PyTorch

Table of Contents

TORCH.STD

torch.std(*input*, *dim=None*, ***, *correction=1*, *keepdim=False*, *out=None*) → Tensor

Calculates the standard deviation over the dimensions specified by `dim`. `dim` can be a single dimension, list of dimensions, or `None` to reduce over all dimensions.

The standard deviation (σ) is calculated as

$$\sigma = \sqrt{\frac{1}{N - \delta N}\sum_{i=0}^{N-1}(x_i - \bar{x})^2}$$

where x is the sample set of elements, $\bar{x}$ is the sample mean, N is the number of samples and δN is the `correction`.

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).

Parameters:

- **input** (*Tensor*) – the input tensor.
- **dim** (*int or tuple of ints*) – the dimension or dimensions to reduce.

Keyword Arguments:

- **correction** (*int*) –

  difference between the sample size and sample degrees of freedom. Defaults to Bessel's correction, `correction=1`.

  Changed in version 2.0: Previously this argument was called `unbiased` and was a boolean with `True` corresponding to `correction=1` and `False` being `correction=0`.

- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not.
- **out** (*Tensor, optional*) – the output tensor.

Example

```
>>> a = torch.tensor(
...     [[ 0.2035,  1.2959,  1.8101, -0.4644],
...      [ 1.5027, -0.3270,  0.5905,  0.6538],
...      [-1.5745,  1.3330, -0.5596, -0.6548],
...      [ 0.1264, -0.5080,  1.6420,  0.1992]])
>>> torch.std(a, dim=1, keepdim=True)
tensor([[1.0311],
        [0.7477],
        [1.2204],
        [0.9087]])
```

Previous Next

PyTorch

Table of Contents

TORCH.MEAN

torch.mean(*input*, ***, *dtype=None*) → Tensor

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

Parameters:

input (*Tensor*) – the input tensor.

Keyword Arguments:

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.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.2294, -0.5481,  1.3288]])
>>> torch.mean(a)
tensor(0.3367)
```

torch.mean(*input*, *dim*, *keepdim=False*, ***, *dtype=None*, *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).

Parameters:

- **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.

Keyword Arguments:

- **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.

• SEE ALSO

`torch.nanmean()` computes the mean value of *non-NaN* elements.

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.3841,  0.6320,  0.4254, -0.7384],
        [-0.9644,  1.0131, -0.6549, -1.4279],
        [-0.2951, -1.3350, -0.7694,  0.5600],
        [ 1.0842, -0.9580,  0.3623,  0.2343]])
>>> torch.mean(a, 1)
tensor([-0.0163, -0.5085, -0.4599,  0.1807])
>>> torch.mean(a, 1, True)
tensor([[-0.0163],
        [-0.5085],
        [-0.4599],
        [ 0.1807]])
```

< Previous Next >

PyTorch

Table of Contents

LINEAR

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

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

This module supports TensorFloat32.

On certain ROCm devices, when using float16 inputs this module will use different precision for backward.

Parameters:

- **in_features** (*int*) – size of each input sample
- **out_features** (*int*) – size of each output sample
- **bias** (*bool*) – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(*, H_{in})$ where $*$ means any number of dimensions including none and $H_{in} = \text{in\_features}$.
- Output: $(*, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables:

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, 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:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

< Previous Next >

PyTorch

Table of Contents

ADAM

CLASS torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, ***, *foreach=None*, *maximize=False*, *capturable=False*, *differentiable=False*, *fused=None*) [SOURCE]

Implements Adam algorithm.

```
input : $\gamma$ (lr), $\beta_1, \beta_2$ (betas), $\theta_0$ (params), $f(\theta)$ (objective)
        $\lambda$ (weight decay), $amsgrad$, $maximize$
initialize : $m_0 \leftarrow 0$ ( first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ to $\ldots$ do
    if $maximize$ :
        $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
    else
        $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    if $\lambda \neq 0$
        $g_t \leftarrow g_t + \lambda \theta_{t-1}$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
    $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
    if $amsgrad$
        $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
    else
        $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return $\theta_t$
```

For further details regarding the algorithm we refer to Adam: A Method for Stochastic Optimization.

Parameters:

- **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** (*bool*, *optional*) – whether to use the AMSGrad variant of this algorithm from the paper On the Convergence of Adam and Beyond (default: False)
- **foreach** (*bool*, *optional*) – whether foreach implementation of optimizer is used. If unspecified by the user (so foreach is None), we will try to use foreach over the for-loop implementation on CUDA, since it is usually significantly more performant. (default: None)
- **maximize** (*bool*, *optional*) – maximize the params based on the objective, instead of minimizing (default: False)
- **capturable** (*bool*, *optional*) – whether this instance is safe to capture in a CUDA graph. Passing True can impair ungraphed performance, so if you don't intend to graph capture this instance, leave it False (default: False)
- **differentiable** (*bool*, *optional*) – whether autograd should occur through the optimizer step in training. Otherwise, the step() function runs in a torch.no_grad() context. Setting to True can impair performance, so leave it False if you don't intend to run autograd through this instance (default: False)
- **fused** (*bool*, *optional*) – whether the fused implementation (CUDA only) is used. Currently, *torch.float64*, *torch.float32*, *torch.float16*, and *torch.bfloat16* are supported. (default: None)

• NOTE

The foreach and fused implementations are typically faster than the for-loop, single-tensor implementation. Thus, if the user has not specified BOTH flags (i.e., when foreach = fused = None), we will attempt defaulting to the foreach implementation when the tensors are all on CUDA. For example, if the user specifies True for fused but nothing for foreach, we will run the fused implementation. If the user specifies False for foreach but nothing for fused (or False for fused but nothing for foreach), we will run the for-loop implementation. If the user specifies True for both foreach and fused, we will prioritize fused over foreach, as it is typically faster. We attempt to use the fastest, so the hierarchy goes fused -> foreach -> for-loop. HOWEVER, since the fused implementation is relatively new, we want to give it sufficient bake-in time, so we default to foreach and NOT fused when the user has not specified either flag.

add_param_group(*param_group*)

Add a param group to the `Optimizer` s *param_groups*.

This can be useful when fine tuning a pre-trained network as frozen layers can be made trainable and added to the `Optimizer` as training progresses.

Parameters:

param_group (*dict*) – Specifies what Tensors should be optimized along with group specific optimization options.

load_state_dict(*state_dict*)

Loads the optimizer state.

Parameters:

state_dict (*dict*) – optimizer state. Should be an object returned from a call to `state_dict()`.

register_step_post_hook(*hook*)

Register an optimizer step post hook which will be called after optimizer step. It should have the following signature:

```
hook(optimizer, args, kwargs) -> None
```

The `optimizer` argument is the optimizer instance being used.

Parameters:

hook (*Callable*) – The user defined hook to be registered.

Returns:

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type:

`torch.utils.hooks.RemoveableHandle`

register_step_pre_hook(*hook*)

Register an optimizer step pre hook which will be called before optimizer step. It should have the following signature:

```
hook(optimizer, args, kwargs) -> None or modified args and kwargs
```

The `optimizer` argument is the optimizer instance being used. If args and kwargs are modified by the pre-hook, then the transformed values are returned as a tuple containing the new_args and new_kwargs.

Parameters:

hook (*Callable*) – The user defined hook to be registered.

Returns:

a handle that can be used to remove the added hook by calling `handle.remove()`

Return type:

`torch.utils.hooks.RemoveableHandle`

state_dict()

Returns the state of the optimizer as a `dict`.

It contains two entries:

- state - a dict holding current optimization state. Its content

  differs between optimizer classes.

- param_groups - a list containing all parameter groups where each

  parameter group is a dict

zero_grad(*set_to_none=True*)

Sets the gradients of all optimized `torch.Tensor` s to zero.

Parameters:

set_to_none (*bool*) – instead of setting to zero, set the grads to None. This will in general have lower memory footprint, and can modestly improve performance. However, it changes certain behaviors. For example: 1. When the user tries to access a gradient and perform manual ops on it, a None attribute or a Tensor full of 0s will behave differently. 2. If the user requests `zero_grad(set_to_none=True)` followed by a backward pass, `.grad`s are guaranteed to be None for params that did not receive a gradient. 3. `torch.optim` optimizers have a different behavior if the gradient is 0 or None (in one case it does the step with a gradient of 0 and in the other it skips the step altogether).

PyTorch

Table of Contents

MSELOSS

CLASS torch.nn.MSELoss(*size_average=None*, *reduce=None*, *reduction='mean'*) [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

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

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

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

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

x and y are tensors of arbitrary shapes with a total of n elements each.

The mean operation still operates over all the elements, and divides by n.

The division by n can be avoided if one sets `reduction = 'sum'`.

Parameters:

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when `reduce` is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*str, 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'`

Shape:

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

Examples:

```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

< Previous   Next >

PyTorch

Table of Contents

RELU

CLASS torch.nn.ReLU(*inplace=False*) [SOURCE]

Applies the rectified linear unit function element-wise:

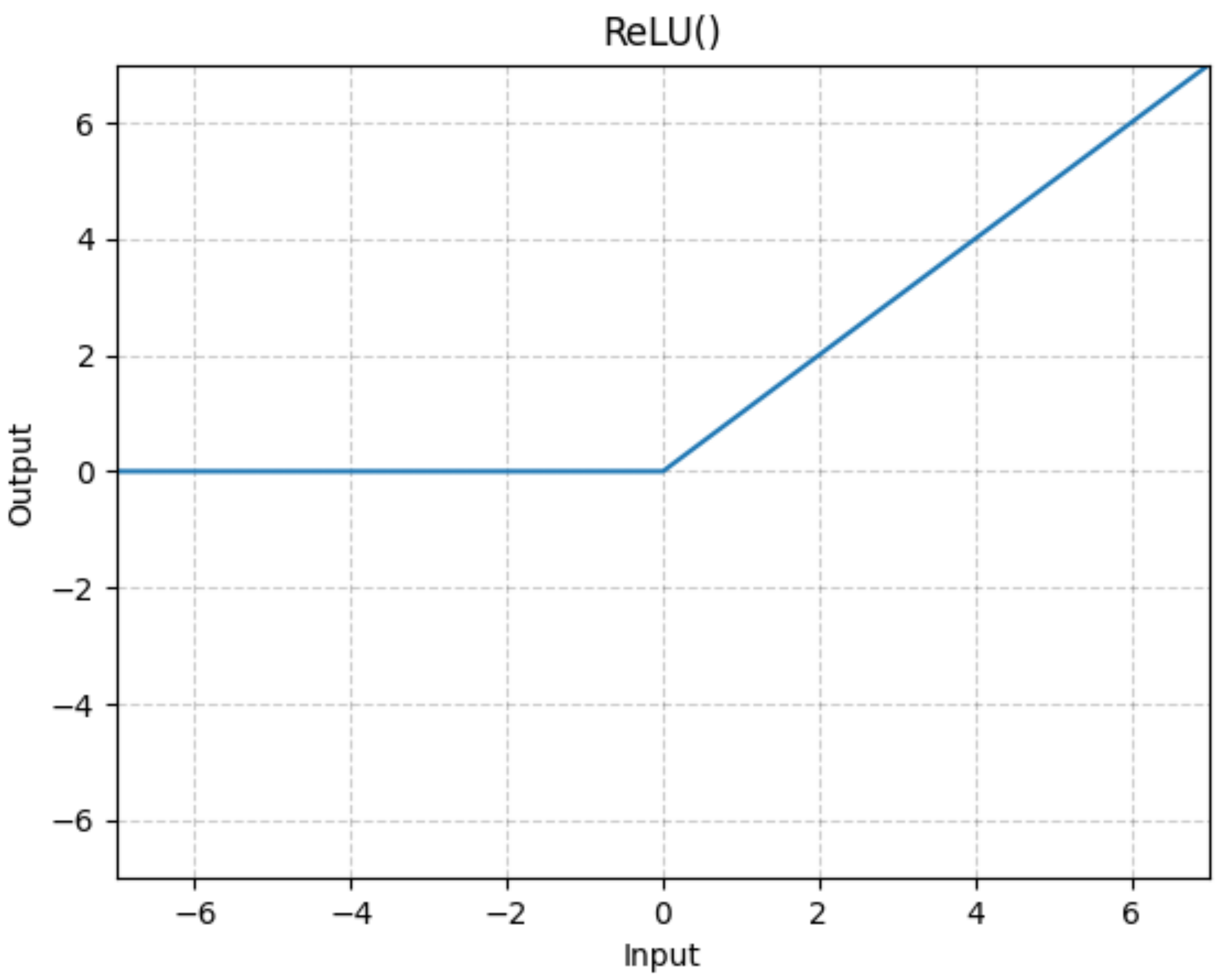
$$\text{ReLU}(x) = (x)^+ = \max(0, x)$$

Parameters:

inplace (*bool*) – can optionally do the operation in-place. Default: `False`

Shape:

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



Examples:

```
  >>> m = nn.ReLU()
  >>> input = torch.randn(2)
  >>> output = m(input)


An implementation of CReLU - https://arxiv.org/abs/1603.05201

  >>> m = nn.ReLU()
  >>> input = torch.randn(2).unsqueeze(0)
  >>> output = torch.cat((m(input), m(-input)))
```

< Previous Next >

PyTorch

Table of Contents

SEQUENTIAL

CLASS torch.nn.Sequential(*args: Module) [SOURCE]

CLASS torch.nn.Sequential(arg: OrderedDict[str, Module])

A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an `OrderedDict` of modules can be passed in. The `forward()` method of `Sequential` accepts any input and forwards it to the first module it contains. It then "chains" outputs to inputs sequentially for each subsequent module, finally returning the output of the last module.

The value a `Sequential` provides over manually calling a sequence of modules is that it allows treating the whole container as a single module, such that performing a transformation on the `Sequential` applies to each of the modules it stores (which are each a registered submodule of the `Sequential`).

What's the difference between a `Sequential` and a `torch.nn.ModuleList`? A `ModuleList` is exactly what it sounds like–a list for storing `Module` s! On the other hand, the layers in a `Sequential` are connected in a cascading way.

Example:

```
# Using Sequential to create a small model. When `model` is run,
# input will first be passed to `Conv2d(1,20,5)`. The output of
# `Conv2d(1,20,5)` will be used as the input to the first
# `ReLU`; the output of the first `ReLU` will become the input
# for `Conv2d(20,64,5)`. Finally, the output of
# `Conv2d(20,64,5)` will be used as input to the second `ReLU`
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Using Sequential with OrderedDict. This is functionally the
# same as the above code
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

append(module) [SOURCE]

Appends a given module to the end.

Parameters:

module (*nn.Module*) – module to append

Return type:

Sequential

< Previous Next >