a decreasing order. If `FALSE`, the input will get sorted unconditionally. Default: `TRUE`.

Details

For unsorted sequences, use `enforce_sorted = FALSE`. If `enforce_sorted` is `TRUE`, the sequences should be sorted by length in a decreasing order, i.e. `input[,1]` 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 x *, 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

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

Details

The returned Tensor's data will be of size T x B x *, 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 x T x * format.

Value

Tuple of Tensor containing the padded sequence, and a Tensor containing the list of lengths of each sequence in the batch. Batch elements will be re-ordered as they were ordered originally when the batch was passed to `nn_utils_rnn_pack_padded_sequence()` or `nn_utils_rnn_pack_sequence()`.

Note

total_length is useful to implement the pack sequence -> recurrent network -> unpack sequence pattern in a nn_module wrapped in ~torch.nn.DataParallel.

Examples

```
if (torch_is_installed()) {
seq <- torch_tensor(rbind(c(1,2,0), c(3,0,0), c(4,5,6)))
lens <- c(2,1,3)
packed <- nn_utils_rnn_pack_padded_sequence(seq, lens, batch_first = TRUE,
                                            enforce_sorted = FALSE)
packed
nn_utils_rnn_pad_packed_sequence(packed, batch_first=TRUE)

}
```

---

nn_utils_rnn_pad_sequence

Pad a list of variable length Tensors with padding_value

Description

pad_sequence stacks a list of Tensors along a new dimension, and pads them to equal length. For example, if the input is list of sequences with size L x * and if batch_first is False, and T x B x * 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 x T x * if TRUE, or in T x B x * 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()

}
```

---

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 On the Convergence of Adam and Beyond (default: FALSE)

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_required *Dummy value indicating a required value.*

Description

export

Usage

```
optim_required()
```

---

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

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

where p, g, v and μ 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

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

The Nesterov version is analogously modified.

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)

}
```

---

tensor_dataset *Dataset wrapping tensors.*

Description

Each sample will be retrieved by indexing tensors along the first dimension.

Usage

```
tensor_dataset(...)
```

Arguments

...	tensors that have the same size of the first dimension.

---

torch_abs *Abs*

Description

Abs

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

abs(input, out=None) -> 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_acos *Acos*

Description

Acos

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

acos(input, out=None) -> 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_adaptive_avg_pool1d` *Adaptive_avg_pool1d*

Description

Adaptive_avg_pool1d

Arguments

`output_size`	NA 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 `~torch.nn.AdaptiveAvgPool1d` for details and output shape.

`torch_add` *Add*

Description

Add

Arguments

`input`	(Tensor) the input tensor.
`value`	(Number) the number to be added to each element of `input`
`other`	(Tensor) the second input tensor
`alpha`	(Number) the scalar multiplier for `other`

add(input, other, out=None)

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=None)

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` *Addbmm*

Description

Addbmm

Arguments

`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` (β)
`input`	(Tensor) matrix to be added
`alpha`	(Number, optional) multiplier for `batch1 @ batch2` (α)
`out`	(Tensor, optional) the output tensor.

addbmm(input, batch1, batch2, *, beta=1, alpha=1, out=None) -> 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.

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

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(3, 5))
batch1 = torch_randn(c(10, 3, 4))
batch2 = torch_randn(c(10, 4, 5))
torch_addbmm(M, batch1, batch2)
}
```

`torch_addcdiv` *Addcdiv*

Description

Addcdiv

Arguments

`input`	(Tensor) the tensor to be added
`tensor1`	(Tensor) the numerator tensor
`tensor2`	(Tensor) the denominator tensor
`value`	(Number, optional) multiplier for tensor1/tensor2
`out`	(Tensor, optional) the output tensor.

addcdiv(input, tensor1, tensor2, *, value=1, out=None) -> Tensor

Performs the element-wise division of `tensor1` by `tensor2`, multiply the result by the scalar `value` and add it to `input`.

Warning

Integer division with addcdiv is deprecated, and in a future release addcdiv will perform a true division of `tensor1` and `tensor2`. The current addcdiv behavior can be replicated using `torch_floor_divide()` for integral inputs (`input + value * tensor1 // tensor2`) and `torch_div()` for float inputs (`input + value * tensor1 / tensor2`). The new addcdiv behavior can be implemented with `torch_true_divide()` (`input + value *` torch.true_divide(`tensor1`, `tensor2`).

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

The shapes of `input`, `tensor1`, and `tensor2` must be broadcastable .

For inputs of type `FloatTensor` or `DoubleTensor`, `value` must be a real number, otherwise an integer.

Examples

```
if (torch_is_installed()) {

t = torch_randn(c(1, 3))
t1 = torch_randn(c(3, 1))
t2 = torch_randn(c(1, 3))
torch_addcdiv(t, t1, t2, 0.1)
}
```

---

torch_addcmul *Addcmul*

Description

Addcmul

Arguments

`input`	(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 $tensor1. * tensor2$
`out`	(Tensor, optional) the output tensor.

addcmul(input, tensor1, tensor2, *, value=1, out=None) -> Tensor

Performs the element-wise multiplication of `tensor1` by `tensor2`, multiply the result by the scalar `value` and add it to `input`.

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

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

Arguments

`input`	(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` (β)
`alpha`	(Number, optional) multiplier for $mat1@mat2$ (α)
`out`	(Tensor, optional) the output tensor.

addmm(input, mat1, mat2, *, beta=1, alpha=1, out=None) -> 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 *Addmv*

Description

Addmv

Arguments

input	(Tensor) vector to be added
mat	(Tensor) matrix to be multiplied
vec	(Tensor) vector to be multiplied
beta	(Number, optional) multiplier for input (β)
alpha	(Number, optional) multiplier for $mat@vec$ (α)
out	(Tensor, optional) the output tensor.

addmv(input, mat, vec, *, beta=1, alpha=1, out=None) -> 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.

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

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

Arguments

`input`	(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` (β)
`alpha`	(Number, optional) multiplier for vec1 $\otimes$ vec2 (α)
`out`	(Tensor, optional) the output tensor.

addr(input, vec1, vec2, *, beta=1, alpha=1, out=None) -> 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., 4.)
vec2 = torch_arange(1., 3.)
M = torch_zeros(c(3, 2))
torch_addr(M, vec1, vec2)
}
```

torch_allclose *Allclose*

Description

Allclose

Arguments

input	(Tensor) first tensor to compare
other	(Tensor) second tensor to compare
atol	(float, optional) absolute tolerance. Default: 1e-08
rtol	(float, optional) relative tolerance. Default: 1e-05
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_angle *Angle*

Description

Angle

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

angle(input, out=None) -> Tensor

Computes the element-wise angle (in radians) of the given `input` tensor.

$$\text{out}_i = angle(\text{input}_i)$$

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

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`.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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=None, dtype=None, layout=torch.strided, device=None, 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 [start, end) taken with common difference `step` beginning from `start`.

Note that non-integer `step` is subject to floating point rounding errors when comparing against `end`; to avoid inconsistency, we advise adding a small epsilon to `end` in such cases.

$$\text{out}_{i+1} = \text{out}_i + \text{step}$$

Examples

```
if (torch_is_installed()) {

torch_arange(start = 0, end = 5)
torch_arange(1, 4)
torch_arange(1, 2.5, 0.5)
}
```

`torch_argmax` *Argmax*

Description

Argmax

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int) the dimension to reduce. If `None`, the argmax of the flattened input is returned.
`keepdim`	(bool) whether the output tensor has `dim` retained or not. Ignored if `dim=None`.

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` *Argmin*

Description

Argmin

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int) the dimension to reduce. If `None`, the argmin of the flattened input is returned.
`keepdim`	(bool) whether the output tensor has `dim` retained or not. Ignored if `dim=None`.

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=None) -> 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` *Argsort*

Description

Argsort

Arguments

`input`	(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 *Asin*

Description

Asin

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

asin(input, out=None) -> 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_as_strided *As_strided*

Description

As_strided

Arguments

input	(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 *Atan*

Description

Atan

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

atan(input, out=None) -> 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 *Atan2*

Description

Atan2

Arguments

input	(Tensor) the first input tensor
other	(Tensor) the second input tensor
out	(Tensor, optional) the output tensor.

atan2(input, other, out=None) -> 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 other_i, the second parameter, is the x-coordinate, while 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_avg_pool1d *Avg_pool1d*

Description

Avg_pool1d

Arguments

input	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
kernel_size	NA the size of the window. Can be a single number or a tuple (kW,)
stride	NA the stride of the window. Can be a single number or a tuple (sW,). Default: `kernel_size`
padding	NA implicit zero paddings on both sides of the input. Can be a single number or a tuple (padW,). Default: 0

`ceil_mode` NA when True, will use `ceil` instead of `floor` to compute the output shape. Default: `False`

`count_include_pad` NA when True, will include the zero-padding in the averaging calculation. Default: `True`

avg_pool1d(input, kernel_size, stride=None, 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 `~torch.nn.AvgPool1d` for details and output shape.

`torch_baddbmm` *Baddbmm*

Description

Baddbmm

Arguments

`input`	(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` (β)
`alpha`	(Number, optional) multiplier for batch1 @ batch2 (α)
`out`	(Tensor, optional) the output tensor.

baddbmm(input, batch1, batch2, *, beta=1, alpha=1, out=None) -> 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

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 None, 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 None, 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=None, layout=torch.strided, device=None, 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` \eqn{=1}, the returned window contains a single value 1.

---

torch_bernoulli *Bernoulli*

Description

Bernoulli

Arguments

input	(Tensor) the input tensor of probability values for the Bernoulli distribution
generator	(torch.Generator, optional) a pseudorandom number generator for sampling
out	(Tensor, optional) the output tensor.

bernoulli(input, *, generator=None, out=None) -> 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 ith element of the output tensor will draw a value 1 according to the ith 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` *Bincount*

Description

Bincount

Arguments

`input`	(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=None, 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(0, 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` *Bitwise_and*

Description

Bitwise_and

Arguments

input	NA the first input tensor
other	NA the second input tensor
out	(Tensor, optional) the output tensor.

bitwise_and(input, other, out=None) -> 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 *Bitwise_not*

Description

Bitwise_not

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

bitwise_not(input, out=None) -> 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 *Bitwise_or*

Description

Bitwise_or

Arguments

input	NA the first input tensor
other	NA the second input tensor
out	(Tensor, optional) the output tensor.

bitwise_or(input, other, out=None) -> 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 *Bitwise_xor*

Description

Bitwise_xor

Arguments

input	NA the first input tensor
other	NA the second input tensor
out	(Tensor, optional) the output tensor.

bitwise_xor(input, other, out=None) -> 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

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 None, 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 None, 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=None, layout=torch.strided, device=None, 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` \eqn{=1}, the returned window contains a single value 1.
```

`torch_bmm` *Bmm*

Description

Bmm

Arguments

`input`	(Tensor) the first batch of matrices to be multiplied
`mat2`	(Tensor) the second batch of matrices to be multiplied
`out`	(Tensor, optional) the output tensor.

bmm(input, mat2, out=None) -> 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.

$$\text{out}_i = \text{input}_i \mathbin{@} \text{mat2}_i$$

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

Arguments

`*tensors`	NA 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)$view(c(1, 3))
y = torch_arange(0, 2)$view(c(2, 1))
out = torch_broadcast_tensors(list(x, y))
out[[1]]
}
```

`torch_can_cast` *Can_cast*

Description

Can_cast

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` *Cartesian_prod*

Description

Cartesian_prod

Arguments

`*tensors`	NA any number of 1 dimensional tensors.

TEST

Do cartesian product of the given sequence of tensors. The behavior is similar to python's `itertools.product`.

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

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
out	(Tensor, optional) the output tensor.

cat(tensors, dim=0, out=None) -> 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 *Cdist*

Description

Cdist

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 *Ceil*

Description

Ceil

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

ceil(input, out=None) -> 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_

celu_(input, alpha=1.) -> Tensor

In-place version of `torch_celu`.

`torch_chain_matmul` *Chain_matmul*

Description

Chain_matmul

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_cholesky` *Cholesky*

Description

Cholesky

Arguments

`input`	(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`
`out`	(Tensor, optional) the output matrix

cholesky(input, upper=False, out=None) -> Tensor

Computes the Cholesky decomposition of a symmetric positive-definite matrix A or for batches of symmetric positive-definite matrices.

If `upper` is `True`, the returned matrix `U` is upper-triangular, and the decomposition has the form:

$$A = U^T U$$

If `upper` is `False`, the returned matrix `L` is lower-triangular, and the decomposition has the form:

$$A = LL^T$$

If `upper` is `True`, and A is a batch of symmetric positive-definite matrices, then the returned tensor will be composed of upper-triangular Cholesky factors of each of the individual matrices. Similarly, when `upper` is `False`, the returned tensor will be composed of lower-triangular Cholesky factors of each of the individual matrices.

Examples

```
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

Arguments

`input`	(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
`out`	(Tensor, optional) the output tensor for `inv`

cholesky_inverse(input, upper=False, out=None) -> 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

Arguments

`input`	(Tensor) input matrix b of size $(*, m, k)$, where $*$ is zero or more batch dimensions
`input2`	(Tensor) input matrix u of size $(*, m, m)$, where $*$ is zero of more batch dimensions composed of upper or lower triangular Cholesky factor
`upper`	(bool, optional) whether to consider the Cholesky factor as a lower or upper triangular matrix. Default: `False`.
`out`	(Tensor, optional) the output tensor for `c`

cholesky_solve(input, input2, upper=False, out=None) -> 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 *Chunk*

Description

Chunk

Arguments

input	(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 *Clamp*

Description

Clamp

Arguments

input	(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
out	(Tensor, optional) the output tensor.
value	(Number) minimal value of each element in the output

clamp(input, min, max, out=None) -> Tensor

Clamp all elements in input into the range [min, max] and return a resulting tensor:

$$y_i = \begin{cases} \text{min} & \text{if } x_i < \text{min} \\ x_i & \text{if min} \leq x_i \leq \text{max} \\ \text{max} & \text{if } x_i > \text{max} \end{cases}$$

If input is of type FloatTensor or DoubleTensor, args min and max must be real numbers, otherwise they should be integers.

clamp(input, *, min, out=None) -> 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=None) -> 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_combinations *Combinations*

Description

Combinations

Arguments

`input`	(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_conj *Conj*

Description

Conj

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

conj(input, out=None) -> Tensor

Computes the element-wise conjugate of the given input tensor.

$$\text{out}_i = conj(\text{input}_i)$$

Examples

```
if (torch_is_installed()) {
## Not run:
torch_conj(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))

## End(Not run)
}
```

`torch_conv1d` *Conv1d*

Description

Conv1d

Arguments

`input`	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
`weight`	NA filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kW)$
`bias`	NA optional bias of shape (out_channels). Default: `None`
`stride`	NA the stride of the convolving kernel. Can be a single number or a one-element tuple (sW,). Default: 1
`padding`	NA implicit paddings on both sides of the input. Can be a single number or a one-element tuple (padW,). Default: 0
`dilation`	NA the spacing between kernel elements. Can be a single number or a one-element tuple (dW,). Default: 1
`groups`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1

conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) -> Tensor

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

See `~torch.nn.Conv1d` for details and output shape.

.. include:: cudnn_deterministic.rst

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

Arguments

`input`	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
`weight`	NA filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kH, kW)$
`bias`	NA optional bias tensor of shape (out_channels). Default: `None`
`stride`	NA the stride of the convolving kernel. Can be a single number or a tuple (sH, sW). Default: 1
`padding`	NA implicit paddings on both sides of the input. Can be a single number or a tuple (padH, padW). Default: 0
`dilation`	NA the spacing between kernel elements. Can be a single number or a tuple (dH, dW). Default: 1
`groups`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1

conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) -> Tensor

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

See `~torch.nn.Conv2d` for details and output shape.

.. include:: cudnn_deterministic.rst

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` *Conv3d*

Description

Conv3d

Arguments

`input`	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
`weight`	NA filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kT, kH, kW)$
`bias`	NA optional bias tensor of shape (out_channels). Default: None
`stride`	NA the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sW). Default: 1
`padding`	NA implicit paddings on both sides of the input. Can be a single number or a tuple (padT, padH, padW). Default: 0
`dilation`	NA the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1
`groups`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1

conv3d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) -> Tensor

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

See ~`torch.nn.Conv3d` for details and output shape.

.. include:: cudnn_deterministic.rst

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` *Conv_tbc*

Description

Conv_tbc

Arguments

`input`	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` *Conv_transpose1d*

Description

Conv_transpose1d

Arguments

`input`	NA input tensor of shape (minibatch, in_channels, iW)
`weight`	NA filters of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kW)
`bias`	NA optional bias of shape (out_channels). Default: None
`stride`	NA the stride of the convolving kernel. Can be a single number or a tuple (sW,). Default: 1
`padding`	NA `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`	NA 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`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1
`dilation`	NA the spacing between kernel elements. Can be a single number or a tuple (dW,). Default: 1

conv_transpose1d(input, weight, bias=None, 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 `~torch.nn.ConvTranspose1d` for details and output shape.

.. include:: cudnn_deterministic.rst

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

Arguments

`input`	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
`weight`	NA filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kH, kW)$
`bias`	NA optional bias of shape (out_channels). Default: None
`stride`	NA the stride of the convolving kernel. Can be a single number or a tuple (sH, sW). Default: 1
`padding`	NA `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`	NA 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`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1
`dilation`	NA the spacing between kernel elements. Can be a single number or a tuple (dH, dW). Default: 1

conv_transpose2d(input, weight, bias=None, 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 `~torch.nn.ConvTranspose2d` for details and output shape.

.. include:: cudnn_deterministic.rst

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

Arguments

`input`	NA input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
`weight`	NA filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kT, kH, kW)$
`bias`	NA optional bias of shape (out_channels). Default: None
`stride`	NA the stride of the convolving kernel. Can be a single number or a tuple (sT, sH, sW). Default: 1
`padding`	NA `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`	NA 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`	NA split input into groups, in_channels should be divisible by the number of groups. Default: 1
`dilation`	NA the spacing between kernel elements. Can be a single number or a tuple (dT, dH, dW). Default: 1

conv_transpose3d(input, weight, bias=None, 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 `~torch.nn.ConvTranspose3d` for details and output shape.

.. include:: cudnn_deterministic.rst

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))
nnf_conv_transpose3d(inputs, weights)

## End(Not run)
}
```

`torch_cos` *Cos*

Description

Cos

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

cos(input, out=None) -> 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 *Cosh*

Description

Cosh

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

cosh(input, out=None) -> 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 *Cosine_similarity*

Description

Cosine_similarity

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_cross` *Cross*

Description

Cross

Arguments

`input`	(Tensor) the input tensor.
`other`	(Tensor) the second input tensor
`dim`	(int, optional) the dimension to take the cross-product in.
`out`	(Tensor, optional) the output tensor.

cross(input, other, dim=-1, out=None) -> 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

Arguments

input	(Tensor) the input tensor.
dim	(int) the dimension to do the operation over
out	(tuple, optional) the result tuple of two output tensors (values, indices)

cummax(input, dim, out=None) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where values is the cumulative maximum of elements of input in the dimension dim. And indices is the index location of each maximum value found in the dimension dim.

$$y_i = max(x_1, x_2, x_3, \ldots, x_i)$$

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cummax(a, dim=1)
}
```

torch_cummin *Cummin*

Description

Cummin

Arguments

input	(Tensor) the input tensor.
dim	(int) the dimension to do the operation over
out	(tuple, optional) the result tuple of two output tensors (values, indices)

cummin(input, dim, out=None) -> (Tensor, LongTensor)

Returns a namedtuple (values, indices) where `values` is the cumulative minimum of elements of `input` in the dimension `dim`. And `indices` is the index location of each maximum value found in the dimension `dim`.

$$y_i = min(x_1, x_2, x_3, \ldots, x_i)$$

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cummin(a, dim=1)
}
```

`torch_cumprod` *Cumprod*

Description

Cumprod

Arguments

`input`	(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: None.
`out`	(Tensor, optional) the output tensor.

cumprod(input, dim, out=None, dtype=None) -> 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.

$$y_i = x_1 \times x_2 \times x_3 \times \ldots \times x_i$$

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cumprod(a, dim=1)
}
```

torch_cumsum *Cumsum*

Description

Cumsum

Arguments

input	(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: None.
out	(Tensor, optional) the output tensor.

cumsum(input, dim, out=None, dtype=None) -> 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.

$$y_i = x_1 + x_2 + x_3 + \ldots + x_i$$

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(10))
a
torch_cumsum(a, dim=1)
}
```

torch_det *Det*

Description

Det

Arguments

input	(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 *Create a Device object*

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 *Diag*

Description

Diag

Arguments

input	(Tensor) the input tensor.
diagonal	(int, optional) the diagonal to consider
out	(Tensor, optional) the output tensor.

diag(input, diagonal=0, out=None) -> Tensor

- If input is a vector (1-D tensor), then returns a 2-D square tensor with the elements of input as the diagonal.
- If input is a matrix (2-D tensor), then returns a 1-D tensor with the diagonal elements of input.

The argument diagonal controls which diagonal to consider:

- If diagonal = 0, it is the main diagonal.
- If diagonal > 0, it is above the main diagonal.
- If diagonal < 0, it is below the main diagonal.

`torch_diagflat` *Diagflat*

Description

Diagflat

Arguments

`input`	(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` *Diagonal*

Description

Diagonal

Arguments

`input`	(Tensor) the input tensor. Must be at least 2-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: 0.
`dim2`	(int, optional) second dimension with respect to which to take diagonal. Default: 1.

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 *Diag_embed*

Description

Diag_embed

Arguments

input	(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_digamma *Digamma*

Description

Digamma

Arguments

input	(Tensor) the tensor to compute the digamma function on

digamma(input, out=None) -> Tensor

Computes the logarithmic derivative of the gamma function on input.

$$\psi(x) = \frac{d}{dx} \ln \left(\Gamma \left(x \right) \right) = \frac{\Gamma'(x)}{\Gamma(x)}$$

Examples

```
if (torch_is_installed()) {

a = torch_tensor(c(1, 0.5))
torch_digamma(a)
}
```

torch_dist *Dist*

Description

Dist

Arguments

input	(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

Arguments

input	(Tensor) the input tensor.
other	(Number) the number to be divided to each element of input

div(input, other, out=None) -> 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 (// in Python) 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_dot` *Dot*

Description

Dot

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_dtype` *Torch data types*

Description

Returns the correspondent data type.

Usage

```
torch_float32()

torch_float()

torch_float64()

torch_double()

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

input	(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
out	(tuple, optional) the output tensors

eig(input, eigenvectors=False, out=None) -> (Tensor, Tensor)

Computes the eigenvalues and eigenvectors of a real square matrix.

Note

Since eigenvalues and eigenvectors might be complex, backward pass is supported only for [`torch_symeig`]

---

torch_einsum *Einsum*

Description

Einsum

Arguments

equation	(string) The equation is given in terms of lower case letters (indices) to be associated with each dimension of the operands and result. The left hand side lists the operands dimensions, separated by commas. There should be one index letter per tensor dimension. The right hand side follows after -> and gives the indices for the output. If the -> and right hand side are omitted, it implicitly defined as the alphabetically sorted list of all indices appearing exactly once in the left hand side. The indices not apprearing in the output are summed over after multiplying the operands entries. If an index appears several times for the same operand, a diagonal is taken. Ellipses ... represent a fixed number of dimensions. If the right hand side is inferred, the ellipsis dimensions are at the beginning of the output.
operands	(Tensor) The operands to compute the Einstein sum of.

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` *Empty*

Description

Empty

Arguments

`size`	(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.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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`.
`memory_format`	(`torch.memory_format`, optional) the desired memory format of returned Tensor. Default: `torch_contiguous_format`.

empty(*size, out=None, dtype=None, layout=torch.strided, device=None, 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` *Empty_like*

Description

Empty_like

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 `None`, defaults to the dtype of `input`.
`layout`	(`torch.layout`, optional) the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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=None, layout=None, device=None, 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

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 `None`, 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 `None`, 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=None, layout=None, device=None, 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

Arguments

input	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare
out	(Tensor, optional) the output tensor. Must be a ByteTensor

eq(input, other, out=None) -> 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 *Equal*

Description

Equal

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 *Erf*

Description

Erf

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

erf(input, out=None) -> Tensor

Computes the error function of each element. The error function is defined as follows:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Examples

```
if (torch_is_installed()) {

torch_erf(torch_tensor(c(0, -1., 10.)))
}
```

torch_erfc *Erfc*

Description

Erfc

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

erfc(input, out=None) -> Tensor

Computes the complementary error function of each element of `input`. The complementary error function is defined as follows:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Examples

```
if (torch_is_installed()) {

torch_erfc(torch_tensor(c(0, -1., 10.)))
}
```

`torch_erfinv` *Erfinv*

Description

Erfinv

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

erfinv(input, out=None) -> 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` *Exp*

Description

Exp

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

exp(input, out=None) -> 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_expm1` *Expm1*

Description

Expm1

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

expm1(input, out=None) -> 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 *Eye*

Description

Eye

Arguments

n	(int) the number of rows
m	(int, optional) the number of columns with default being n
out	(Tensor, optional) the output tensor.
dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, uses the current device for the default tensor type (see torch_set_default_tensor_type). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
requires_grad	(bool, optional) If autograd should record operations on the returned tensor. Default: False.

eye(n, m=None, out=None, dtype=None, layout=torch.strided, device=None, 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)
}
```

torch_fft *Fft*

Description

Fft

Arguments

input	(Tensor) the input tensor of at least signal_ndim + 1 dimensions
signal_ndim	(int) the number of dimensions in each signal. signal_ndim can only be 1, 2 or 3
normalized	(bool, optional) controls whether to return normalized results. Default: False

fft(input, signal_ndim, normalized=False) -> Tensor

Complex-to-complex Discrete Fourier Transform

This method computes the complex-to-complex discrete Fourier transform. Ignoring the batch dimensions, it computes the following expression:

$$X[\omega_1, \dots, \omega_d] = \sum_{n_1=0}^{N_1-1} \dots \sum_{n_d=0}^{N_d-1} x[n_1, \dots, n_d] e^{-j\ 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = signal_ndim is number of dimensions for the signal, and N_i is the size of signal dimension i.

This method supports 1D, 2D and 3D complex-to-complex transforms, indicated by signal_ndim. input must be a tensor with last dimension of size 2, representing the real and imaginary components of complex numbers, and should have at least signal_ndim + 1 dimensions with optionally arbitrary number of leading batch dimensions. If normalized is set to True, this normalizes the result by dividing it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary.

Returns the real and the imaginary parts together as one tensor of the same shape of input.

The inverse of this function is torch_ifft.

Warning

For CPU tensors, this method is currently only available with MKL. Use torch_backends.mkl.is_available to check if MKL is installed.

Note

For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same configuration. See cufft-plan-cache for more details on how to monitor and control the cache.

Examples

```
if (torch_is_installed()) {

# unbatched 2D FFT
x = torch_randn(c(4, 3, 2))
torch_fft(x, 2)
# batched 1D FFT
torch_fft(x, 1)
# arbitrary number of batch dimensions, 2D FFT
```

```
x = torch_randn(c(3, 3, 5, 5, 2))
torch_fft(x, 2)

}
```

`torch_flatten` *Flatten*

Description

Flatten

Arguments

`input`	(Tensor) the input tensor.
`start_dim`	(int) the first dim to flatten
`end_dim`	(int) the last dim to flatten

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` *Flip*

Description

Flip

Arguments

`input`	(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(0, 8)$view(c(2, 2, 2))
x
torch_flip(x, c(1, 2))
}
```

`torch_floor` *Floor*

Description

Floor

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

floor(input, out=None) -> 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` *Floor_divide*

Description

Floor_divide

Arguments

`input`	(Tensor) the numerator tensor
`other`	(Tensor or Scalar) the denominator

floor_divide(input, other, out=None) -> 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` *Fmod*

Description

Fmod

Arguments

`input`	(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
`out`	(Tensor, optional) the output tensor.

fmod(input, other, out=None) -> 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` *Frac*

Description

Frac

frac(input, out=None) -> 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` *Full*

Description

Full

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.
out	(Tensor, optional) the output tensor.
dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, 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=None, dtype=None, layout=torch.strided, device=None, 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 *Full_like*

Description

Full_like

Arguments

input	(Tensor) the size of input will determine size of the output tensor.
fill_value	NA the number to fill the output tensor with.
dtype	(torch.dtype, optional) the desired data type of returned Tensor. Default: if None, defaults to the dtype of input.
layout	(torch.layout, optional) the desired layout of returned tensor. Default: if None, defaults to the layout of input.
device	(torch.device, optional) the desired device of returned tensor. Default: if None, 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=None, dtype=None, layout=torch.strided, device=None, 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.d

---

torch_gather *Gather*

Description

Gather

Arguments

input	(Tensor) the source tensor
dim	(int) the axis along which to index
index	(LongTensor) the indices of elements to gather
out	(Tensor, optional) the destination tensor
sparse_grad	(bool,optional) If True, gradient w.r.t. input will be a sparse tensor.

gather(input, dim, index, out=None, 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 $(x_0, x_1..., x_{i-1}, x_i, x_{i+1}, ..., x_{n-1})$ and dim = i, then index must be an n-dimensional tensor with size $(x_0, x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_{n-1})$ where $y \geq 1$ and out will have the same size as index.

Examples

```
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_ge *Ge*

Description

Ge

Arguments

input	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare
out	(Tensor, optional) the output tensor that must be a BoolTensor

ge(input, other, out=None) -> Tensor

Computes input $\geq$ other element-wise.

The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

Examples

```
if (torch_is_installed()) {

torch_ge(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
         torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

---

torch_generator *Create a Generator object*

Description

A torch_generator is an object which manages the state of the algorithm that produces pseudo random numbers. Used as a keyword argument in many In-place random sampling functions.

Usage

```
torch_generator()
```

Examples

```
if (torch_is_installed()) {

# Via string
generator <- torch_generator()
generator$current_seed()
generator$set_current_seed(1234567L)
generator$current_seed()


}
```

`torch_geqrf` *Geqrf*

Description

Geqrf

Arguments

`input`	(Tensor) the input matrix
`out`	(tuple, optional) the output tuple of (Tensor, Tensor)

geqrf(input, out=None) -> (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

Arguments

input	(Tensor) 1-D input vector
vec2	(Tensor) 1-D input vector
out	(Tensor, optional) optional output matrix

ger(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_ger(v1, v2)
}
```

torch_gt *Gt*

Description

Gt

Arguments

input	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare
out	(Tensor, optional) the output tensor that must be a BoolTensor

gt(input, other, out=None) -> 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

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 α in the equation above
beta	(float, optional) The coefficient β in the equation above
dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, 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=None, layout=torch.strided, device=None, 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` \eqn{=1}, the returned window contains a single value 1.

This is a generalized version of `torch_hann_window`.
```

`torch_hann_window` *Hann_window*

Description

Hann_window

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 `None`, 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 `None`, 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=None, layout=torch.strided, device=None, 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` \eqn{=1}, the returned window contains a single value 1.
```

`torch_histc` *Histc*

Description

Histc

Arguments

`input`	(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)
`out`	(Tensor, optional) the output tensor.

histc(input, bins=100, min=0, max=0, out=None) -> 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_ifft` *Ifft*

Description

Ifft

Arguments

`input`	(Tensor) the input tensor of at least `signal_ndim + 1` dimensions
`signal_ndim`	(int) the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
`normalized`	(bool, optional) controls whether to return normalized results. Default: `False`

ifft(input, signal_ndim, normalized=False) -> Tensor

Complex-to-complex Inverse Discrete Fourier Transform

This method computes the complex-to-complex inverse discrete Fourier transform. Ignoring the batch dimensions, it computes the following expression:

$$X[\omega_1, \dots, \omega_d] = \frac{1}{\prod_{i=1}^{d} N_i} \sum_{n_1=0}^{N_1-1} \dots \sum_{n_d=0}^{N_d-1} x[n_1, \dots, n_d] e^{\, j \, 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = `signal_ndim` is number of dimensions for the signal, and N_i is the size of signal dimension i.

The argument specifications are almost identical with `torch_fft`. However, if `normalized` is set to `True`, this instead returns the results multiplied by $\sqrt{\prod_{i=1}^{d} N_i}$, to become a unitary operator. Therefore, to invert a `torch_fft`, the `normalized` argument should be set identically for `torch_fft`.

Returns the real and the imaginary parts together as one tensor of the same shape of `input`.

The inverse of this function is `torch_fft`.

Warning

For CPU tensors, this method is currently only available with MKL. Use `torch_backends.mkl.is_available` to check if MKL is installed.

Note

```
For CUDA tensors, an LRU cache is used for cuFFT plans to speed up
repeatedly running FFT methods on tensors of same geometry with same
configuration. See cufft-plan-cache for more details on how to
monitor and control the cache.
```

Examples

```
if (torch_is_installed()) {

x = torch_randn(c(3, 3, 2))
x
y = torch_fft(x, 2)
torch_ifft(y, 2)  # recover x
}
```

`torch_imag` *Imag*

Description

Imag

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

imag(input, out=None) -> Tensor

Returns the imaginary part of the `input` tensor.

Warning

Not yet implemented.

$$\text{out}_i = imag(\text{input}_i)$$

Examples

```
if (torch_is_installed()) {
## Not run:
torch_imag(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))

## End(Not run)
}
```

`torch_index_select` *Index_select*

Description

Index_select

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int) the dimension in which we index
`index`	(LongTensor) the 1-D tensor containing the indices to index
`out`	(Tensor, optional) the output tensor.

index_select(input, dim, index, out=None) -> 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_inverse` *Inverse*

Description

Inverse

Arguments

`input`	(Tensor) the input tensor of size $(*, n, n)$ where $*$ is zero or more batch dimensions
`out`	(Tensor, optional) the output tensor.

inverse(input, out=None) -> 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_irfft` *Irfft*

Description

Irfft

Arguments

`input`	(Tensor) the input tensor of at least `signal_ndim + 1` dimensions
`signal_ndim`	(int) the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
`normalized`	(bool, optional) controls whether to return normalized results. Default: `False`
`onesided`	(bool, optional) controls whether `input` was halfed to avoid redundancy, e.g., by `torch_rfft()`. Default: `True`
`signal_sizes`	(list or `torch.Size`, optional) the size of the original signal (without batch dimension). Default: `None`

irfft(input, signal_ndim, normalized=False, onesided=True, signal_sizes=None) -> Tensor

Complex-to-real Inverse Discrete Fourier Transform

This method computes the complex-to-real inverse discrete Fourier transform. It is mathematically equivalent with `torch_ifft` with differences only in formats of the input and output.

The argument specifications are almost identical with `torch_ifft`. Similar to `torch_ifft`, if `normalized` is set to `True`, this normalizes the result by multiplying it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary, where N_i is the size of signal dimension i.

Warning

Generally speaking, input to this function should contain values following conjugate symmetry. Note that even if `onesided` is `True`, often symmetry on some part is still needed. When this requirement is not satisfied, the behavior of `torch_irfft` is undefined. Since `torch_autograd.gradcheck` estimates numerical Jacobian with point perturbations, `torch_irfft` will almost certainly fail the check.

For CPU tensors, this method is currently only available with MKL. Use `torch_backends.mkl.is_available` to check if MKL is installed.

Note

```
Due to the conjugate symmetry, `input` do not need to contain the full
complex frequency values. Roughly half of the values will be sufficient, as
is the case when `input` is given by [`~torch.rfft`] with
``rfft(signal, onesided=True)``. In such case, set the `onesided`
argument of this method to ``True``. Moreover, the original signal shape
information can sometimes be lost, optionally set `signal_sizes` to be
```

the size of the original signal (without the batch dimensions if in batched mode) to recover it with correct shape.

Therefore, to invert an [torch_rfft()], the `normalized` and `onesided` arguments should be set identically for [torch_irfft()], and preferably a `signal_sizes` is given to avoid size mismatch. See the example below for a case of size mismatch.

See [torch_rfft()] for details on conjugate symmetry.

The inverse of this function is torch_rfft().

For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same configuration. See cufft-plan-cache for more details on how to monitor and control the cache.

Examples

```
if (torch_is_installed()) {

x = torch_randn(c(4, 4))
torch_rfft(x, 2, onesided=TRUE)
x = torch_randn(c(4, 5))
torch_rfft(x, 2, onesided=TRUE)
y = torch_rfft(x, 2, onesided=TRUE)
torch_irfft(y, 2, onesided=TRUE, signal_sizes=c(4,5))  # recover x
}
```

---

torch_isfinite *Isfinite*

Description

Isfinite

Arguments

tensor	(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 *Isinf*

Description

Isinf

Arguments

tensor	(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 *Isnan*

Description

Isnan

Arguments

input	(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_is_complex` *Is_complex*

Description

Is_complex

Arguments

`input`	(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

Arguments

`input`	(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_kthvalue *Kthvalue*

Description

Kthvalue

Arguments

input	(Tensor) the input tensor.
k	(int) k for the k-th smallest element
dim	(int, optional) the dimension to find the kth value along
keepdim	(bool) whether the output tensor has dim retained or not.
out	(tuple, optional) the output tuple of (Tensor, LongTensor) can be optionally given to be used as output buffers

kthvalue(input, k, dim=None, keepdim=False, out=None) -> (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.,7.)$resize_(c(2,3))
x
torch_kthvalue(x, 2, 1, TRUE)
}
```

torch_layout *Creates the corresponding layout*

Description

Creates the corresponding layout

Usage

```
torch_strided()

torch_sparse_coo()
```

torch_le *Le*

Description

Le

Arguments

input	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare
out	(Tensor, optional) the output tensor that must be a BoolTensor

le(input, other, out=None) -> Tensor

Computes input $\leq$ other element-wise.

The second argument can be a number or a tensor whose shape is broadcastable with the first argument.

Examples

```
if (torch_is_installed()) {

torch_le(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
         torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

torch_lerp *Lerp*

Description

Lerp

Arguments

input	(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
out	(Tensor, optional) the output tensor.

lerp(input, end, weight, out=None)

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., 5.)
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_lgamma *Lgamma*

Description

Lgamma

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

lgamma(input, out=None) -> 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 *Linspace*

Description

Linspace

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.
out	(Tensor, optional) the output tensor.
dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, 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=None, dtype=None, layout=torch.strided, device=None, 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 *Loads a saved object*

Description

Loads a saved object

Usage

```
torch_load(path)
```

Arguments

path	a path to the saved object

See Also

Other torch_save: `torch_save()`

---

torch_log *Log*

Description

Log

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

log(input, out=None) -> 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` *Log10*

Description

Log10

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

log10(input, out=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

log1p(input, out=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

log2(input, out=None) -> 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_logdet *Logdet*

Description

Logdet

Arguments

input	(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

Arguments

input	(Tensor) the input tensor.
other	(Tensor) the tensor to compute AND with
out	(Tensor, optional) the output tensor.

logical_and(input, other, out=None) -> 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

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

logical_not(input, out=None) -> 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` *Logical_or*

Description

Logical_or

Arguments

`input`	(Tensor) the input tensor.
`other`	(Tensor) the tensor to compute OR with
`out`	(Tensor, optional) the output tensor.

logical_or(input, other, out=None) -> 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

Arguments

input	(Tensor) the input tensor.
other	(Tensor) the tensor to compute XOR with
out	(Tensor, optional) the output tensor.

logical_xor(input, other, out=None) -> 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_logspace *Logspace*

Description

Logspace

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.
out	(Tensor, optional) the output tensor.

dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, 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=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) -> Tensor

Returns a one-dimensional tensor of steps points logarithmically spaced with base base between $\text{base}^{\text{start}}$ and base^{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

Arguments

input	(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.
out	(Tensor, optional) the output tensor.

logsumexp(input, dim, keepdim=False, out=None)

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` *Lstsq*

Description

Lstsq

Arguments

`input`	(Tensor) the matrix B
`A`	(Tensor) the m by n matrix A
`out`	(tuple, optional) the optional destination tensor

lstsq(input, A, out=None) -> 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:

$$\min_X \quad \|AX - B\|_2.$$

If $m < n$, `torch_lstsq()` solves the least-norm problem:

$$\min_X \quad \|X\|_2 \quad \text{subject to} \quad AX = B.$$

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 \eqn{m < n} is not supported on the GPU.

Examples

```
if (torch_is_installed()) {

A = torch_tensor(rbind(
 c(1,1,1),
 c(2,3,4),
 c(3,5,2),
 c(4,2,5),
 c(5,4,3)
))
B = torch_tensor(rbind(
 c(-10, -3),
 c(12, 14),
 c(14, 12),
 c(16, 16),
 c(18, 16)
))
out = torch_lstsq(B, A)
out[[1]]
}
```

---

torch_lt *Lt*

Description

Lt

Arguments

input	(Tensor) the tensor to compare
other	(Tensor or float) the tensor or value to compare
out	(Tensor, optional) the output tensor that must be a BoolTensor

lt(input, other, out=None) -> 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_lt(torch_tensor(matrix(1:4, ncol = 2, byrow=TRUE)),
         torch_tensor(matrix(c(1,1,4,4), ncol = 2, byrow=TRUE)))
}
```

---

torch_lu *LU*

Description

Computes the LU factorization of a matrix or batches of matrices A. Returns a tuple containing the LU factorization and pivots of A. Pivoting is done if pivot is set to True.

Usage

```
torch_lu(A, pivot = TRUE, get_infos = FALSE, out = NULL)
```

Arguments

A	(Tensor) the tensor to factor of size (*, m, n)*(*,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 *Lu_solve*

Description

Lu_solve

Arguments

b	(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.
out	(Tensor, optional) the output tensor.

lu_solve(input, LU_data, LU_pivots, out=None) -> 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_masked_select *Masked_select*

Description

Masked_select

Arguments

input	(Tensor) the input tensor.
mask	(BoolTensor) the tensor containing the binary mask to index with
out	(Tensor, optional) the output tensor.

masked_select(input, mask, out=None) -> 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` *Matmul*

Description

Matmul

Arguments

`input`	(Tensor) the first tensor to be multiplied
`other`	(Tensor) the second tensor to be multiplied
`out`	(Tensor, optional) the output tensor.

matmul(input, other, out=None) -> Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.

- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if `input` is a $(j \times 1 \times n \times m)$ tensor and `other` is a $(k \times m \times p)$ tensor, `out` will be an $(j \times k \times n \times p)$ tensor.

Note

The 1-dimensional dot product version of this function does not support an `out` parameter.

Examples

```
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_power *Matrix_power*

Description

Matrix_power

Arguments

`input`	(Tensor) the input tensor.
`n`	(int) the power to raise the matrix to

matrix_power(input, n) -> Tensor

Returns the matrix raised to the power `n` for square matrices. For batch of matrices, each individual matrix is raised to the power `n`.

If `n` is negative, then the inverse of the matrix (if invertible) is raised to the power `n`. For a batch of matrices, the batched inverse (if invertible) is raised to the power `n`. If `n` is 0, then an identity matrix is returned.

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(2, 2, 2))
a
torch_matrix_power(a, 3)
}
```

`torch_matrix_rank` *Matrix_rank*

Description

Matrix_rank

Arguments

`input`	(Tensor) the input 2-D tensor
`tol`	(float, optional) the tolerance value. Default: `None`
`symmetric`	(bool, optional) indicates whether `input` is symmetric. Default: `False`

matrix_rank(input, tol=None, symmetric=False) -> Tensor

Returns the numerical rank of a 2-D tensor. The method to compute the matrix rank is done using SVD by default. If `symmetric` is `True`, then `input` is assumed to be symmetric, and the computation of the rank is done by obtaining the eigenvalues.

`tol` is the threshold below which the singular values (or the eigenvalues when `symmetric` is `True`) are considered to be 0. If `tol` is not specified, `tol` is set to `S.max() * max(S.size()) * eps` where `S` is the singular values (or the eigenvalues when `symmetric` is `True`), and `eps` is the epsilon value for the datatype of `input`.

Examples

```
if (torch_is_installed()) {

a = torch_eye(10)
torch_matrix_rank(a)
}
```

`torch_max` *Max*

Description

Max

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int) the dimension to reduce.
`keepdim`	(bool) whether the output tensor has `dim` retained or not. Default: `False`.
`out`	(tuple, optional) the result tuple of two output tensors (max, max_indices)
`other`	(Tensor) the second input tensor

max(input) -> Tensor

Returns the maximum value of all elements in the `input` tensor.

max(input, dim, keepdim=False, out=None) -> (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=None) -> 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_mean` *Mean*

Description

Mean

Arguments

`input`	(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.
`out`	(Tensor, optional) the output tensor.

mean(input) -> Tensor

Returns the mean value of all elements in the `input` tensor.

mean(input, dim, keepdim=False, out=None) -> 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` *Median*

Description

Median

Arguments

`input`	(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 result tuple of two output tensors (max, max_indices)

median(input) -> Tensor

Returns the median value of all elements in the `input` tensor.

median(input, dim=-1, keepdim=False, out=None) -> (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` *Meshgrid*

Description

Meshgrid

Arguments

`tensors`	(list of Tensor) list of scalars or 1 dimensional tensors. Scalars will be
`treated`	(1,)

TEST

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.

Examples

```
if (torch_is_installed()) {

x = torch_tensor(c(1, 2, 3))
y = torch_tensor(c(4, 5, 6))
out = torch_meshgrid(list(x, y))
out
}
```

`torch_min` *Min*

Description

Min

Arguments

`input`	(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=None) -> (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=None) -> 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_mm` *Mm*

Description

Mm

Arguments

`input`	(Tensor) the first matrix to be multiplied
`mat2`	(Tensor) the second matrix to be multiplied
`out`	(Tensor, optional) the output tensor.

mm(input, mat2, out=None) -> 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

Arguments

`input`	(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 result tuple of two output tensors (values, indices)

mode(input, dim=-1, keepdim=False, out=None) -> (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_mul` *Mul*

Description

Mul

Arguments

`input`	NA
`value`	(Number) the number to be multiplied to each element of `input`
`out`	NA
`input`	(Tensor) the first multiplicand tensor
`other`	(Tensor) the second multiplicand tensor
`out`	(Tensor, optional) the output tensor.

mul(input, other, out=None)

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` *Multinomial*

Description

Multinomial

Arguments

`input`	(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
`out`	(Tensor, optional) the output tensor.

multinomial(input, num_samples, replacement=False, *, generator=None, out=None) -> 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_mv *Mv*

Description

Mv

Arguments

input	(Tensor) matrix to be multiplied
vec	(Tensor) vector to be multiplied
out	(Tensor, optional) the output tensor.

mv(input, vec, out=None) -> 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

Arguments

input	(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>_) 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_narrow *Narrow*

Description

Narrow

Arguments

input	(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, torch_tensor(0L)$sum(dim = 1), 2)
torch_narrow(x, 2, torch_tensor(1L)$sum(dim = 1), 2)
}
```

---

torch_ne *Ne*

Description

Ne

Arguments

`input`	(Tensor) the tensor to compare
`other`	(Tensor or float) the tensor or value to compare
`out`	(Tensor, optional) the output tensor that must be a `BoolTensor`

ne(input, other, out=None) -> 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 *Neg*

Description

Neg

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

neg(input, out=None) -> Tensor

Returns a new tensor with the negative of the elements of input.

$$\text{out} = -1 \times \text{input}$$

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(5))
a
torch_neg(a)
}
```

torch_nonzero *Nonzero*

Description

Nonzero

Arguments

input	(Tensor) the input tensor.
out	(LongTensor, optional) the output tensor containing indices

nonzero(input, *, out=None, as_tuple=False) -> LongTensor or tuple of LongTensors

When `as_tuple` **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_tuple` **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.

Note

```
[`torch_nonzero(..., as_tuple=False) <torch.nonzero>`] (default) returns a
2-D tensor where each row is the index for a nonzero value.

[`torch_nonzero(..., as_tuple=True) <torch.nonzero>`] returns a tuple of 1-D
index tensors, allowing for advanced indexing, so ``x[x.nonzero(as_tuple=True)]``
gives all nonzero values of tensor ``x``. Of the returned tuple, each index tensor
contains nonzero indices for a certain dimension.

See below for more details on the two behaviors.
```

Examples

```
if (torch_is_installed()) {

torch_nonzero(torch_tensor(c(1, 1, 1, 0, 1)))
}
```

`torch_norm` *Norm*

Description

Norm

Arguments

input	(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 ===== ========== ========== None Frobenius norm 2-norm 'fro' Frobenius norm – 'nuc' nuclear norm – Other as vec norm when dim is None 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 None, 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 = None and out = None. Default: False
out	(Tensor, optional) the output tensor. Ignored if dim = None and out = None.
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: None.

TEST

Returns the matrix norm or vector norm of a given tensor.

Examples

```
if (torch_is_installed()) {

a = torch_arange(0, 9, dtype = torch_float())
b = a$reshape(list(3, 3))
torch_norm(a)
torch_norm(b)
torch_norm(a, Inf)
torch_norm(b, Inf)

}
```

---

torch_normal *Normal*

Description

Normal

Arguments

mean	(Tensor) the tensor of per-element means
std	(Tensor) the tensor of per-element standard deviations
generator	(torch.Generator, optional) a pseudorandom number generator for sampling
out	(Tensor, optional) the output tensor.
size	(int...) a sequence of integers defining the shape of the output tensor.

normal(mean, std, *, generator=None, out=None) -> 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, out=None) -> Tensor

Similar to the function above, but the means are shared among all drawn elements.

normal(mean, std=1.0, out=None) -> Tensor

Similar to the function above, but the standard-deviations are shared among all drawn elements.

normal(mean, std, size, *, out=None) -> 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()) {

## Not run:
torch_normal(mean=0, std=torch_arange(1, 0, -0.1))


torch_normal(mean=0.5, std=torch_arange(1., 6.))


torch_normal(mean=torch_arange(1., 6.))


torch_normal(2, 3, size=list(1, 4))
```

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

`torch_ones` *Ones*

Description

Ones

Arguments

`size`	(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.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) -> Tensor

Returns a tensor filled with the scalar value `1`, with the shape defined by the variable argument `size`.

Examples

```
if (torch_is_installed()) {

torch_ones(c(2, 3))
torch_ones(c(5))
}
```

torch_ones_like *Ones_like*

Description

Ones_like

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 None, defaults to the dtype of input.
layout	(torch.layout, optional) the desired layout of returned tensor. Default: if None, defaults to the layout of input.
device	(torch.device, optional) the desired device of returned tensor. Default: if None, 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=None, layout=None, device=None, 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 *Orgqr*

Description

Orgqr

Arguments

input	(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 *Ormqr*

Description

Ormqr

Arguments

input	(Tensor) the a from torch_geqrf.
input2	(Tensor) the tau from torch_geqrf.
input3	(Tensor) the matrix to be multiplied.

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. See LAPACK documentation for ormqr_ for further details.

torch_pdist *Pdist*

Description

Pdist

Arguments

input	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[:, None] - 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 *Pinverse*

Description

Pinverse

Arguments

input	(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

Arguments

input	(Tensor) the input tensor
upscale_factor	(int) factor to increase spatial resolution by

Rearranges elements in a tensor of shape

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

Arguments

input	(Tensor) the input tensor containing the rates of the Poisson distribution
generator	(torch.Generator, optional) a pseudorandom number generator for sampling

poisson(input *, generator=None) -> 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_polygamma *Polygamma*

Description

Polygamma

Arguments

n	(int) the order of the polygamma function
input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

polygamma(n, input, out=None) -> Tensor

Computes the n^{th} derivative of the digamma function on input. $n \geq 0$ is called the order of the polygamma function.

$$\psi^{(n)}(x) = \frac{d^{(n)}}{dx^{(n)}}\psi(x)$$

Note

This function is not implemented for \eqn{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

Arguments

input	(Tensor) the input tensor.
exponent	(float or tensor) the exponent value
out	(Tensor, optional) the output tensor.
self	(float) the scalar base value for the power operation

pow(input, exponent, out=None) -> 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=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`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: None.
`dim`	(int) the dimension to reduce.
`keepdim`	(bool) whether the output tensor has `dim` retained or not.

prod(input, dtype=None) -> Tensor

Returns the product of all elements in the `input` tensor.

prod(input, dim, keepdim=False, dtype=None) -> 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

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

Arguments

input	(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.
out	(tuple, optional) tuple of Q and R tensors satisfying input = torch.matmul(Q,R). The dimensions of Q and R are $(*, m, k)$ and $(*, k, n)$ respectively, where $k = \min(m, n)$ if some: is True and $k = m$ otherwise.

qr(input, some=True, out=None) -> (Tensor, Tensor)

Computes the QR decomposition of a matrix or a batch of matrices input, and returns a namedtuple (Q, R) of tensors such that input $= QR$ with Q being an orthogonal matrix or batch of orthogonal matrices and R being an upper triangular matrix or batch of upper triangular matrices.

If some is True, then this function returns the thin (reduced) QR factorization. Otherwise, if some is False, this function returns the complete QR factorization.

Note

precision may be lost if the magnitudes of the elements of input are large

While it should always give you a valid decomposition, it may not give you the same one across platforms - it will depend on your LAPACK implementation.

Examples

```
if (torch_is_installed()) {

a = torch_tensor(matrix(c(12., -51, 4, 6, 167, -68, -4, 24, -41), ncol = 3, byrow = TRUE))
out = torch_qr(a)
q = out[[1]]
r = out[[2]]
torch_mm(q, r)$round()
torch_mm(q$t(), q)$round()
}
```

torch_qscheme *Creates the corresponding Scheme object*

Description

Creates the corresponding Scheme object

Usage

```
torch_per_channel_affine()

torch_per_tensor_affine()

torch_per_channel_symmetric()

torch_per_tensor_symmetric()
```

torch_quantize_per_channel *Quantize_per_channel*

Description

Quantize_per_channel

Arguments

input	(Tensor) float tensor to quantize
scales	(Tensor) float 1D tensor of scales to use, size should match input.size(axis)
zero_points	(int) integer 1D tensor of offset to use, size should match input.size(axis)
axis	(int) dimension on which apply per-channel quantization
dtype	(torch.dtype) the desired data type of returned tensor. Has to be one of the quantized dtypes: torch_quint8, torch.qint8, torch.qint32

quantize_per_channel(input, scales, zero_points, axis, dtype) -> Tensor

Converts a float tensor to per-channel quantized tensor with given scales and zero points.

Examples

```
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

Arguments

`input`	(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_rand` *Rand*

Description

Rand

Arguments

`size`	(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.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 None, 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=None, dtype=None, layout=torch.strided, device=None, 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 *Randint*

Description

Randint

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
out	(Tensor, optional) the output tensor.
dtype	(torch.dtype, optional) the desired data type of returned tensor. Default: if None, 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 None, 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.

**randint(low=0, high, size, *, generator=None, out=None, **

dtype=None, layout=torch.strided, device=None, 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

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 `None`, defaults to the dtype of `input`.
`layout`	(`torch.layout`, optional) the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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`.

randint_like(input, low=0, high, dtype=None, layout=torch.strided, device=None, 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

Arguments

`size`	(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.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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=None, dtype=None, layout=torch.strided, device=None, 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` *Randn_like*

Description

Randn_like

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 `None`, defaults to the dtype of `input`.
`layout`	(`torch.layout`, optional) the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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=None, layout=None, device=None, requires_grad=False, memory_format=torch.preserve_format -> Tensor

Returns a tensor with the same size as `input` that is filled with random numbers from a normal distribution with mean 0 and variance 1. `torch_randn_like(input)` is equivalent to `torch_randn(input.size(),dtype=input`

`torch_randperm` *Randperm*

Description

Randperm

Arguments

`n`	(int) the upper bound (exclusive)
`out`	(Tensor, optional) the output tensor.
`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 `None`, 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=None, dtype=torch.int64, layout=torch.strided, device=None, 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` *Rand_like*

Description

Rand_like

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 `None`, defaults to the dtype of `input`.
`layout`	(`torch.layout`, optional) the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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=None, layout=None, device=None, 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.dtyp`

`torch_range` *Range*

Description

Range

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`.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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=None, dtype=None, layout=torch.strided, device=None, 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 *Real*

Description

Real

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

real(input, out=None) -> 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.

$$\text{out}_i = real(\text{input}_i)$$

Examples

```
if (torch_is_installed()) {
## Not run:
torch_real(torch_tensor(c(-1 + 1i, -2 + 2i, 3 - 3i)))

## End(Not run)
}
```

torch_reciprocal *Reciprocal*

Description

Reciprocal

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

reciprocal(input, out=None) -> 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` *Creates the reduction objet*

Description

Creates the reduction objet

Usage

```
torch_reduction_sum()

torch_reduction_mean()

torch_reduction_none()
```

`torch_relu_` *Relu_*

Description

Relu_

relu_(input) -> Tensor

In-place version of `torch_relu`.

torch_remainder *Remainder*

Description

Remainder

Arguments

input	(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
out	(Tensor, optional) the output tensor.

remainder(input, other, out=None) -> 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 *Renorm*

Description

Renorm

Arguments

input	(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
out	(Tensor, optional) the output tensor.

renorm(input, p, dim, maxnorm, out=None) -> 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

Arguments

`input`	(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.

repeat_interleave(input, repeats, dim=None) -> 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 *Reshape*

Description

Reshape

Arguments

input	(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, 4)
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 *Result_type*

Description

Result_type

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(tensor = torch_tensor(c(1, 2), dtype=torch_int()), 1.0)
}
```

torch_rfft *Rfft*

Description

Rfft

Arguments

input	(Tensor) the input tensor of at least signal_ndim dimensions
signal_ndim	(int) the number of dimensions in each signal. signal_ndim can only be 1, 2 or 3
normalized	(bool, optional) controls whether to return normalized results. Default: False
onesided	(bool, optional) controls whether to return half of results to avoid redundancy. Default: True

rfft(input, signal_ndim, normalized=False, onesided=True) -> Tensor

Real-to-complex Discrete Fourier Transform

This method computes the real-to-complex discrete Fourier transform. It is mathematically equivalent with `torch_fft` with differences only in formats of the input and output.

This method supports 1D, 2D and 3D real-to-complex transforms, indicated by `signal_ndim`. `input` must be a tensor with at least `signal_ndim` dimensions with optionally arbitrary number of leading batch dimensions. If `normalized` is set to `True`, this normalizes the result by dividing it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary, where N_i is the size of signal dimension i.

The real-to-complex Fourier transform results follow conjugate symmetry:

$$X[\omega_1, \ldots, \omega_d] = X^*[N_1 - \omega_1, \ldots, N_d - \omega_d],$$

where the index arithmetic is computed modulus the size of the corresponding dimension, * is the conjugate operator, and d = `signal_ndim`. `onesided` flag controls whether to avoid redundancy in the output results. If set to `True` (default), the output will not be full complex result of shape $(*, 2)$, where $*$ is the shape of `input`, but instead the last dimension will be halfed as of size $\lfloor \frac{N_d}{2} \rfloor + 1$.

The inverse of this function is `torch_irfft`.

Warning

For CPU tensors, this method is currently only available with MKL. Use `torch_backends.mkl.is_available` to check if MKL is installed.

Note

```
For CUDA tensors, an LRU cache is used for cuFFT plans to speed up
repeatedly running FFT methods on tensors of same geometry with same
configuration. See cufft-plan-cache for more details on how to
monitor and control the cache.
```

Examples

```
if (torch_is_installed()) {

x = torch_randn(c(5, 5))
torch_rfft(x, 2)
torch_rfft(x, 2, onesided=FALSE)
}
```

`torch_roll` *Roll*

Description

Roll

Arguments

input	(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=None) -> 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 *Rot90*

Description

Rot90

Arguments

input	(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(0, 4)$view(c(2, 2))
x
torch_rot90(x, 1, c(1, 2))
x = torch_arange(0, 8)$view(c(2, 2, 2))
x
torch_rot90(x, 1, c(1, 2))
}
```

---

torch_round *Round*

Description

Round

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

round(input, out=None) -> 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_ *Rrelu_*

Description

Rrelu_

rrelu_(input, lower=1./8, upper=1./3, training=False) -> Tensor

In-place version of `torch_rrelu`.

torch_rsqrt *Rsqrt*

Description

Rsqrt

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

rsqrt(input, out=None) -> 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_rsqrt(a)
}
```

torch_save *Saves an object to a disk file.*

Description

This function is experimental, don't use for long term storage.

Usage

```
torch_save(obj, path, ...)
```

Arguments

obj	the saved object
path	a connection or the name of the file to save.
...	not currently used.

See Also

Other torch_save: torch_load()

`torch_selu_` *Selu_*

Description

Selu_

selu_(input) -> Tensor

In-place version of `toch_selu`.

`torch_set_default_dtype` *Gets and sets the default floating point dtype.*

Description

Gets and sets the default floating point dtype.

Usage

```
torch_set_default_dtype(d)

torch_get_default_dtype()
```

Arguments

`d`	The default floating point dtype to set. Initially set to `torch_float()`.

`torch_sigmoid` *Sigmoid*

Description

Sigmoid

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

sigmoid(input, out=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

sign(input, out=None) -> 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_sin *Sin*

Description

Sin

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

sin(input, out=None) -> 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

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

sinh(input, out=None) -> 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` *Slogdet*

Description

Slogdet

Arguments

`input`	(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_solve *Solve*

Description

Solve

Arguments

input	(Tensor) input matrix B of size $(*, m, k)$, where $*$ is zero or more batch dimensions.
A	(Tensor) input square matrix of size $(*, m, m)$, where $*$ is zero or more batch dimensions.
out	((Tensor, Tensor) optional output tuple.

torch.solve(input, A, out=None) -> (Tensor, Tensor)

This function returns the solution to the system of linear equations represented by $AX = B$ and the LU factorization of A, in order as a namedtuple solution, LU.

LU contains L and U factors for LU factorization of A.

torch_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 solution, LU.

Note

```
Irrespective of the original strides, the returned matrices
`solution` and `LU` will be transposed, i.e. with strides like
`B.contiguous().transpose(-1, -2).stride()` and
`A.contiguous().transpose(-1, -2).stride()` respectively.
```

Examples

```
if (torch_is_installed()) {

A = torch_tensor(rbind(c(6.80, -2.11,  5.66,  5.97,  8.23),
                       c(-6.05, -3.30,  5.36, -4.44,  1.08),
                       c(-0.45,  2.58, -2.70,  0.27,  9.04),
                       c(8.32,  2.71,  4.35,  -7.17,  2.14),
                       c(-9.67, -5.14, -7.26,  6.08, -6.87)))$t()
B = torch_tensor(rbind(c(4.02,  6.19, -8.22, -7.57, -3.03),
                       c(-1.56,  4.00, -8.67,  1.75,  2.86),
                       c(9.81, -4.09, -4.57, -8.61,  8.99)))$t()
out = torch_solve(B, A)
X = out[[1]]
LU = out[[2]]
torch_dist(B, torch_mm(A, X))
# Batched solver example
```

```
A = torch_randn(c(2, 3, 1, 4, 4))
B = torch_randn(c(2, 3, 1, 4, 6))
out = torch_solve(B, A)
X = out[[1]]
LU = out[[2]]
torch_dist(B, A$matmul(X))
}
```

`torch_sort` *Sort*

Description

Sort

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int, optional) the dimension to sort along
`descending`	(bool, optional) controls the sorting order (ascending or descending)
`out`	(tuple, optional) the output tuple of (`Tensor`, `LongTensor`) that can be optionally given to be used as output buffers

sort(input, dim=-1, descending=False, out=None) -> (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

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 None, infers data type from values.
device	(torch.device, optional) the desired device of returned tensor. Default: if None, 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=None, dtype=None, device=None, 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

Split

Arguments

`tensor`	(Tensor) tensor to split.
`split_size_or_sections`	(int) size of a single chunk or list of sizes for each chunk
`dim`	(int) dimension along which to split the tensor.

TEST

Splits the tensor into chunks. Each chunk is a view of the original tensor.

```
If `split_size_or_sections` 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_or_sections` is a list, then `tensor` will be split
into ``len(split_size_or_sections)`` chunks with sizes in `dim` according
to `split_size_or_sections`.
```

torch_sqrt *Sqrt*

Description

Sqrt

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

sqrt(input, out=None) -> 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 *Square*

Description

Square

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

square(input, out=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`dim`	(int, optional) if given, the input will be squeezed only in this dimension
`out`	(Tensor, optional) the output tensor.

squeeze(input, dim=None, out=None) -> 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 *Stack*

Description

Stack

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)
out	(Tensor, optional) the output tensor.

stack(tensors, dim=0, out=None) -> Tensor

Concatenates sequence of tensors along a new dimension.

All tensors need to be of the same size.

torch_std *Std*

Description

Std

Arguments

input	(Tensor) the input tensor.
unbiased	(bool) whether to use the unbiased estimation or not
dim	(int or tuple of ints) the dimension or dimensions to reduce.
keepdim	(bool) whether the output tensor has dim retained or not.
out	(Tensor, optional) the output tensor.

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=None) -> 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)
}
```

`torch_std_mean` *Std_mean*

Description

Std_mean

Arguments

`input`	(Tensor) the input tensor.
`unbiased`	(bool) whether to use the unbiased estimation or not
`dim`	(int or tuple of ints) the dimension or dimensions to reduce.
`keepdim`	(bool) whether the output tensor has `dim` retained or not.

std_mean(input, unbiased=True) -> (Tensor, Tensor)

Returns the standard-deviation and mean of all elements in the `input` tensor.

If `unbiased` is `False`, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

std_mean(input, dim, unbiased=True, keepdim=False) -> (Tensor, Tensor)

Returns the standard-deviation and mean of each row of the `input` tensor in the dimension `dim`. If `dim` is a list of dimensions, reduce over all of them.

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see `torch_squeeze`), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).

If `unbiased` is `False`, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessel's correction will be used.

Examples

```
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

Arguments

`input`	(Tensor) the input tensor
`n_fft`	(int) size of Fourier transform
`hop_length`	(int, optional) the distance between neighboring sliding window frames. Default: `None` (treated as equal to `floor(n_fft / 4)`)
`win_length`	(int, optional) the size of window frame and STFT filter. Default: `None` (treated as equal to `n_fft`)
`window`	(Tensor, optional) the optional window function. Default: `None` (treated as window of all 1 s)
`center`	(bool, optional) whether to pad `input` on both sides so that the t-th frame is centered at time $t \times$ hop_length. Default: `True`
`pad_mode`	(string, optional) controls the padding method used when `center` is `True`. Default: `"reflect"`

normalized (bool, optional) controls whether to return the normalized STFT results Default: `False`

onesided (bool, optional) controls whether to return half of results to avoid redundancy Default: `True`

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 ω is the frequency that $0 \leq \omega < \text{n\_fft}$. 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 ``None`` (default), it is treated as equal to
  ``floor(n_fft / 4)``.

* If `win_length` is ``None`` (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 ``None`` (default), it is
  treated as if having \eqn{1} everywhere in the window. If
  \eqn{\mbox{win_length} < \mbox{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 \eqn{t}-th frame is centered at time
  \eqn{t \times \mbox{hop_length}}. Otherwise, the \eqn{t}-th frame
  begins at time  \eqn{t \times \mbox{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 \eqn{\omega}
 in \eqn{\left[0, 1, 2, ..., \left\lfloor \frac{\mbox{n_fft}}{2} \right\rfloor + 1\right]}
  are returned because the real-to-complex Fourier transform satisfies the
  conjugate symmetry, i.e., \eqn{X[m, \omega] = X[m, \mbox{n_fft} - \omega]^*}.
```

```
* If `normalized` is ``True`` (default is ``False``), the function
  returns the normalized STFT results, i.e., multiplied by \eqn{(\mbox{frame_length})^{-0.5}}.

Returns the real and the imaginary parts together as one tensor of size
\eqn{(* \times N \times T \times 2)}, where \eqn{*} is the optional
batch size of `input`, \eqn{N} is the number of frequencies where
STFT is applied, \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_sum` *Sum*

Description

Sum

Arguments

`input`	(Tensor) the input tensor.
`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: None.
`dim`	(int or tuple of ints) the dimension or dimensions to reduce.
`keepdim`	(bool) whether the output tensor has `dim` retained or not.

sum(input, dtype=None) -> Tensor

Returns the sum of all elements in the `input` tensor.

sum(input, dim, keepdim=False, dtype=None) -> 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(0, 4 * 5 * 6)$view(c(4, 5, 6))
torch_sum(b, list(2, 1))
}
```

`torch_svd` *Svd*

Description

Svd

Arguments

`input`	(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
`out`	(tuple, optional) the output tuple of tensors

svd(input, some=True, compute_uv=True, out=None) -> (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_symeig *Symeig*

Description

Symeig

Arguments

`input`	(Tensor) the input tensor of size $(*, n, n)$ where $*$ is zero or more batch dimensions consisting of symmetric matrices.
`eigenvectors`	(boolean, optional) controls whether eigenvectors have to be computed
`upper`	(boolean, optional) controls whether to consider upper-triangular or lower-triangular region
`out`	(tuple, optional) the output tuple of (Tensor, Tensor)

symeig(input, eigenvectors=False, upper=True, out=None) -> (Tensor, Tensor)

This function returns eigenvalues and eigenvectors of a real symmetric matrix `input` or a batch of real symmetric matrices, represented by a namedtuple (eigenvalues, eigenvectors).

This function calculates all eigenvalues (and vectors) of `input` such that input $= V\text{diag}(e)V^T$.

The boolean argument `eigenvectors` defines computation of both eigenvectors and eigenvalues or eigenvalues only.

If it is `False`, only eigenvalues are computed. If it is `True`, both eigenvalues and eigenvectors are computed.

Since the input matrix `input` is supposed to be symmetric, only the upper triangular portion is used by default.

If `upper` is `False`, then lower triangular portion is used.

Note

The eigenvalues are returned in ascending order. If `input` is a batch of matrices, then the eigenvalues of each matrix in the batch is returned in ascending order.

Irrespective of the original strides, the returned matrix `V` will be transposed, i.e. with strides V.contiguous().transpose(-1, -2).stride().

Extra care needs to be taken when backward through outputs. Such operation is really only stable when all eigenvalues are distinct. Otherwise, `NaN` can appear as the gradients are not properly defined.

Examples

```
if (torch_is_installed()) {

a = torch_randn(c(5, 5))
a = a + a$t()  # To make a symmetric
a
o = torch_symeig(a, eigenvectors=TRUE)
e = o[[1]]
v = o[[2]]
e
v
a_big = torch_randn(c(5, 2, 2))
a_big = a_big + a_big$transpose(-2, -1)  # To make a_big symmetric
o = a_big$symeig(eigenvectors=TRUE)
e = o[[1]]
v = o[[2]]
torch_allclose(torch_matmul(v, torch_matmul(e$diag_embed(), v$transpose(-2, -1))), a_big)
}
```

torch_t *T*

Description

T

Arguments

input	(Tensor) the input tensor.

t(input) -> Tensor

Expects `input` to be <= 2-D tensor and transposes dimensions 0 and 1.

0-D and 1-D tensors are returned as is. When input is a 2-D tensor this is equivalent to `transpose(input,0,1)`.

Examples

```
if (torch_is_installed()) {

x = torch_randn(c(2,3))
x
torch_t(x)
x = torch_randn(c(3))
x
torch_t(x)
x = torch_randn(c(2, 3))
x
torch_t(x)
}
```

torch_take *Take*

Description

Take

Arguments

input	(Tensor) the input tensor.
indices	(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(0, 2, 5), dtype = torch_int64()))
}
```

`torch_tan` *Tan*

Description

Tan

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

tan(input, out=None) -> 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

Arguments

`input`	(Tensor) the input tensor.
`out`	(Tensor, optional) the output tensor.

tanh(input, out=None) -> 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` *Converts R objects to a torch tensor*

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 torch_dtype instance
`device`	a device creted with `torch_device()`
`requires_grad`	if autograd should record operations on the returned tensor.
`pin_memory`	If set, returned tensor would be allocated in the pinned memory.

Examples

```
if (torch_is_installed()) {
torch_tensor(c(1,2,3,4))
torch_tensor(c(1,2,3,4), dtype = torch_int())

}
```

`torch_tensordot` *Tensordot*

Description

Tensordot

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

TEST

Returns a contraction of a and b over multiple dimensions.

`tensordot` implements a generalized matrix product.

Examples

```
if (torch_is_installed()) {

a = torch_arange(start = 0, end = 60.)$reshape(c(3, 4, 5))
b = torch_arange(start = 0, end = 24.)$reshape(c(4, 3, 2))
torch_tensordot(a, b, dims_self=c(1, 0), dims_other = c(0, 1))
## 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_threshold_` *Threshold_*

Description

Threshold_

threshold_(input, threshold, value) -> Tensor

In-place version of `torch_threshold`.

torch_topk *Topk*

Description

Topk

Arguments

input	(Tensor) the input tensor.
k	(int) the k in "top-k"
dim	(int, optional) the dimension to sort along
largest	(bool, optional) controls whether to return largest or smallest elements
sorted	(bool, optional) controls whether to return the elements in sorted order
out	(tuple, optional) the output tuple of (Tensor, LongTensor) that can be optionally given to be used as output buffers

topk(input, k, dim=None, largest=True, sorted=True, out=None) -> (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 *Trace*

Description

Trace

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., 10.)$view(c(3, 3))
x
torch_trace(x)
}
```

torch_transpose *Transpose*

Description

Transpose

Arguments

input	(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

Arguments

y	(Tensor) The values of the function to integrate
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.
dx	(float) The distance between points at which y is sampled.

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

Arguments

input	(Tensor) multiple right-hand sides of size $(*, m, k)$ where $*$ is zero of more batch dimensions (b)
A	(Tensor) the input triangular coefficient matrix of size $(*, m, m)$ where $*$ is zero or more batch dimensions
upper	(bool, optional) whether to solve the upper-triangular system of equations (default) or the lower-triangular system of equations. Default: True.
transpose	(bool, optional) whether A should be transposed before being sent into the solver. Default: False.
unitriangular	(bool, optional) whether A is unit triangular. If True, the diagonal elements of A are assumed to be 1 and not referenced from A. Default: False.

triangular_solve(input, A, upper=True, transpose=False, unitriangular=False) -> (Tensor, Tensor)

Solves a system of equations with a triangular coefficient matrix A and multiple right-hand sides b.

In particular, solves $AX = b$ and assumes A is upper-triangular with the default keyword arguments.

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

Arguments

input	(Tensor) the input tensor.
diagonal	(int, optional) the diagonal to consider
out	(Tensor, optional) the output tensor.

tril(input, diagonal=0, out=None) -> 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

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 `None`, `torch_long`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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 \eqn{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

Arguments

`input`	(Tensor) the input tensor.
`diagonal`	(int, optional) the diagonal to consider
`out`	(Tensor, optional) the output tensor.

triu(input, diagonal=0, out=None) -> 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

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 `None`, `torch_long`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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 \eqn{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` *True_divide*

Description

True_divide

Arguments

`dividend`	(Tensor) the dividend
`divisor`	(Tensor or Scalar) the divisor

true_divide(dividend, divisor) -> Tensor

Performs "true division" that always computes the division in floating point. Analogous to division in Python 3 and equivalent to `torch_div` except when both inputs have bool or integer scalar types, in which case they are cast to the default (floating) scalar type before the division.

$$\text{out}_i = \frac{\text{dividend}_i}{\text{divisor}}$$

Examples

```
if (torch_is_installed()) {

dividend = torch_tensor(c(5, 3), dtype=torch_int())
divisor = torch_tensor(c(3, 2), dtype=torch_int())
torch_true_divide(dividend, divisor)
torch_true_divide(dividend, 2)
}
```

---

torch_trunc *Trunc*

Description

Trunc

Arguments

input	(Tensor) the input tensor.
out	(Tensor, optional) the output tensor.

trunc(input, out=None) -> 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 *Unbind*

Description

Unbind

Arguments

input	(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

Arguments

`input`	(Tensor) the input tensor
`return_inverse`	(bool) Whether to also return the indices for where elements in the original input ended up in the returned unique list.
`return_counts`	(bool) Whether to also return the counts for each unique element.
`dim`	(int) the dimension to apply unique. If `None`, the unique of the flattened input is returned. default: `None`

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_unsqueeze *Unsqueeze*

Description

Unsqueeze

Arguments

input	(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_var *Var*

Description

Var

Arguments

input	(Tensor) the input tensor.
unbiased	(bool) whether to use the unbiased estimation or not
dim	(int or tuple of ints) the dimension or dimensions to reduce.
keepdim	(bool) whether the output tensor has `dim` retained or not.
out	(Tensor, optional) the output tensor.

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=None) -> 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` *Var_mean*

Description

Var_mean

Arguments

`input`	(Tensor) the input tensor.
`unbiased`	(bool) whether to use the unbiased estimation or not
`dim`	(int or tuple of ints) the dimension or dimensions to reduce.
`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_where` *Where*

Description

Where

Arguments

`condition`	(BoolTensor) When True (nonzero), yield x, otherwise yield y
`x`	(Tensor) values selected at indices where `condition` is `True`
`y`	(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:

$$\text{out}_i = \begin{cases} \text{x}_i & \text{if condition}_i \\ \text{y}_i & \text{otherwise} \end{cases}$$

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

Arguments

`size`	(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.
`out`	(Tensor, optional) the output tensor.
`dtype`	(`torch.dtype`, optional) the desired data type of returned tensor. Default: if `None`, 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 `None`, 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=None, dtype=None, layout=torch.strided, device=None, 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

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 `None`, defaults to the dtype of `input`.
`layout`	(`torch.layout`, optional) the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
`device`	(`torch.device`, optional) the desired device of returned tensor. Default: if `None`, 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=None, layout=None, device=None, 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_enable_grad *Enable grad*

Description

Context-manager that enables gradient calculation. Enables gradient calculation, if it has been disabled via with_no_grad.

Usage

```
with_enable_grad(code)
```

Arguments

code	code to be executed with gradient recording.

Details

This context manager is thread local; it will not affect computation in other threads.

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 *Temporarily modify gradient recording.*

Description

Temporarily modify gradient recording.

Usage

```
with_no_grad(code)
```

Arguments

code	code to be executed with no gradient recording.

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