

## EE-559 – Deep learning

### 9.1. Transposed convolutions

François Fleuret

<https://fleuret.org/ee559/>

Mon Nov 19 08:07:48 UTC 2018

Constructing deep generative architectures requires layers to increase the signal dimension, the contrary of what we have done so far with feed-forward networks.

Constructing deep generative architectures requires layers to increase the signal dimension, the contrary of what we have done so far with feed-forward networks.

Generative processes that consist of optimizing the input rely on back-propagation to expend the signal from a low-dimension representation to the high-dimension signal space.

Constructing deep generative architectures requires layers to increase the signal dimension, the contrary of what we have done so far with feed-forward networks.

Generative processes that consist of optimizing the input rely on back-propagation to expend the signal from a low-dimension representation to the high-dimension signal space.

The same can be done in the forward pass with **transposed convolution layers** whose forward operation corresponds to a convolution layer's backward pass.

Consider a 1d convolution with a kernel  $\kappa$

$$\begin{aligned}y_i &= (x \circledast \kappa)_i \\&= \sum_a x_{i+a-1} \kappa_a \\&= \sum_u x_u \kappa_{u-i+1}.\end{aligned}$$

Consider a 1d convolution with a kernel  $\kappa$

$$\begin{aligned} y_i &= (x \circledast \kappa)_i \\ &= \sum_a x_{i+a-1} \kappa_a \\ &= \sum_u x_u \kappa_{u-i+1}. \end{aligned}$$

We get

$$\begin{aligned} \left[ \frac{\partial \ell}{\partial x} \right]_u &= \frac{\partial \ell}{\partial x_u} \\ &= \sum_i \frac{\partial \ell}{\partial y_i} \frac{\partial y_i}{\partial x_u} \\ &= \sum_i \frac{\partial \ell}{\partial y_i} \kappa_{u-i+1}. \end{aligned}$$

which looks a lot like a standard convolution layer, except that the kernel coefficients are visited in reverse order.

This is actually the standard convolution operator from signal processing. If  $*$  denotes this operation, we have

$$(\chi * \kappa)_i = \sum_a \chi_a \kappa_{i-a+1}.$$

This is actually the standard convolution operator from signal processing. If  $*$  denotes this operation, we have

$$(x * \kappa)_i = \sum_a x_a \kappa_{i-a+1}.$$

Coming back to the backward pass of the convolution layer, if

$$y = x \circledast \kappa$$

then

$$\left[ \frac{\partial \ell}{\partial x} \right] = \left[ \frac{\partial \ell}{\partial y} \right] * \kappa.$$



In the deep-learning field, since it corresponds to transposing the weight matrix of the equivalent fully-connected layer, it is called a **transposed convolution**.

$$\begin{pmatrix} \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\ 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\ 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\ 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\ 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \end{pmatrix}^T = \begin{pmatrix} \kappa_1 & 0 & 0 & 0 & 0 \\ \kappa_2 & \kappa_1 & 0 & 0 & 0 \\ \kappa_3 & \kappa_2 & \kappa_1 & 0 & 0 \\ 0 & \kappa_3 & \kappa_2 & \kappa_1 & 0 \\ 0 & 0 & \kappa_3 & \kappa_2 & \kappa_1 \\ 0 & 0 & 0 & \kappa_3 & \kappa_2 \\ 0 & 0 & 0 & 0 & \kappa_3 \end{pmatrix}$$

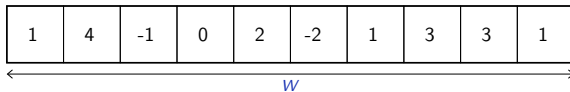
In the deep-learning field, since it corresponds to transposing the weight matrix of the equivalent fully-connected layer, it is called a **transposed convolution**.

$$\begin{pmatrix} \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\ 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\ 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\ 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\ 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \end{pmatrix}^T = \begin{pmatrix} \kappa_1 & 0 & 0 & 0 & 0 \\ \kappa_2 & \kappa_1 & 0 & 0 & 0 \\ \kappa_3 & \kappa_2 & \kappa_1 & 0 & 0 \\ 0 & \kappa_3 & \kappa_2 & \kappa_1 & 0 \\ 0 & 0 & \kappa_3 & \kappa_2 & \kappa_1 \\ 0 & 0 & 0 & \kappa_3 & \kappa_2 \\ 0 & 0 & 0 & 0 & \kappa_3 \end{pmatrix}$$

While a convolution can be seen as a series of inner products, a transposed convolution can be seen as a weighted sum of translated kernels.

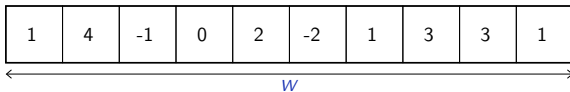
## Convolution layer

Input

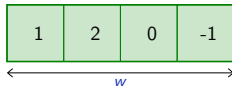


## Convolution layer

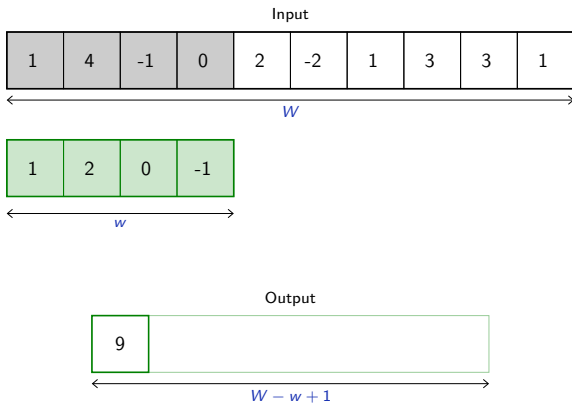
Input



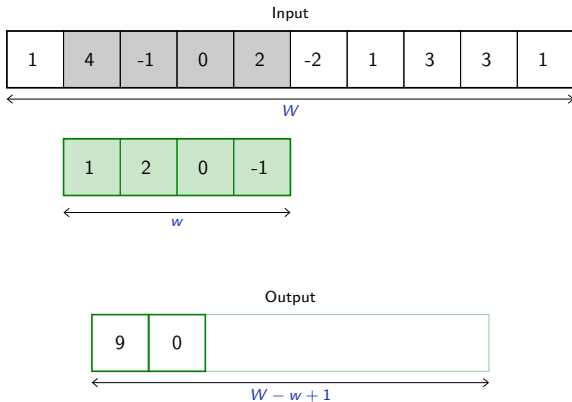
Kernel



## Convolution layer

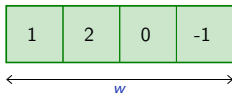
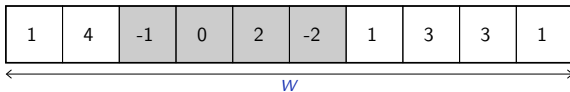


## Convolution layer

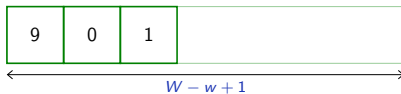


## Convolution layer

Input

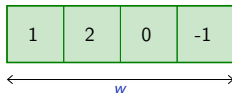
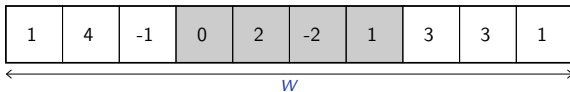


Output

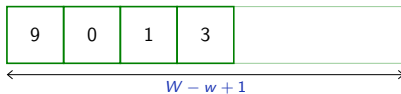


## Convolution layer

Input



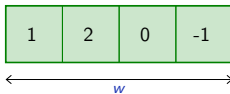
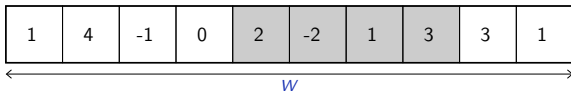
Output



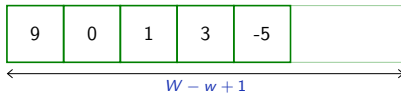


## Convolution layer

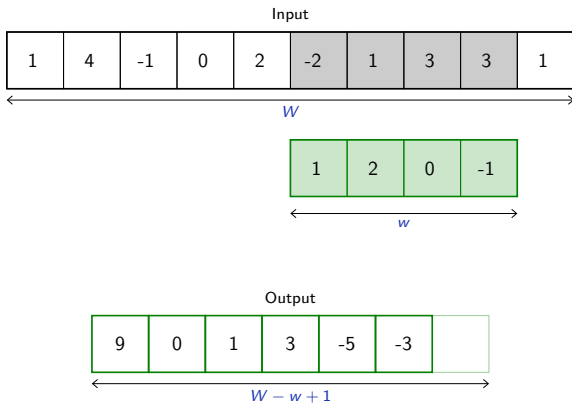
### Input



### Output

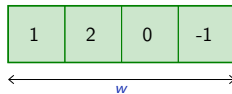
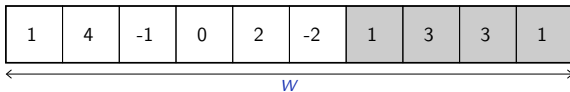


## Convolution layer

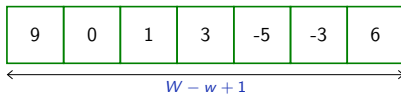


## Convolution layer

Input

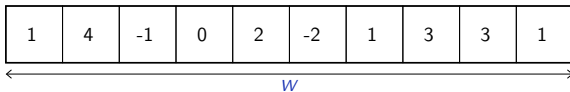


Output

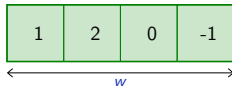


## Convolution layer

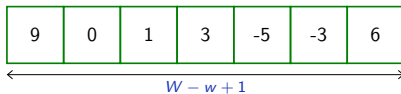
Input



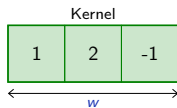
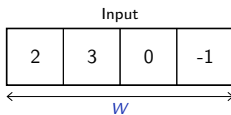
Kernel



Output

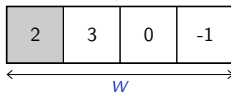


## Transposed convolution layer

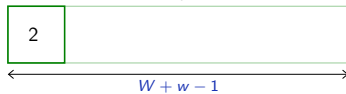


## Transposed convolution layer

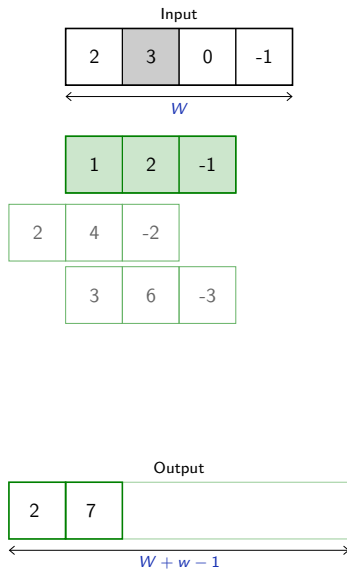
Input



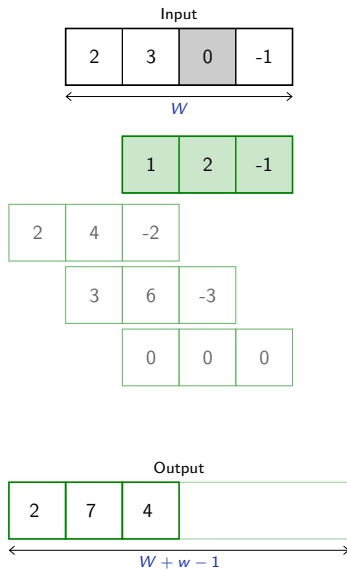
### Output



## Transposed convolution layer

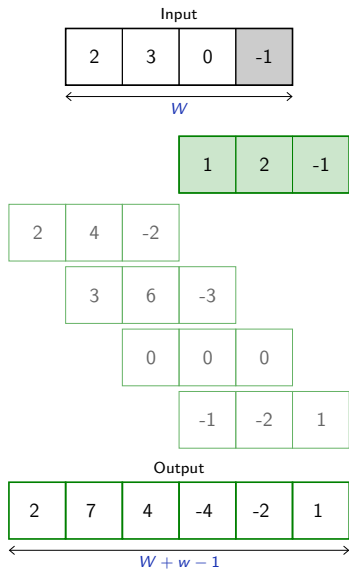


## Transposed convolution layer

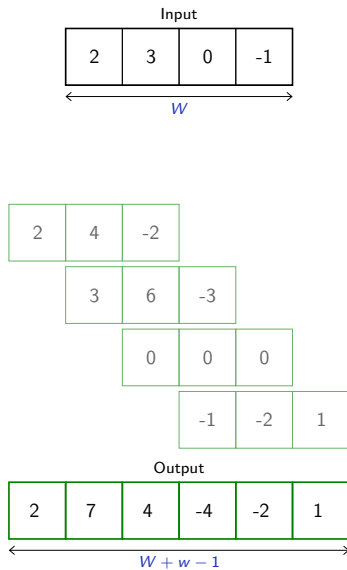




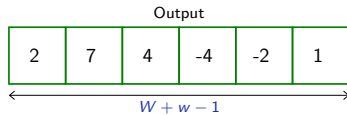
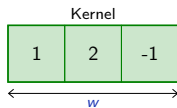
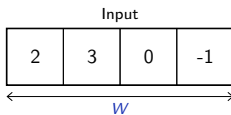
## Transposed convolution layer



## Transposed convolution layer



## Transposed convolution layer



`torch.nn.functional.conv_transpose1d` implements the operation we just described. It takes as input a batch of multi-channel samples, and produces a batch of multi-channel samples.

```
>>> x = torch.tensor([[[[0., 0., 1., 0., 0., 0., 0.]]]])
>>> k = torch.tensor([[[[1., 2., 3.]]]])
>>> F.conv1d(x, k)
tensor([[[[ 3.,  2.,  1.,  0.,  0.]]]])
```

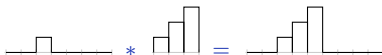


`torch.nn.functional.conv_transpose1d` implements the operation we just described. It takes as input a batch of multi-channel samples, and produces a batch of multi-channel samples.

```
>>> x = torch.tensor([[[[0., 0., 1., 0., 0., 0., 0.]]]])
>>> k = torch.tensor([[[[1., 2., 3.]]]])
>>> F.conv1d(x, k)
tensor([[[[ 3.,  2.,  1.,  0.,  0.]]]])
```



```
>>> F.conv_transpose1d(x, k)
tensor([[[[ 0.,  0.,  1.,  2.,  3.,  0.,  0.,  0.,  0.]]]])
```



The class `torch.nn.ConvTranspose1d` embeds that operation into a `torch.nn.Module`.

```
>>> x = torch.tensor([[[ 2., 3., 0., -1.]]])
>>> m = nn.ConvTranspose1d(1, 1, kernel_size=3)
>>> m.bias.data.zero_()
tensor([0.])
>>> m.weight.data.copy_(torch.tensor([ 1, 2, -1 ]))
tensor([[[ 1., 2., -1.]]])
>>> y = m(x)
>>> y
tensor([[[ 2., 7., 4., -4., -2., 1.] ]], grad_fn=<SqueezeBackward1>)
```

Transposed convolutions also have a `dilation` parameter that behaves as for convolution and expands the kernel size without increasing the number of parameters by making it sparse.

Transposed convolutions also have a `dilation` parameter that behaves as for convolution and expands the kernel size without increasing the number of parameters by making it sparse.

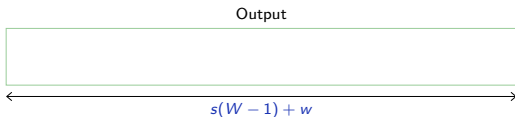
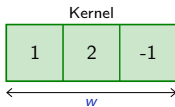
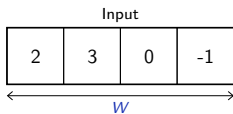
They also have a `stride` and `padding` parameters, however, due to the relation between convolutions and transposed convolutions:



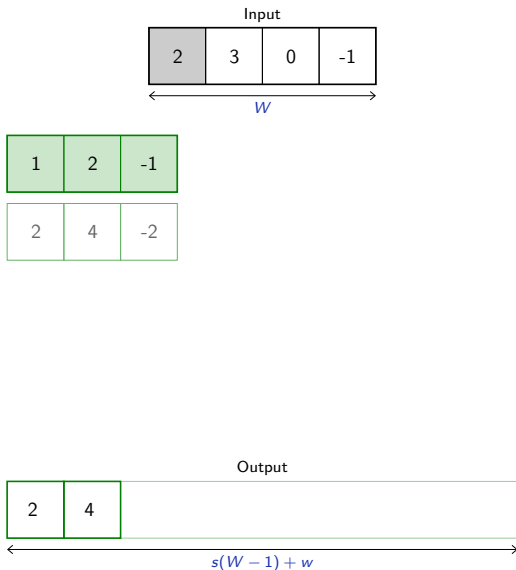
While for convolutions `stride` and `padding` are defined in the input map, for transposed convolutions these parameters are defined in the output map, and the latter modulates a cropping operation.



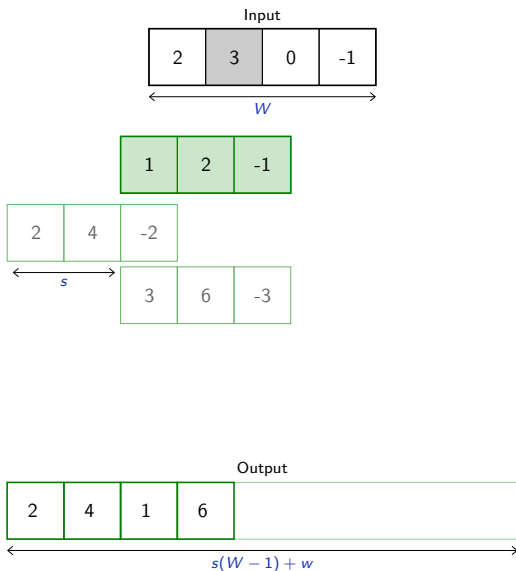
## Transposed convolution layer (stride = 2)



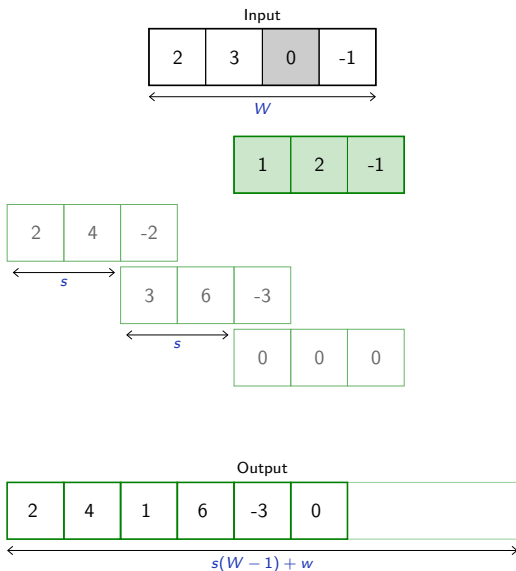
Transposed convolution layer (stride = 2)



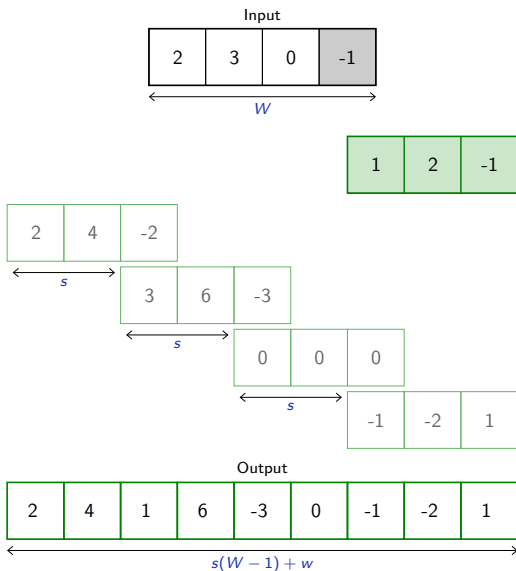
## Transposed convolution layer (stride = 2)



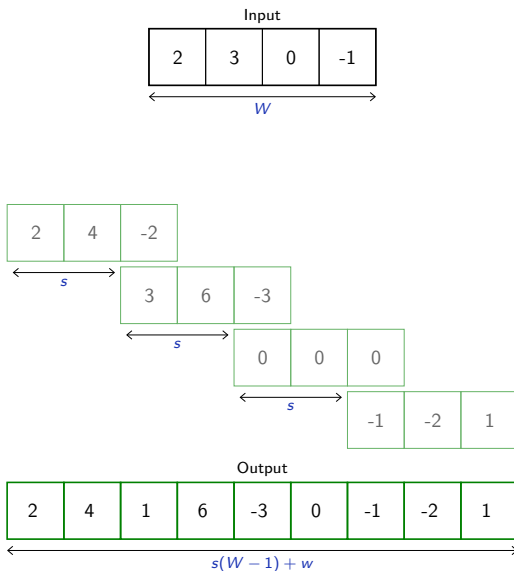
## Transposed convolution layer (stride = 2)



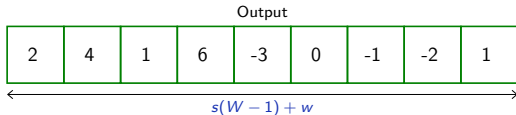
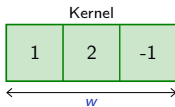
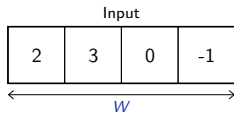
## Transposed convolution layer (stride = 2)



## Transposed convolution layer (stride = 2)



## Transposed convolution layer (stride = 2)



The composition of a convolution and a transposed convolution of same parameters keep the signal size [roughly] unchanged.



A convolution with a stride greater than one may ignore parts of the signal. Its composition with the corresponding transposed convolution generates a map **of the size of the observed area**.



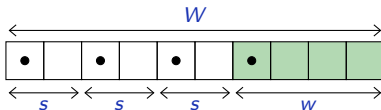
The composition of a convolution and a transposed convolution of same parameters keep the signal size [roughly] unchanged.



A convolution with a stride greater than one may ignore parts of the signal. Its composition with the corresponding transposed convolution generates a map **of the size of the observed area**.

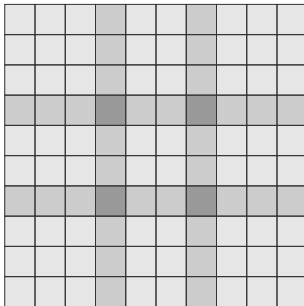
For instance, a 1d convolution of kernel size  $w$  and stride  $s$  composed with the transposed convolution of same parameters maintains the signal size  $W$ , only if

$$\exists q \in \mathbb{N}, W = w + s q.$$



It has been observed that transposed convolutions may create some grid-structure artifacts, since generated pixels are not all covered similarly.

For instance with a  $4 \times 4$  kernel and stride 3



An alternative is to use an analytic up-scaling, implemented in the PyTorch modules `nn.Upsample`.

```
>>> x = torch.tensor([[[[ 1., 2. ], [ 3., 4. ]]]])
>>> b = nn.Upsample(scale_factor = 3, mode = 'bilinear')
>>> b(x)
tensor([[[[ 1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
           [ 1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
           [ 1.6667, 1.6667, 2.0000, 2.3333, 2.6667, 2.6667],
           [ 2.3333, 2.3333, 2.6667, 3.0000, 3.3333, 3.3333],
           [ 3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000],
           [ 3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000]]]])
```

An alternative is to use an analytic up-scaling, implemented in the PyTorch modules `nn.Upsample`.

```
>>> x = torch.tensor([[[[ 1., 2. ], [ 3., 4. ]]]])
>>> b = nn.Upsample(scale_factor = 3, mode = 'bilinear')
>>> b(x)
tensor([[[[ 1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
           [ 1.0000, 1.0000, 1.3333, 1.6667, 2.0000, 2.0000],
           [ 1.6667, 1.6667, 2.0000, 2.3333, 2.6667, 2.6667],
           [ 2.3333, 2.3333, 2.6667, 3.0000, 3.3333, 3.3333],
           [ 3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000],
           [ 3.0000, 3.0000, 3.3333, 3.6667, 4.0000, 4.0000]]]])

>>> u = nn.Upsample(scale_factor = 3, mode = 'nearest')
>>> u(x)
tensor([[[[ 1., 1., 1., 2., 2., 2.],
           [ 1., 1., 1., 2., 2., 2.],
           [ 1., 1., 1., 2., 2., 2.],
           [ 3., 3., 3., 4., 4., 4.],
           [ 3., 3., 3., 4., 4., 4.],
           [ 3., 3., 3., 4., 4., 4.]]]]])
```

Such module is usually combined with a convolution to learn local corrections to undesirable artifacts of the up-scaling.

In practice, a transposed convolution such as

```
nn.ConvTranspose2d(nic, noc,  
                   kernel_size = 3, stride = 2,  
                   padding = 1, output_padding = 1),
```

can be replaced by

```
nn.Upsample(scale_factor = 2, mode = 'bilinear')  
nn.Conv2d(nic, noc, kernel_size = 3, padding = 1)
```

The end