Ta le of Contents

TORCH.NN.F NCTIONAL

Convolution functions

conv1

`torch.nn.functional.conv1d(`*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*`)` → Tensor

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

This operator supports TensorFloat32.

See `Conv1d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kW)$
- **ias** – optional ias of shape (out_channels). Default: `None`
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a one-element tuple *(sW,)*. Default: 1
- **a ing** – implicit pa ings on oth si es of the input. Can e a single num er or a one-element tuple *(padW,)*. Default: 0
- **ilation** – the spacing etween kernel elements. Can e a single num er or a one-element tuple *(dW,)*. Default: 1
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1

Examples:

```
>>> filters = torch.randn(33, 16, 3)
>>> inputs = torch.randn(20, 16, 50)
>>> F.conv1d(inputs, filters)
```

conv2

`torch.nn.functional.conv2d(`*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*`)` → Tensor

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

This operator supports TensorFloat32.

See `Conv2d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kH, kW)$
- **ias** – optional ias tensor of shape (out_channels). Default: `None`
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a tuple *(sH, sW)*. Default: 1
- **a ing** – implicit pa ings on oth si es of the input. Can e a single num er or a tuple *(padH, padW)*. Default: 0
- **ilation** – the spacing etween kernel elements. Can e a single num er or a tuple *(dH, dW)*. Default: 1
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1

Examples:

```
>>> # With square kernels and equal stride
>>> filters = torch.randn(8,4,3,3)
>>> inputs = torch.randn(1,4,5,5)
>>> F.conv2d(inputs, filters, padding=1)
```

conv3

conv3

`torch.nn.functional.conv3d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)` → Tensor

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

This operator supports TensorFloat32.

See `Conv3d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kT, kH, kW)$
- **ias** – optional ias tensor of shape (out_channels). Default: None
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a tuple *(sT, sH, sW)*. Default: 1
- **a ing** – implicit pa ings on oth si es of the input. Can e a single num er or a tuple *(padT, padH, padW)*. Default: 0
- **ilation** – the spacing etween kernel elements. Can e a single num er or a tuple *(dT, dH, dW)*. Default: 1
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1

Examples:

```
>>> filters = torch.randn(33, 16, 3, 3, 3)
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> F.conv3d(inputs, filters)
```

conv_trans ose1

`torch.nn.functional.conv_transpose1d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1)` → Tensor

Applies a 1D transpose convolution operator over an input signal compose of several input planes, sometimes also calle " econvolution".

This operator supports TensorFloat32.

See `ConvTranspose1d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kW)$
- **ias** – optional ias of shape (out_channels). Default: None
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a tuple `(sW,)`. Default: 1
- **a ing** – `dilation * (kernel_size - 1) - padding` zero-pa ing will e a e to oth si es of each imension in the input. Can e a single num er or a tuple `(padW,)`. Default: 0
- **out ut_ a ing** – a itional size a e to one si e of each imension in the output shape. Can e a single num er or a tuple `(out_padW)`. Default: 0
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1
- **ilation** – the spacing etween kernel elements. Can e a single num er or a tuple `(dW,)`. Default: 1

Examples:

```
>>> inputs = torch.randn(20, 16, 50)
>>> weights = torch.randn(16, 33, 5)
>>> F.conv_transpose1d(inputs, weights)
```

conv_trans ose2

`torch.nn.functional.conv_transpose2d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1)` → Tensor

Applies a 2D transpose convolution operator over an input image compose of several input planes, sometimes also calle " econvolution".

This operator supports TensorFloat32.

See `ConvTranspose2d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kH, kW)$
- **ias** – optional ias of shape (out_channels). Default: None
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a tuple `(sH, sW)`. Default: 1
- **a ing** – `dilation * (kernel_size - 1) - padding` zero-pa ing will e a e to oth si es of each imension in the input. Can e a single num er or a tuple `(padH, padW)`. Default: 0
- **out ut_ a ing** – a itional size a e to one si e of each imension in the output shape. Can e a single num er or a tuple `(out_padH, out_padW)`. Default: 0
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1
- **ilation** – the spacing etween kernel elements. Can e a single num er or a tuple `(dH, dW)`. Default: 1

Examples:

```
>>> # With square kernels and equal stride
>>> inputs = torch.randn(1, 4, 5, 5)
>>> weights = torch.randn(4, 8, 3, 3)
>>> F.conv_transpose2d(inputs, weights, padding=1)
```

conv_trans ose3

`torch.nn.functional.conv_transpose3d`(*input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1*) → Tensor

Applies a 3D transpose convolution operator over an input image compose of several input planes, sometimes also calle " econvolution"

This operator supports TensorFloat32.

See `ConvTranspose3d` for etails an output shape.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kT, kH, kW)$
- **ias** – optional ias of shape (out_channels). Default: None
- **stri e** – the stri e of the convolving kernel. Can e a single num er or a tuple `(sT, sH, sW)`. Default: 1
- **a ing** – `dilation * (kernel_size - 1) - padding` zero-pa ing will e a e to oth si es of each imension in the input. Can e a single num er or a tuple `(padT, padH, padW)`. Default: 0
- **out ut_ a ing** – a itional size a e to one si e of each imension in the output shape. Can e a single num er or a tuple `(out_padT, out_padH, out_padW)`. Default: 0
- **grou s** – split input into groups, in_channels shoul e ivisi le y the num er of groups. Default: 1
- **ilation** – the spacing etween kernel elements. Can e a single num er or a tuple *(dT, dH, dW)*. Default: 1

Examples:

```
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> weights = torch.randn(16, 33, 3, 3, 3)
>>> F.conv_transpose3d(inputs, weights)
```

unfol

`torch.nn.functional.unfold`(*input, kernel_size, dilation=1, padding=0, stride=1*) [SO RCE]

Extracts sli ing local locks from an atche input tensor.

• WARNING

Currently, only 4-D input tensors (atche image-like tensors) are supporte .

• WARNING

More than one element of the unfol e tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorize) may result in incorrect ehavior. If you nee to write to the tensor, please clone it first.

See `torch.nn.Unfold` for etails

fol

`torch.nn.functional.fold(input, output_size, kernel_size, dilation=1, padding=0, stride=1)` [SO RCE]

Com ines an array of sli ing local locks into a large containing tensor.

• WARNING

Currently, only 3-D output tensors (unfol e atche image-like tensors) are supporte .

See `torch.nn.Fold` for etails

Pooling functions

avg_ ool1

`torch.nn.functional.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 compose of several input planes.

See `AvgPool1d` for etails an output shape.

Parameters

- **in ut** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **kernel_size** – the size of the win ow. Can e a single num er or a tuple *(kW,)*
- **stri e** – the stri e of the win ow. Can e a single num er or a tuple *(sW,)*. Default: `kernel_size`
- **a ing** – implicit zero pa ings on oth si es of the input. Can e a single num er or a tuple *(padW,)*. Default: 0
- **ceil_mo e** – when True, will use *ceil* instea of *floor* to compute the output shape. Default: `False`
- **count_inclu e_ a** – when True, will inclu e the zero-pa ing in the averaging calculation. Default: `True`

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> input = torch.tensor([[[1, 2, 3, 4, 5, 6, 7]]], dtype=torch.float32)
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
tensor([[[ 2.,  4.,  6.]]])
```

avg_ ool2

`torch.nn.functional.avg_pool2d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)` → Tensor

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

See `AvgPool2d` for etails an output shape.

Parameters

- **in ut** – input tensor $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **kernel_size** – size of the pooling region. Can e a single num er or a tuple *(kH, kW)*
- **stri e** – stri e of the pooling operation. Can e a single num er or a tuple *(sH, sW)*. Default: `kernel_size`
- **a ing** – implicit zero pa ings on oth si es of the input. Can e a single num er or a tuple *(padH, padW)*. Default: 0
- **ceil_mo e** – when True, will use *ceil* instea of *floor* in the formula to compute the output shape. Default: `False`
- **count_inclu e_ a** – when True, will inclu e the zero-pa ing in the averaging calculation. Default: `True`
- **ivisor_overri e** – if specifie , it will e use as ivisor, otherwise size of the pooling region will e use . Default: None

avg_ ool3

`torch.nn.functional.avg_pool3d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)` → Tensor

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

See `AvgPool3d` for etails an output shape.

Parameters

- **in ut** – input tensor $(\text{minibatch}, \text{in\_channels}, iT \times iH, iW)$
- **kernel_size** – size of the pooling region. Can e a single num er or a tuple *(kT, kH, kW)*
- **stri e** – stri e of the pooling operation. Can e a single num er or a tuple *(sT, sH, sW)*. Default: `kernel_size`
- **a ing** – implicit zero pa ings on oth si es of the input. Can e a single num er or a tuple *(padT, padH, padW)*, Default: 0
- **ceil_mo e** – when True, will use *ceil* instea of *floor* in the formula to compute the output shape
- **count_inclu e_ a** – when True, will inclu e the zero-pa ing in the averaging calculation
- **ivisor_overri e** – if specifie , it will e use as ivisor, otherwise size of the pooling region will e use . Default: None

max_ ool1

`torch.nn.functional.max_pool1d(*args, **kwargs)`

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

See `MaxPool1d` for etails.

max_ ool2

`torch.nn.functional.max_pool2d(*args, **kwargs)`

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

See `MaxPool2d` for etails.

max_ ool3

`torch.nn.functional.max_pool3d(*args, **kwargs)`

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

See `MaxPool3d` for etails.

max_un ool1

`torch.nn.functional.max_unpool1d(input, indices, kernel_size, stride=None, padding=0, output_size=None)` [SO RCE]

Computes a partial inverse of `MaxPool1d`.

See `MaxUnpool1d` for etails.

max_un ool2

`torch.nn.functional.max_unpool2d(input, indices, kernel_size, stride=None, padding=0, output_size=None)` [SO RCE]

Computes a partial inverse of `MaxPool2d`.

See `MaxUnpool2d` for etails.

max_un ool3

`torch.nn.functional.max_unpool3d(input, indices, kernel_size, stride=None, padding=0, output_size=None)` [SO RCE]

Computes a partial inverse of `MaxPool3d`.

See `MaxUnpool3d` for etails.

l _ ool1

`torch.nn.functional.lp_pool1d(input, norm_type, kernel_size, stride=None, ceil_mode=False)` [SO RCE]

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

See `LPPool1d` for etails.

l _ ool2

`torch.nn.functional.lp_pool2d(input, norm_type, kernel_size, stride=None, ceil_mode=False)` [SO RCE]

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

See `LPPool2d` for etails.

a a tive_max_ ool1

`torch.nn.functional.adaptive_max_pool1d(*args, **kwargs)`

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

See `AdaptiveMaxPool1d` for etails an output shape.

Parameters

- **out ut_size** – the target output size (single integer)
- **return_in ices** – whether to return pooling in ices. Default: `False`

a a tive_max_ ool2

`torch.nn.functional.adaptive_max_pool2d(*args, **kwargs)`

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

See `AdaptiveMaxPool2d` for etails an output shape.

Parameters

- **out ut_size** – the target output size (single integer or ou le-integer tuple)
- **return_in ices** – whether to return pooling in ices. Default: `False`

a a tive_max_ ool3

`torch.nn.functional.adaptive_max_pool3d(*args, **kwargs)`

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

See `AdaptiveMaxPool3d` for etails an output shape.

Parameters

- **out ut_size** – the target output size (single integer or triple-integer tuple)
- **return_in ices** – whether to return pooling in ices. Default: `False`

a a tive_avg_ ool1

`torch.nn.functional.adaptive_avg_pool1d(input, output_size)` → Tensor

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

See `AdaptiveAvgPool1d` for etails an output shape.

Parameters

out ut_size – the target output size (single integer)

a a tive_avg_ ool2

`torch.nn.functional.adaptive_avg_pool2d(input, output_size)` [SO RCE]

Applies a 2D a aptive average pooling over an input signal compose of several input planes.

See `AdaptiveAvgPool2d` for etails an output shape.

Parameters

out ut_size – the target output size (single integer or ou le-integer tuple)

a a tive_avg_ ool3

`torch.nn.functional.adaptive_avg_pool3d(input, output_size)` [SO RCE]

Applies a 3D a aptive average pooling over an input signal compose of several input planes.

See `AdaptiveAvgPool3d` for etails an output shape.

Parameters

out ut_size – the target output size (single integer or triple-integer tuple)

Non-linear activation functions

threshol

`torch.nn.functional.threshold(input, threshold, value, inplace=False)`

Threshol s each element of the input Tensor.

See `Threshold` for more etails.

`torch.nn.functional.threshold_(input, threshold, value)` → Tensor

In-place version of `threshold()` .

relu

`torch.nn.functional.relu(input, inplace=False)` → Tensor [SO RCE]

Applies the rectifie linear unit function element-wise. See `ReLU` for more etails.

`torch.nn.functional.relu_(input)` → Tensor

In-place version of `relu()` .

har tanh

`torch.nn.functional.hardtanh(input, min_val=-1., max_val=1., inplace=False)` → Tensor [SO RCE]

Applies the Har Tanh function element-wise. See `Hardtanh` for more etails.

`torch.nn.functional.hardtanh_(input, min_val=-1., max_val=1.)` → Tensor

In-place version of `hardtanh()` .

har swish

`torch.nn.functional.hardswish(input: torch.Tensor, inplace: bool = False)` → torch.Tensor [SO RCE]

Applies the har swish function, element-wise, as escri e in the paper:

Searching for Mo ileNetV3.

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

See `Hardswish` for more etails.

relu6

`torch.nn.functional.relu6(input, inplace=False)` → Tensor [SO RCE]

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

See `ReLU6` for more etails.

elu

`torch.nn.functional.elu(input, alpha=1.0, inplace=False)` [SO RCE]

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

See `ELU` for more etails.

`torch.nn.functional.elu_(input, alpha=1.)` → Tensor

In-place version of `elu()`.

selu

`torch.nn.functional.selu(input, inplace=False)` → Tensor [SO RCE]

Applies element-wise, $\text{SELU}(x) = scale * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1)))$, with $\alpha = 1.6732632423543772848170429916717$ an $scale = 1.0507009873554804934193349852946$.

See `SELU` for more etails.

celu

`torch.nn.functional.celu(input, alpha=1., inplace=False)` → Tensor [SO RCE]

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

See `CELU` for more etails.

leaky_relu

`torch.nn.functional.leaky_relu(input, negative_slope=0.01, inplace=False)` → Tensor [SO RCE]

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

See `LeakyReLU` for more etails.

`torch.nn.functional.leaky_relu_(input, negative_slope=0.01)` → Tensor

In-place version of `leaky_relu()`.

relu

`torch.nn.functional.prelu(input, weight)` → Tensor [SO RCE]

Applies element-wise the function $\text{PReLU}(x) = \max(0, x) + \text{weight} * \min(0, x)$ where weight is a learna le parameter.

See `PReLU` for more etails.

rrelu

`torch.nn.functional.rrelu(input, lower=1./8, upper=1./3, training=False, inplace=False)` → Tensor [SO RCE]

Ran omize leaky ReL .

See `RReLU` for more etails.

`torch.nn.functional.rrelu_(input, lower=1./8, upper=1./3, training=False)` → Tensor

In-place version of `rrelu()`.

glu

`torch.nn.functional.glu(input, dim=-1)` → Tensor [SO RCE]

The gate linear unit. Computes:

$$\text{GLU}(a, b) = a \otimes \sigma(b)$$

where *input* is split in half along *dim* to form a an b, σ is the sigmoi function an $\otimes$ is the element-wise pro uct etween matrices.

See Language Mo eling with Gate Convolutional Networks.

Parameters

- **in ut** (*Tensor*) – input tensor
- **im** (*int*) – imension on which to split the input. Default: -1

gelu

`torch.nn.functional.gelu(input)` → Tensor [SO RCE]

Applies element-wise the function $\text{GELU}(x) = x * \Phi(x)$

where $\Phi(x)$ is the Cumulative Distri ution Function for Gaussian Distri ution.

See Gaussian Error Linear nits (GEL s).

logsigmoi

`torch.nn.functional.logsigmoid(input)` → Tensor

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

See `LogSigmoid` for more etails.

har shrink

`torch.nn.functional.hardshrink(input, lambd=0.5)` → Tensor [SO RCE]

Applies the har shrinkage function element-wise

See `Hardshrink` for more etails.

tanhshrink

`torch.nn.functional.tanhshrink(input)` → Tensor [SO RCE]

Applies element-wise, $\text{Tanhshrink}(x) = x - \text{Tanh}(x)$

See `Tanhshrink` for more etails.

softsign

`torch.nn.functional.softsign(input)` → Tensor [SO RCE]

Applies element-wise, the function $\text{SoftSign}(x) = \frac{x}{1 + |x|}$

See `Softsign` for more etails.

soft lus

`torch.nn.functional.softplus(input, beta=1, threshold=20)` → Tensor

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

For numerical sta ility the implementation reverts to the linear function when $input \times \beta > threshold$.

See `Softplus` for more etails.

softmin

`torch.nn.functional.softmin(input, dim=None, _stacklevel=3, dtype=None)` [SO RCE]

Applies a softmin function.

Note that $\text{Softmin}(x) = \text{Softmax}(-x)$. See softmax efinition for mathematical formula.

See `Softmin` for more etails.

Parameters

- **in ut** (*Tensor*) – input
- **im** (*int*) – A imension along which softmin will e compute (so every slice along im will sum to 1).
- **ty e** (`torch.dtype`, optional) – the esire ata type of returne tensor. If specifie , the input tensor is caste to `dtype` efore the operation is performe . This is useful for preventing ata type overflows. Default: None.

softmax

`torch.nn.functional.softmax(input, dim=None, _stacklevel=3, dtype=None)` [SO RCE]

Applies a softmax function.

Softmax is efine as:

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

It is applie to all slices along im, an will re-scale them so that the elements lie in the range [0, 1] an sum to 1.

See `Softmax` for more etails.

Parameters

- **in ut** (*Tensor*) – input
- **im** (*int*) – A imension along which softmax will e compute .
- **ty e** (`torch.dtype` , optional) – the esire ata type of returne tensor. If specifie , the input tensor is caste to `dtype` efore the operation is performe . This is useful for preventing ata type overflows. Default: None.

• NOTE

This function oesn't work irectly with NLLLoss, which expects the Log to e compute etween the Softmax an itself. se log_softmax instea (it's faster an has etter numerical properties).

softshrink

`torch.nn.functional.softshrink(`*input*, *lambd=0.5*`)` → Tensor

Applies the soft shrinkage function elementwise

See `Softshrink` for more etails.

gum el_softmax

`torch.nn.functional.gumbel_softmax(`*logits*, *tau=1*, *hard=False*, *eps=1e-10*, *dim=-1*`)` [SO RCE]

Samples from the Gum el-Softmax istri ution (Link 1 Link 2) an optionally iscretizes.

Parameters

- **logits** – [..., *num_features*] unnormalize log pro a ilities
- **tau** – non-negative scalar temperature
- **har** – if `True` , the returne samples will e iscretize as one-hot vectors, ut will e ifferentiate as if it is the soft sample in autogra
- **im** (*int*) – A imension along which softmax will e compute . Default: -1.

Returns

Sample tensor of same shape as *logits* from the Gum el-Softmax istri ution. If `hard=True` , the returne samples will e one-hot, otherwise they will e pro a ility istri utions that sum to 1 across *dim*.

• NOTE

This function is here for legacy reasons, may e remove from nn.Functional in the future.

• NOTE

The main trick for *hard* is to o *y_hard - y_soft.detach() + y_soft*

It achieves two things: - makes the output value exactly one-hot (since we a then su tract y_soft value) - makes the gra ient equal to y_soft gra ient (since we strip all other gra ients)

Examples::

```
>>> logits = torch.randn(20, 32)
>>> # Sample soft categorical using reparametrization trick:
>>> F.gumbel_softmax(logits, tau=1, hard=False)
>>> # Sample hard categorical using "Straight-through" trick:
>>> F.gumbel_softmax(logits, tau=1, hard=True)
```

log_softmax

`torch.nn.functional.log_softmax(`*input*, *dim=None*, *_stacklevel=3*, *dtype=None*`)` [SO RCE]

Applies a softmax followe y a logarithm.

While mathematically equivalent to log(softmax(x)), oing these two operations separately is slower, an numerically unsta le. This function uses an alternative formulation to compute the output an gra ient correctly.

See `LogSoftmax` for more etails.

Parameters

- **in ut** (*Tensor*) – input
- **im** (*int*) – A imension along which log_softmax will e compute .
- **ty e** (`torch.dtype` , optional) – the esire ata type of returne tensor. If specifie , the input tensor is caste to `dtype` efore the operation is performe . This is useful for preventing ata type overflows. Default: None.

tanh

`torch.nn.functional.tanh(`*input*`)` → Tensor [SO RCE]

Applies element-wise, $\text{Tanh}(x) = \tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$

See `Tanh` for more etails.

sigmoi

`torch.nn.functional.sigmoid(`*input*`)` → Tensor [SO RCE]

Applies the element-wise function $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$

See `Sigmoid` for more etails.

har sigmoi

`torch.nn.functional.hardsigmoid(`*input*`)` → Tensor [SO RCE]

Applies the element-wise function

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \le -3, \\ 1 & \text{if } x \ge +3, \\ x/6 + 1/2 & \text{otherwise} \end{cases}$$

Parameters

in lace – If set to `True`, will o this operation in-place. Default: `False`

See `Hardsigmoid` for more etails.

silu

`torch.nn.functional.silu(`*input*, *inplace=False*`)` [SO RCE]

Applies the silu function, element-wise.

$$\text{silu}(x) = x * \sigma(x), \text{where } \sigma(x) \text{ is the logistic sigmoid.}$$

• NOTE

See Gaussian Error Linear nits (GEL s) where the SiL (Sigmoi Linear nit) was originally coine , an see Sigmoi -Weighte Linear nits for Neural Network Function Approximation in Reinforcement Learning an Swish: a Self-Gate Activation Function where the SiL was experimente with later.

See `SiLU` for more etails.

Normalization functions

atch_norm

`torch.nn.functional.batch_norm(`*input*, *running_mean*, *running_var*, *weight=None*, *bias=None*, *training=False*, *momentum=0.1*, *eps=1e-05*`)` [SO RCE]

Applies Batch Normalization for each channel across a atch of ata.

See `BatchNorm1d`, `BatchNorm2d`, `BatchNorm3d` for etails.

instance_norm

`torch.nn.functional.instance_norm(`*input*, *running_mean=None*, *running_var=None*, *weight=None*, *bias=None*, *use_input_stats=True*, *momentum=0.1*, *eps=1e-05*`)` [SO RCE]

Applies Instance Normalization for each channel in each ata sample in a atch.

See `InstanceNorm1d`, `InstanceNorm2d`, `InstanceNorm3d` for etails.

layer_norm

`torch.nn.functional.layer_norm(`*input*, *normalized_shape*, *weight=None*, *bias=None*, *eps=1e-05*`)` [SO RCE]

Applies Layer Normalization for last certain num er of imensions.

See `LayerNorm` for etails.

local_res onse_norm

`torch.nn.functional.local_response_norm(`*input*, *size*, *alpha=0.0001*, *beta=0.75*, *k=1.0*`)` [SO RCE]

Applies local response normalization over an input signal compose of several input planes, where channels occupy the secon imension. Applies normalization across channels.

See `LocalResponseNorm` for etails.

normalize

`torch.nn.functional.normalize(`*input*, *p=2*, *dim=1*, *eps=1e-12*, *out=None*`)` [SO RCE]

Performs L_p normalization of inputs over specifie imension.

For a tensor `input` of sizes $(n_0, \ldots, n_{dim}, \ldots, n_k)$, each n_{dim} -element vector v along imension `dim` is transforme as

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

With the efault arguments it uses the Eucli ean norm over vectors along imension 1 for normalization.

Parameters

- **in ut** – input tensor of any shape
- (*float*) – the exponent value in the norm formulation. Default: 2
- **im** (*int*) – the imension to re uce. Default: 1
- **e s** (*float*) – small value to avoi ivision y zero. Default: 1e-12
- **out** (*Tensor, optional*) – the output tensor. If `out` is use , this operation won't e ifferentia le.

Linear functions

linear

`torch.nn.functional.linear(input, weight, bias=None)` [SO RCE]

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

This operator supports TensorFloat32.

Shape:

- Input: $(N, *, in\_features)$ N is the atch size, * means any num er of a itional imensions
- Weight: $(out\_features, in\_features)$
- Bias: $(out\_features)$
- Output: $(N, *, out\_features)$

ilinear

`torch.nn.functional.bilinear(input1, input2, weight, bias=None)` [SO RCE]

Applies a ilinear transformation to the incoming ata: $y = x_1^T A x_2 + b$

Shape:

- input1: $(N, *, H_{in1})$ where $H_{in1} = in1\_features$ an $*$ means any num er of a itional imensions. All ut the last imension of the inputs shoul e the same.
- input2: $(N, *, H_{in2})$ where $H_{in2} = in2\_features$
- weight: $(out\_features, in1\_features, in2\_features)$
- ias: $(out\_features)$
- output: $(N, *, H_{out})$ where $H_{out} = out\_features$ an all ut the last imension are the same shape as the input.

Dro out functions

ro out

`torch.nn.functional.dropout(input, p=0.5, training=True, inplace=False)` [SO RCE]

During training, ran omly zeroes some of the elements of the input tensor with pro a ility `p` using samples from a Bernoulli istri ution.

See `Dropout` for etails.

Parameters

- – pro a ility of an element to e zeroe . Default: 0.5
- **training** – apply ropout if is `True`. Default: `True`
- **in lace** – If set to `True`, will o this operation in-place. Default: `False`

al ha_ ro out

`torch.nn.functional.alpha_dropout(input, p=0.5, training=False, inplace=False)` [SO RCE]

Applies alpha ropout to the input.

See `AlphaDropout` for etails.

feature_al ha_ ro out

`torch.nn.functional.feature_alpha_dropout(input, p=0.5, training=False, inplace=False)` [SO RCE]

Ran omly masks out entire channels (a channel is a feature map, e.g. the j -th channel of the i -th sample in the atch input is a tensor $\text{input}[i, j]$) of the input tensor). Instea of setting activations to zero, as in regular Dropout, the activations are set to the negative saturation value of the SEL activation function.

Each element will e maske in epen ently on every forwar call with pro a ility `p` using samples from a Bernoulli istri ution. The elements to e maske are ran omize on every forwar call, an scale an shifte to maintain zero mean an unit variance.

See `FeatureAlphaDropout` for etails.

Parameters

- – ropout pro a ility of a channel to e zeroe . Default: 0.5
- **training** – apply ropout if is `True` . Default: `True`
- **in lace** – If set to `True` , will o this operation in-place. Default: `False`

ro out2

`torch.nn.functional.dropout2d(`*input*, *p=0.5*, *training=True*, *inplace=False*`)` [SO RCE]

Ran omly zero out entire channels (a channel is a 2D feature map, e.g., the j -th channel of the i -th sample in the atche input is a 2D tensor $\text{input}[i, j]$) of the input tensor). Each channel will e zeroe out in epen ently on every forwar call with pro a ility `p` using samples from a Bernoulli istri ution.

See `Dropout2d` for etails.

Parameters

- – pro a ility of a channel to e zeroe . Default: 0.5
- **training** – apply ropout if is `True` . Default: `True`
- **in lace** – If set to `True` , will o this operation in-place. Default: `False`

ro out3

`torch.nn.functional.dropout3d(`*input*, *p=0.5*, *training=True*, *inplace=False*`)` [SO RCE]

Ran omly zero out entire channels (a channel is a 3D feature map, e.g., the j -th channel of the i -th sample in the atche input is a 3D tensor $\text{input}[i, j]$) of the input tensor). Each channel will e zeroe out in epen ently on every forwar call with pro a ility `p` using samples from a Bernoulli istri ution.

See `Dropout3d` for etails.

Parameters

- – pro a ility of a channel to e zeroe . Default: 0.5
- **training** – apply ropout if is `True` . Default: `True`
- **in lace** – If set to `True` , will o this operation in-place. Default: `False`

S arse functions

em e ing

`torch.nn.functional.embedding(`*input*, *weight*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*`)` [SO RCE]

A simple lookup ta le that looks up em e ings in a fixe ictionary an size.

This mo ule is often use to retrieve wor em e ings using in ices. The input to the mo ule is a list of in ices, an the em e ing matrix, an the output is the correspon ing wor em e ings.

See `torch.nn.Embedding` for more etails.

Parameters

- **in ut** (*LongTensor*) – Tensor containing in ices into the em e ing matrix
- **weight** (*Tensor*) – The em e ing matrix with num er of rows equal to the maximum possi le in ex + 1, an num er of columns equal to the em e ing size
- **a ing_i x** (*int, optional*) – If given, pa s the output with the em e ing vector at `padding_idx` (initialize to zeros) whenever it encounters the in ex.
- **max_norm** (*float, optional*) – If given, each em e ing vector with norm larger than `max_norm` is renormalize to have norm `max_norm` . Note: this will mo ify `weight` in-place.
- **norm_ty e** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2` .
- **scale_gra _ y_fre** (*boolean, optional*) – If given, this will scale gra ients y the inverse of frequency of the wor s in the mini- atch. Default `False` .
- **s arse** (*bool, optional*) – If `True` , gra ient w.r.t. `weight` will e a sparse tensor. See Notes un er `torch.nn.Embedding` for more etails regar ing sparse gra ients.

Shape:

- Input: LongTensor of ar itrary shape containing the in ices to extract
- Weight: Em e ing matrix of floating point type with shape (*V, embedding_dim*),

  where V = maximum in ex + 1 an em e ing_ im = the em e ing size
- Output: (*, *embedding_dim*), where * is the input shape

Examples:

```
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.tensor([[1,2,4,5],[4,3,2,9]])
>>> # an embedding matrix containing 10 tensors of size 3
>>> embedding_matrix = torch.rand(10, 3)
>>> F.embedding(input, embedding_matrix)
tensor([[[ 0.8490,  0.9625,  0.6753],
         [ 0.9666,  0.7761,  0.6108],
         [ 0.6246,  0.9751,  0.3618],
         [ 0.4161,  0.2419,  0.7383]],

        [[ 0.6246,  0.9751,  0.3618],
         [ 0.0237,  0.7794,  0.0528],
         [ 0.9666,  0.7761,  0.6108],
         [ 0.3385,  0.8612,  0.1867]]])

>>> # example with padding_idx
>>> weights = torch.rand(10, 3)
>>> weights[0, :].zero_()
>>> embedding_matrix = weights
>>> input = torch.tensor([[0,2,0,5]])
>>> F.embedding(input, embedding_matrix, padding_idx=0)
tensor([[[ 0.0000,  0.0000,  0.0000],
         [ 0.5609,  0.5384,  0.8720],
         [ 0.0000,  0.0000,  0.0000],
         [ 0.6262,  0.2438,  0.7471]]])
```

em e ing_ ag

`torch.nn.functional.embedding_bag(input, weight, offsets=None, max_norm=None, norm_type=2, scale_grad_by_freq=False, mode='mean', sparse=False, per_sample_weights=None, include_last_offset=False)` [SO RCE]

Computes sums, means or maxes of *bags* of em e ings, without instantiating the interme iate em e ings.

See `torch.nn.EmbeddingBag` for more etails.

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** (*LongTensor*) – Tensor containing ags of in ices into the em e ing matrix
- **weight** (*Tensor*) – The em e ing matrix with num er of rows equal to the maximum possi le in ex + 1, an num er of columns equal to the em e ing size
- **offsets** (*LongTensor, optional*) – Only use when `input` is 1D. `offsets` etermines the starting in ex position of each ag (sequence) in `input`.
- **max_norm** (*float, optional*) – If given, each em e ing vector with norm larger than `max_norm` is renormalize to have norm `max_norm`. Note: this will mo ify `weight` in-place.
- **norm_ty e** (*float, optional*) – The `p` in the `p`-norm to compute for the `max_norm` option. Default `2`.
- **scale_gra _ y_fre** (*boolean, optional*) – if given, this will scale gra ients y the inverse of frequency of the wor s in the mini- atch. Default `False`. Note: this option is not supporte when `mode="max"`.
- **mo e** (*string, optional*) – `"sum"`, `"mean"` or `"max"`. Specifies the way to re uce the ag. Default: `"mean"`
- **s arse** (*bool, optional*) – if `True`, gra ient w.r.t. `weight` will e a sparse tensor. See Notes un er `torch.nn.Embedding` for more etails regar ing sparse gra ients. Note: this option is not supporte when `mode="max"`.
- **er_sam le_weights** (*Tensor, optional*) – a tensor of float / ou le weights, or None to in icate all weights shoul e taken to e 1. If specifie , `per_sample_weights` must have exactly the same shape as input an is treate as having the same `offsets`, if those are not None.
- **inclu e_last_offset** (*bool, optional*) – if `True`, the size of offsets is equal to the num er of ags + 1.
- **last element is the size of the in ut, or the en ing in ex osition of the last ag** (*The*) –

Shape:

- `input` (LongTensor) an `offsets` (LongTensor, optional)
  - If `input` is 2D of shape (B, N),

    it will e treate as `B` ags (sequences) each of fixe length `N`, an this will return `B` values aggregate in a way epen ing on the `mode`. `offsets` is ignore an require to e `None` in this case.
  - If `input` is 1D of shape (N),

    it will e treate as a concatenation of multiple ags (sequences). `offsets` is require to e a 1D tensor containing the starting in ex positions of each ag in `input`. Therefore, for `offsets` of shape (B), `input` will e viewe as having `B` ags. Empty ags (i.e., having 0-length) will have returne vectors fille y zeros.
- `weight` (Tensor): the learna le weights of the mo ule of shape $(num\_embeddings, embedding\_dim)$
- `per_sample_weights` (Tensor, optional). Has the same shape as `input`.
- `output`: aggregate em e ing values of shape $(B, embedding\_dim)$

Examples:

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding_matrix = torch.rand(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.tensor([1,2,4,5,4,3,2,9])
>>> offsets = torch.tensor([0,4])
>>> F.embedding_bag(embedding_matrix, input, offsets)
tensor([[ 0.3397,  0.3552,  0.5545],
        [ 0.5893,  0.4386,  0.5882]])
```

one_hot

`torch.nn.functional.one_hot`(*tensor*, *num_classes=-1*) → LongTensor

Takes LongTensor with in ex values of shape `(*)` an returns a tensor of shape `(*, num_classes)` that have zeros everywhere except where the in ex of last imension matches the correspon ing value of the input tensor, in which case it will e 1.

See also One-hot on Wikipe ia .

Parameters

- **tensor** (*LongTensor*) – class values of any shape.
- **num_classes** (*int*) – Total num er of classes. If set to -1, the num er of classes will e inferre as one greater than the largest class value in the input tensor.

Returns

LongTensor that has one more imension with 1 values at the in ex of last imension in icate y the input, an 0 everywhere else.

Examples

```
>>> F.one_hot(torch.arange(0, 5) % 3)
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1],
        [1, 0, 0],
        [0, 1, 0]])
>>> F.one_hot(torch.arange(0, 5) % 3, num_classes=5)
tensor([[1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0]])
>>> F.one_hot(torch.arange(0, 6).view(3,2) % 3)
tensor([[[1, 0, 0],
         [0, 1, 0]],
        [[0, 0, 1],
         [1, 0, 0]],
        [[0, 1, 0],
         [0, 0, 1]]])
```

Distance functions

airwise_ istance

`torch.nn.functional.pairwise_distance`(*x1*, *x2*, *p=2.0*, *eps=1e-06*, *keepdim=False*) [SO RCE]

See `torch.nn.PairwiseDistance` for etails

cosine_similarity

`torch.nn.functional.cosine_similarity`(*x1*, *x2*, *dim=1*, *eps=1e-8*) → Tensor

Returns cosine similarity etween x1 an x2, compute along im.

$$\text{similarity} = \frac{x_1 \cdot x_2}{\max(\Vert x_1 \Vert_2 \cdot \Vert x_2 \Vert_2, \epsilon)}$$

Parameters

- **x1** (*Tensor*) – First input.
- **x** (*Tensor*) – Secon input (of size matching x1).
- **im** (*int*, *optional*) – Dimension of vectors. Default: 1
- **e s** (*float*, *optional*) – Small value to avoi ivision y zero. Default: 1e-8

Shape:

- Input: $(\ast_1, D, \ast_2)$ where D is at position *dim*.
- Output: $(\ast_1, \ast_2)$ where 1 is at position *dim*.

Example:

```
>>> input1 = torch.randn(100, 128)
>>> input2 = torch.randn(100, 128)
>>> output = F.cosine_similarity(input1, input2)
>>> print(output)
```

ist

`torch.nn.functional.pdist(`*input, p=2*`)` → Tensor

Computes the p-norm istance etween every pair of row vectors in the input. This is i entical to the upper triangular portion, exclu ing the iagonal, of *torch.norm(input[:, None] - input, dim=2, p=p)*. This function will e 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())*.

Parameters

- **in ut** – input tensor of shape $N \times M$.
- – p value for the p-norm istance to calculate etween each vector pair $\in [0, \infty]$.

Loss functions

inary_cross_entro y

`torch.nn.functional.binary_cross_entropy(`*input, target, weight=None, size_average=None, reduce=None, reduction='mean'*`)` [SO RCE]

Function that measures the Binary Cross Entropy etween the target an the output.

See `BCELoss` for etails.

Parameters

- **in ut** – Tensor of ar itrary shape
- **target** – Tensor of the same shape as input
- **weight** (*Tensor, optional*) – a manual rescaling weight if provi e it's repeate to match input tensor shape
- **size_average** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **re uce** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the sum of the output will e ivi e y the num er of elements in the output, `'sum'`: the output will e summe . Note: `size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`. Default: `'mean'`

Examples:

```
>>> input = torch.randn((3, 2), requires_grad=True)
>>> target = torch.rand((3, 2), requires_grad=False)
>>> loss = F.binary_cross_entropy(F.sigmoid(input), target)
>>> loss.backward()
```

inary_cross_entro y_with_logits

`torch.nn.functional.binary_cross_entropy_with_logits(`*input, target, weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None*`)` [SO RCE]

Function that measures Binary Cross Entropy etween target an output logits.

See `BCEWithLogitsLoss` for etails.

Parameters

- **in ut** – Tensor of ar itrary shape
- **target** – Tensor of the same shape as input
- **weight** (*Tensor, optional*) – a manual rescaling weight if provi e it's repeate to match input tensor shape
- **size_average** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **re uce** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the sum of the output will e ivi e y the num er of elements in the output, `'sum'`: the output will e summe . Note: `size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`. Default: `'mean'`
- **os_weight** (*Tensor, optional*) – a weight of positive examples. Must e a vector with length equal to the num er of classes.

- **os_weight** (*Tensor, optional*) – a weight of positive examples. Must e a vector with length equal to the num er of classes.

Examples:

```
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> loss = F.binary_cross_entropy_with_logits(input, target)
>>> loss.backward()
```

oisson_nll_loss

`torch.nn.functional.poisson_nll_loss(`*input, target, log_input=True, full=False, size_average=None, eps=1e-08, reduce=None, reduction='mean'*`)` [SO RCE]

Poisson negative log likelihoo loss.

See `PoissonNLLLoss` for etails.

Parameters

- **in ut** – expectation of un erlying Poisson istri ution.
- **target** – ran om sample $\mathrm{target} \sim \mathrm{Poisson}(\mathrm{input})$.
- **log_in ut** – if `True` the loss is compute as $\exp(\mathrm{input}) - \mathrm{target} * \mathrm{input}$, if `False` then loss is $\mathrm{input} - \mathrm{target} * \log(\mathrm{input} + \mathrm{eps})$. Default: `True`
- **full** – whether to compute full loss, i. e. to a the Stirling approximation term. Default: `False` $\mathrm{target} * \log(\mathrm{target}) - \mathrm{target} + 0.5 * \log(2 * \pi * \mathrm{target})$.
- **size_average** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **e s** (*float, optional*) – Small value to avoi evaluation of $\log(0)$ when `log_input`=``False``. Default: 1e-8
- **re uce** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the sum of the output will e ivi e y the num er of elements in the output, `'sum'`: the output will e summe . Note: `size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`. Default: `'mean'`

cosine_em e ing_loss

`torch.nn.functional.cosine_embedding_loss(`*input1, input2, target, margin=0, size_average=None, reduce=None, reduction='mean'*`)` → Tensor [SO RCE]

See `CosineEmbeddingLoss` for etails.

cross_entro y

`torch.nn.functional.cross_entropy(`*input, target, weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean'*`)` [SO RCE]

This criterion com ines *log_softmax* an *nll_loss* in a single function.

See `CrossEntropyLoss` for etails.

Parameters

- **in ut** (*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- imensional loss.
- **target** (*Tensor*) – (N) where each value is $0 \leq \mathrm{targets}[i] \leq C - 1$, or $(N, d_1, d_2, ..., d_K)$ where $K \geq 1$ for K- imensional loss.
- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, has to e a Tensor of size *C*
- **size_average** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **ignore_in ex** (*int, optional*) – Specifies a target value that is ignore an oes not contri ute to the input gra ient. When `size_average` is `True`, the loss is average over non-ignore targets. Default: -100
- **re uce** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the sum of the output will e ivi e y the num er of elements in the output, `'sum'`: the output will e summe . Note: `size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`. Default: `'mean'`

Examples:

```
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randint(5, (3,), dtype=torch.int64)
>>> loss = F.cross_entropy(input, target)
>>> loss.backward()
```

ctc_loss

`torch.nn.functional.ctc_loss(`*log_probs, targets, input_lengths, target_lengths, blank=0, reduction='mean', zero_infinity=False*`)` [SO RCE]

The Connectionist Temporal Classification loss.

See `CTCLoss` for etails.

• NOTE

In some circumstances when using the C DA acken with CuDNN, this operator may select a non eterministic algorithm to increase performance. If this is un esira le, you can try to make the operation eterministic (potentially at a performance cost) y setting `torch.backends.cudnn.deterministic = True`. Please see the notes on Repro uci ility for ackgroun .

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **log_ ro s** – (T, N, C) where *C = number of characters in alphabet including blank*, *T = input length*, an *N = batch size*. The logarithmize pro a ilities of the outputs (e.g. o taine with `torch.nn.functional.log_softmax()`).
- **targets** – (N, S) or *(sum(target_lengths))*. Targets cannot e lank. In the secon form, the targets are assume to e concatenate .
- **in ut_lengths** – (N). Lengths of the inputs (must each e $\leq T$)
- **target_lengths** – (N). Lengths of the targets
- **lank** (*int, optional*) – Blank la el. Default 0.
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the output losses will e ivi e y the target lengths an then the mean over the atch is taken, `'sum'`: the output will e summe . Default: `'mean'`
- **zero_infinity** (*bool, optional*) – Whether to zero infinite losses an the associate gra ients. Default: `False` Infinite losses mainly occur when the inputs are too short to e aligne to the targets.

Example:

```
>>> log_probs = torch.randn(50, 16, 20).log_softmax(2).detach().requires_grad_()
>>> targets = torch.randint(1, 20, (16, 30), dtype=torch.long)
>>> input_lengths = torch.full((16,), 50, dtype=torch.long)
>>> target_lengths = torch.randint(10,30,(16,), dtype=torch.long)
>>> loss = F.ctc_loss(log_probs, targets, input_lengths, target_lengths)
>>> loss.backward()
```

hinge_em e ing_loss

`torch.nn.functional.hinge_embedding_loss`(*input, target, margin=1.0, size_average=None, reduce=None, reduction='mean'*) → Tensor [SO RCE]

See `HingeEmbeddingLoss` for etails.

kl_ iv

`torch.nn.functional.kl_div`(*input, target, size_average=None, reduce=None, reduction='mean', log_target=False*) [SO RCE]

The Kull ack-Lei ler ivergence Loss

See `KLDivLoss` for etails.

Parameters

- **in ut** – Tensor of ar itrary shape
- **target** – Tensor of the same shape as input
- **size_average** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **re uce** (*bool, optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string, optional*) – Specifies the re uction to apply to the output: `'none'` | `'batchmean'` | `'sum'` | `'mean'`. `'none'`: no re uction will e applie `'batchmean'`: the sum of the output will e ivi e y the atchsize `'sum'`: the output will e summe `'mean'`: the output will e ivi e y the num er of elements in the output Default: `'mean'`
- **log_target** (*bool*) – A flag in icating whether `target` is passe in the log space. It is recommen e to pass certain istri utions (like `softmax`) in the log space to avoi numerical issues cause y explicit `log`. Default: `False`

• NOTE

`size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`.

• NOTE

:attr:`reduction` = `'mean'` oesn't return the true kl ivergence value, please use :attr:`reduction` = `'batchmean'` which aligns with KL math efinition. In the next ma or release, `'mean'` will e change to e the same as ' atchmean'.

l1_loss

`torch.nn.functional.l1_loss`(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor [SO RCE]

Function that takes the mean element-wise a solute value ifference.

See `L1Loss` for etails.

mse_loss

`torch.nn.functional.mse_loss`(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor [SO RCE]

Measures the element-wise mean square error.

See `MSELoss` for etails.

margin_ranking_loss

`torch.nn.functional.margin_ranking_loss`(*input1*, *input2*, *target*, *margin=0*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor [SO RCE]

See `MarginRankingLoss` for etails.

multila el_margin_loss

`torch.nn.functional.multilabel_margin_loss`(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor [SO RCE]

See `MultiLabelMarginLoss` for etails.

multila el_soft_margin_loss

`torch.nn.functional.multilabel_soft_margin_loss`(*input*, *target*, *weight=None*, *size_average=None*) → Tensor [SO RCE]

See `MultiLabelSoftMarginLoss` for etails.

multi_margin_loss

`torch.nn.functional.multi_margin_loss`(*input*, *target*, *p=1*, *margin=1.0*, *weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SO RCE]

> multi_margin_loss(input, target, p=1, margin=1, weight=None, size_average=None,
>
> re uce=None, re uction='mean') -> Tensor

See `MultiMarginLoss` for etails.

nll_loss

`torch.nn.functional.nll_loss`(*input*, *target*, *weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*) [SO RCE]

The negative log likelihoo loss.

See `NLLLoss` for etails.

Parameters

- **in ut** – (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- imensional loss.
- **target** – (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- imensional loss.
- **weight** (*Tensor*, *optional*) – a manual rescaling weight given to each class. If given, has to e a Tensor of size *C*
- **size_average** (*bool*, *optional*) – Deprecate (see `reduction`). By efault, the losses are average over each loss element in the atch. Note that for some losses, there multiple elements per sample. If the fiel `size_average` is set to `False`, the losses are instea summe for each mini atch. Ignore when re uce is `False`. Default: `True`
- **ignore_in ex** (*int*, *optional*) – Specifies a target value that is ignore an oes not contri ute to the input gra ient. When `size_average` is `True`, the loss is average over non-ignore targets. Default: -100
- **re uce** (*bool*, *optional*) – Deprecate (see `reduction`). By efault, the losses are average or summe over o servations for each mini atch epen ing on `size_average`. When `reduce` is `False`, returns a loss per atch element instea an ignores `size_average`. Default: `True`
- **re uction** (*string*, *optional*) – Specifies the re uction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no re uction will e applie , `'mean'`: the sum of the output will e ivi e y the num er of elements in the output, `'sum'`: the output will e summe . Note: `size_average` an `reduce` are in the process of eing eprecate , an in the meantime, specifying either of those two args will overri e `reduction`. Default: `'mean'`

Example:

```
>>> # input is of size N x C = 3 x 5
>>> input = torch.randn(3, 5, requires_grad=True)
>>> # each element in target has to have 0 <= value < C
>>> target = torch.tensor([1, 0, 4])
>>> output = F.nll_loss(F.log_softmax(input), target)
>>> output.backward()
```

smooth_l1_loss

`torch.nn.functional.smooth_l1_loss`(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*, *beta=1.0*) [SO RCE]

Function that uses a square term if the a solute element-wise error falls elow eta an an L1 term otherwise.

See `SmoothL1Loss` for etails.

soft_margin_loss

`torch.nn.functional.soft_margin_loss`(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor [SO RCE]

See `SoftMarginLoss` for etails.

tri let_margin_loss

`torch.nn.functional.triplet_margin_loss`(*anchor*, *positive*, *negative*, *margin=1.0*, *p=2*, *eps=1e-06*, *swap=False*, *size_average=None*, *reduce=None*, *reduction='mean'*) [SO RCE]

See `TripletMarginLoss` for etails

tri let_margin_with_ istance_loss

`torch.nn.functional.triplet_margin_with_distance_loss`(*anchor*, *positive*, *negative*, ***, *distance_function=None*, *margin=1.0*, *swap=False*, *reduction='mean'*) [SO RCE]

See `TripletMarginWithDistanceLoss` for etails.

Vision functions

ixel_shuffle

`torch.nn.functional.pixel_shuffle()`

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 `PixelShuffle` for etails.

Parameters

- **in ut** (*Tensor*) – the input tensor
- **u scale_factor** (*int*) – factor to increase spatial resolution y

Examples:

```
>>> input = torch.randn(1, 9, 4, 4)
>>> output = torch.nn.functional.pixel_shuffle(input, 3)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

a

`torch.nn.functional.pad`(*input*, *pad*, *mode='constant'*, *value=0*)

Pa s tensor.

Pa ing size:

The pa ing size y which to pa some imensions of `input` are escri e starting from the last imension an moving forwar . $\left\lfloor \frac{\text{len(pad)}}{2} \right\rfloor$ imensions of `input` will e pa e . For example, to pa only the last imension of the input tensor, then `pad` has the form $(\text{padding\_left}, \text{padding\_right})$; to pa the last 2 imensions of the input tensor, then use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom})$; to pa the last 3 imensions, use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom} \; \text{padding\_front}, \text{padding\_back})$.

Pa ing mo e:

See `torch.nn.ConstantPad2d`, `torch.nn.ReflectionPad2d`, an `torch.nn.ReplicationPad2d` for concrete examples on how each of the pa ing mo es works. Constant pa ing is implemente for ar itrary imensions. Replicate pa ing is implemente for pa ing the last 3 imensions of 5D input tensor, or the last 2 imensions of 4D input tensor, or the last imension of 3D input tensor. Reflect pa ing is only implemente for pa ing the last 2 imensions of 4D input tensor, or the last imension of 3D input tensor.

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

Parameters

- **in ut** (*Tensor*) – N- imensional tensor
- **a** (*tuple*) – m-elements tuple, where $\frac{m}{2} \leq$ input imensions an m is even.
- **mo e** – `'constant'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'constant'`
- **value** – fill value for `'constant'` pa ing. Default: `0`

Examples:

```
>>> t4d = torch.empty(3, 3, 4, 2)
>>> p1d = (1, 1) # pad last dim by 1 on each side
>>> out = F.pad(t4d, p1d, "constant", 0)  # effectively zero padding
>>> print(out.size())
torch.Size([3, 3, 4, 4])
>>> p2d = (1, 1, 2, 2) # pad last dim by (1, 1) and 2nd to last by (2, 2)
>>> out = F.pad(t4d, p2d, "constant", 0)
>>> print(out.size())
torch.Size([3, 3, 8, 4])
>>> t4d = torch.empty(3, 3, 4, 2)
>>> p3d = (0, 1, 2, 1, 3, 3) # pad by (0, 1), (2, 1), and (3, 3)
>>> out = F.pad(t4d, p3d, "constant", 0)
>>> print(out.size())
torch.Size([3, 9, 7, 3])
```

inter olate

`torch.nn.functional.interpolate`(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*, *recompute_scale_factor=None*) [SO RCE]

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

The algorithm use for interpolation is etermine y `mode`.

Currently temporal, spatial an volumetric sampling are supporte , i.e. expecte inputs are 3-D, 4-D or 5-D in shape.

The input imensions are interprete in the form: *mini-batch x channels x [optional depth] x [optional height] x width*.

The mo es availa le for resizing are: *nearest*, *linear* (3D-only), *bilinear*, *bicubic* (4D-only), *trilinear* (5D-only), *area*

Parameters

- **in ut** (*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.
- **mo e** (*str*) – algorithm use for upsampling: `'nearest'` | `'linear'` | `'bilinear'` | `'bicubic'` | `'trilinear'` | `'area'`. Default: `'nearest'`
- **align_corners** (*bool, optional*) – Geometrically, we consi er the pixels of the input an output as squares rather than points. If set to `True`, the input an output tensors are aligne y the center points of their corner pixels, preserving the values at the corner pixels. If set to `False`, the input an output tensors are aligne y the corner points of their corner pixels, an the interpolation uses e ge value pa ing for out-of- oun ary 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`
- **recom ute_scale_factor** (*bool, optional*) – recompute the scale_factor for use in the interpolation calculation. When *scale_factor* is passe as a parameter, it is use to compute the *output_size*. If *recompute_scale_factor* is ``False` or not specifie , the passe -in *scale_factor* will e use in the interpolation computation. Otherwise, a new *scale_factor* will e compute ase on the output an input sizes for use in the interpolation computation (i.e. the computation will e i entical to if the compute *output_size* were passe -in explicitly). Note that when *scale_factor* is floating-point, the recompute scale_factor may iffer from the one passe in ue to roun ing an precision issues.

• NOTE

With `mode='bicubic'`, it's possi le to cause overshoot, in other wor s it can pro uce negative values or values greater than 255 for images. Explicitly call `result.clamp(min=0, max=255)` if you want to re uce the overshoot when isplaying the image.

• WARNING

With `align_corners = True`, the linearly interpolating mo es (*linear*, *bilinear*, an *trilinear*) on't proportionally align the output an input pixels, an thus the output values can epen on the input size. This was the efault ehavior for these mo es up to version 0.3.1. Since then, the efault ehavior is `align_corners = False`. See `Upsample` for concrete examples on how this affects the outputs.

• WARNING

When scale_factor is specifie , if recompute_scale_factor=True, scale_factor is use to compute the output_size which will then e use to infer new scales for the interpolation. The efault ehavior for recompute_scale_factor change to False in 1.6.0, an scale_factor is use in the interpolation calculation.

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

u sam le

`torch.nn.functional.upsample`(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*) [SO RCE]

psamples the input to either the given `size` or the given `scale_factor`

• WARNING

This function is eprecate in favor of `torch.nn.functional.interpolate()`. This is equivalent with `nn.functional.interpolate(...)`.

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

The algorithm use for upsampling is etermine y `mode`.

Currently temporal, spatial an volumetric upsampling are supporte , i.e. expecte inputs are 3-D, 4-D or 5-D in shape.

The input imensions are interprete in the form: *mini-batch x channels x* [*optional depth*] *x* [*optional height*] *x width*.

The mo es availa le for upsampling are: *nearest*, *linear* (3D-only), *bilinear*, *bicubic* (4D-only), *trilinear* (5D-only)

Parameters

- **in ut** (*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.
- **mo e** (*string*) – algorithm use for upsampling: `'nearest'` | `'linear'` | `'bilinear'` | `'bicubic'` | `'trilinear'`. Default: `'nearest'`
- **align_corners** (*bool, optional*) – Geometrically, we consi er the pixels of the input an output as squares rather than points. If set to `True`, the input an output tensors are aligne y the center points of their corner pixels, preserving the values at the corner pixels. If set to `False`, the input an output tensors are aligne y the corner points of their corner pixels, an the interpolation uses e ge value pa ing for out-of- oun ary 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`

• NOTE

With `mode='bicubic'`, it's possi le to cause overshoot, in other wor s it can pro uce negative values or values greater than 255 for images. Explicitly call `result.clamp(min=0, max=255)` if you want to re uce the overshoot when isplaying the image.

• WARNING

With `align_corners = True`, the linearly interpolating mo es (*linear*, *bilinear*, an *trilinear*) on't proportionally align the output an input pixels, an thus the output values can epen on the input size. This was the efault ehavior for these mo es up to version 0.3.1. Since then, the efault ehavior is `align_corners = False`. See `Upsample` for concrete examples on how this affects the outputs.

u sam le_nearest

`torch.nn.functional.upsample_nearest(`*input, size=None, scale_factor=None*`)` [SO RCE]

psamples the input, using nearest neigh ours' pixel values.

• WARNING

This function is eprecate in favor of `torch.nn.functional.interpolate()`. This is equivalent with `nn.functional.interpolate(..., mode='nearest')`.

Currently spatial an volumetric upsampling are supporte (i.e. expecte inputs are 4 or 5 imensional).

Parameters

- **in ut** (*Tensor*) – input
- **size** (*int or Tuple*[*int, int*] *or Tuple*[*int, int, int*]) – output spatia size.
- **scale_factor** (*int*) – multiplier for spatial size. Has to e an integer.

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

u sam le_ ilinear

`torch.nn.functional.upsample_bilinear(`*input, size=None, scale_factor=None*`)` [SO RCE]

psamples the input, using ilinear upsampling.

• WARNING

This function is eprecate in favor of `torch.nn.functional.interpolate()`. This is equivalent with `nn.functional.interpolate(..., mode='bilinear', align_corners=True)`.

Expecte inputs are spatial (4 imensional). se *upsample_trilinear* fo volumetric (5 imensional) inputs.

Parameters

- **in ut** (*Tensor*) – input
- **size** (*int or Tuple*[*int, int*]) – output spatial size.
- **scale_factor** (*int or Tuple*[*int, int*]) – multiplier for spatial size

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

gri _sam le

torch.nn.functional.grid_sample(*input*, *grid*, *mode='bilinear'*, *padding_mode='zeros'*, *align_corners=None*) [SO RCE]

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

Currently, only spatial (4-D) an volumetric (5-D) `input` are supporte .

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

For each output location `output[n, :, h, w]`, the size-2 vector `grid[n, h, w]` specifies `input` pixel locations `x` an `y`, which are use 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 metho to sample the input pixels.

`grid` specifies the sampling pixel locations normalize y the `input` spatial imensions. Therefore, it shoul have most values in the range of `[-1, 1]`. For example, values `x = -1, y = -1` is the left-top pixel of `input`, an values `x = 1, y = 1` is the right- ottom pixel of `input`.

If `grid` has values outsi e the range of `[-1, 1]`, the correspon ing outputs are han le as efine y `padding_mode`. Options are

- `padding_mode="zeros"`: use `0` for out-of- oun gri locations,
- `padding_mode="border"`: use or er values for out-of- oun gri locations,
- `padding_mode="reflection"`: use values at locations reflecte y the or er for out-of- oun gri locations. For location far away from the or er, it will keep eing reflecte until ecoming in oun , e.g., (normalize) pixel location `x = -3.5` reflects y or er `-1` an ecomes `x' = 1.5`, then reflects y or er `1` an ecomes `x'' = -0.5`.

• NOTE

This function is often use in con unction with `affine_grid()` to uil Spatial Transformer Networks .

• NOTE

When using the C DA acken , this operation may in uce non eterministic ehaviour in its ackwar pass that is not easily switche off. Please see the notes on Repro uci ility for ackgroun .

• NOTE

NaN values in `grid` woul e interprete as `-1`.

Parameters

- **in ut** (*Tensor*) – input of shape (N, C, H_{in}, W_{in}) (4-D case) or $(N, C, D_{in}, H_{in}, W_{in})$ (5-D case)
- **gri** (*Tensor*) – flow-fiel of shape $(N, H_{out}, W_{out}, 2)$ (4-D case) or $(N, D_{out}, H_{out}, W_{out}, 3)$ (5-D case)
- **mo e** (*str*) – interpolation mo e to calculate output values `'bilinear'` | `'nearest'`. Default: `'bilinear'` Note: When `mode='bilinear'` an the input is 5-D, the interpolation mo e use internally will actually e trilinear. However, when the input is 4-D, the interpolation mo e will legitimately e ilinear.
- **a ing_mo e** (*str*) – pa ing mo e for outsi e gri values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`
- **align_corners** (*bool, optional*) – Geometrically, we consi er the pixels of the input as squares rather than points. If set to `True`, the extrema (`-1` an `1`) are consi ere as referring to the center points of the input's corner pixels. If set to `False`, they are instea consi ere 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 `interpolate()`, an so whichever option is use here shoul also e use there to resize the input image efore gri sampling. Default: `False`

Returns

output Tensor

Return type

output (*Tensor*)

• WARNING

When `align_corners = True`, the gri positions epen on the pixel size relative to the input image size, an so the locations sample y `grid_sample()` will iffer for the same input given at ifferent resolutions (that is, after eing upsample or ownsample). The efault ehavior up to version 1.2.0 was `align_corners = True`. Since then, the efault ehavior has een change to `align_corners = False`, in or er to ring it in line with the efault for `interpolate()`.

affine_gri

torch.nn.functional.affine_grid(*theta*, *size*, *align_corners=None*) [SO RCE]

Generates a 2D or 3D flow fiel (sampling gri), given a atch of affine matrices `theta`.

• NOTE

This function is often use in con unction with `grid_sample()` to uil Spatial Transformer Networks .

Parameters

- **theta** (*Tensor*) – input atch 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`, consi er `-1` an `1` to refer to the centers of the corner pixels rather than the image corners. Refer to `grid_sample()` for a more complete escription. A gri generate y `affine_grid()` shoul e passe to `grid_sample()` with the same setting for this option. Default: `False`

Returns

output Tensor of size ($N \times H \times W \times 2$)

Return type

output (Tensor)

• WARNING

When `align_corners = True`, the gri positions epen on the pixel size relative to the input image size, an so the locations sample y `grid_sample()` will iffer for the same input given at ifferent resolutions (that is, after eing upsample or ownsample). The efault ehavior up to version 1.2.0 was `align_corners = True`. Since then, the efault ehavior has een change to `align_corners = False`, in or er to ring it in line with the efault for `interpolate()`.

• WARNING

When `align_corners = True`, 2D affine transforms on 1D ata an 3D affine transforms on 2D ata (that is, when one of the spatial imensions has unit size) are ill- efine , an not an inten e use case. This is not a pro lem when `align_corners = False`. p to version 1.2.0, all gri points along a unit imension were consi ere ar itrarily to e at `-1`. From version 1.3.0, un er `align_corners = True` all gri points along a unit imension are consi ere to e at ``0` (the center of the input image).

DataParallel functions (multi-GP , istri ute)

ata_ arallel

`torch.nn.parallel.data_parallel`(*module, inputs, device_ids=None, output_device=None, dim=0, module_kwargs=None*) [SO RCE]

Evaluates mo ule(input) in parallel across the GP s given in evice_i s.

This is the functional version of the DataParallel mo ule.

Parameters

- **mo ule** (*Module*) – the mo ule to evaluate in parallel
- **in uts** (*Tensor*) – inputs to the mo ule
- **evice_i s** (*list of python:int or torch.device*) – GP i s on which to replicate mo ule
- **out ut_ evice** (*list of python:int or torch.device*) – GP location of the output se -1 to in icate the CP . (efault: evice_i s[0])

Returns

a Tensor containing the result of mo ule(input) locate on output_ evice