Deep learning

7.1. Transposed convolutions

François Fleuret

https://fleuret.org/dlc/

Dec 20, 2020
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.

Some generative processes optimize the input, and as such 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.

Some generative processes optimize the input, and as such 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 $\kappa$

$$y_i = (x \ast \kappa)_i$$

$$= \sum_a x_{i+a-1} \kappa_a$$

$$= \sum_u x_u \kappa_{u-i+1}.$$
Consider a 1d convolution with a kernel $\kappa$

\[
y_i = (x \ast \kappa)_i = \sum_a x_{i+a-1} \kappa_a = \sum_u x_u \kappa_{u-i+1}.
\]

We get

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

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 $\ast$ denotes this operation, we have

$$(x \ast \kappa)_i = \sum_a x_a \kappa_{i-a+1}.$$
This is actually the standard convolution operator from signal processing. If $\ast$ denotes this operation, we have

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

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

$$y = x \otimes \kappa$$

then

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

\[
\begin{pmatrix}
\kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\
0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\
0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\
0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\
0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \\
0 & 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3
\end{pmatrix}
\]

\[=\]

\[
\begin{pmatrix}
\kappa_1 & 0 & 0 & 0 & 0 & 0 \\
\kappa_2 & \kappa_1 & 0 & 0 & 0 \\
\kappa_3 & \kappa_2 & \kappa_1 & 0 & 0 \\
0 & \kappa_3 & \kappa_2 & \kappa_1 & 0 \\
0 & 0 & \kappa_3 & \kappa_2 & \kappa_1 \\
0 & 0 & 0 & \kappa_3 & \kappa_2 \\
0 & 0 & 0 & 0 & \kappa_3
\end{pmatrix}
\]

\[\top\]
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**.

\[
\begin{bmatrix}
\kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 & 0 \\
0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 & 0 \\
0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 & 0 \\
0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 & 0 \\
0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \\
0 & 0 & 0 & 0 & 0 & \kappa_1 & \kappa_2 & \kappa_3 \\
\end{bmatrix}
\end{array}
\begin{bmatrix}
\kappa_1 & 0 & 0 & 0 & 0 \\
\kappa_2 & \kappa_1 & 0 & 0 & 0 \\
\kappa_3 & \kappa_2 & \kappa_1 & 0 & 0 \\
0 & \kappa_3 & \kappa_2 & \kappa_1 & 0 \\
0 & 0 & \kappa_3 & \kappa_2 & \kappa_1 \\
0 & 0 & 0 & \kappa_3 & \kappa_2 \\
0 & 0 & 0 & 0 & \kappa_3 \\
\end{bmatrix}
\]

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

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{array}
\]

Kernel

\[
W
\]
Convolution layer

Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{array}
\]

Kernel

\[
\begin{array}{cccc}
1 & 2 & 0 & -1
\end{array}
\]
Convolution layer

Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 \\
\end{array}
\]

\[W \rightarrow \]

\[
\begin{array}{cccc}
1 & 2 & 0 & -1 \\
\end{array}
\]

\[\leftarrow w \]

Output

\[\begin{array}{l}
9 \\
\end{array}\]

\[W - w + 1 \rightarrow \]
Convolution layer

Input

| 1 | 4 | -1 | 0 | 2 | -2 | 1 | 3 | 3 | 1 |

\[ W - w + 1 \]

Kernel

\[ w \]

Output

| 9 | 0 |

\[ W - w + 1 \]
Convolution layer

Input

\[
\begin{array}{ccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 \ \\
\end{array}
\]

\[W\]

\[
\begin{array}{cccc}
1 & 2 & 0 & -1 \\
\end{array}
\]

\[w\]

Output

\[
\begin{array}{ccc}
9 & 0 & 1 \\
\end{array}
\]

\[W - w + 1\]
Convolution layer

Input

\[
\begin{array}{ccccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{array}
\]

\[W \]

\[\text{Kernel} \]

\[
\begin{array}{cccc}
1 & 2 & 0 & -1
\end{array}
\]

\[w \]

Output

\[
\begin{array}{cccc}
9 & 0 & 1 & 3
\end{array}
\]

\[W - w + 1 \]
Convolution layer

Input

\[ \begin{bmatrix} 1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1 \end{bmatrix} \]

\[ W \]

\[ w \]

Output

\[ \begin{bmatrix} 9 & 0 & 1 & 3 & -5 \end{bmatrix} \]

\[ W - w + 1 \]
Convolution layer

Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1 \\
\end{array}
\]

\[
W
\]

Output

\[
\begin{array}{cccc}
9 & 0 & 1 & 3 & -5 & -3 \\
\end{array}
\]

\[
W - w + 1
\]
Convolution layer

Input

\[
\begin{array}{cccccc}
1 & 4 & -1 & 0 & 2 & -2 \\
\end{array}
\]

\[W\]

\[w\]

\[
\begin{array}{cccc}
1 & 3 & 3 & 1 \\
\end{array}
\]

Output

\[
\begin{array}{ccccccc}
9 & 0 & 1 & 3 & -5 & -3 & 6 \\
\end{array}
\]

\[W - w + 1\]
Convolution layer

Input

\[
\begin{array}{cccccccc}
1 & 4 & -1 & 0 & 2 & -2 & 1 & 3 & 3 & 1
\end{array}
\]

\[W\]

Kernel

\[
\begin{array}{cccc}
1 & 2 & 0 & -1
\end{array}
\]

\[w\]

Output

\[
\begin{array}{cccccccc}
9 & 0 & 1 & 3 & -5 & -3 & 6
\end{array}
\]

\[W - w + 1\]
Transposed convolution layer

Input

\[
\begin{array}{c}
2 \\
3 \\
0 \\
-1
\end{array}
\]

Kernel

\[
\begin{array}{c}
1 \\
2 \\
-1
\end{array}
\]
Transposed convolution layer

\[
\begin{array}{c}
\text{Input} \\
\begin{array}{cccc}
2 & 3 & 0 & -1 \\
\end{array} \\
\end{array}
\begin{array}{c}
\text{Kernel} \\
\begin{array}{ccc}
1 & 2 & -1 \\
2 & 4 & -2 \\
\end{array} \\
\end{array}
\begin{array}{c}
\text{Output} \\
\begin{array}{c}
2 \\
\end{array} \\
\end{array}
\begin{array}{c}
W + w - 1 \\
\end{array}
\end{array}
\]
Transposed convolution layer

Input

\[
\begin{bmatrix}
2 & 3 & 0 & -1
\end{bmatrix}
\]

\[W\]

Kernel

\[
\begin{bmatrix}
1 & 2 & -1 \\
2 & 4 & -2 \\
3 & 6 & -3
\end{bmatrix}
\]

Output

\[
\begin{bmatrix}
2 & 7
\end{bmatrix}
\]

\[W + w - 1\]
Transposed convolution layer

Input

\[
\begin{bmatrix}
2 & 3 & 0 & -1
\end{bmatrix}
\]

Kernel

\[
\begin{bmatrix}
1 & 2 & -1
\end{bmatrix}
\]

\[
\begin{bmatrix}
2 & 4 & -2 \\
3 & 6 & -3 \\
0 & 0 & 0
\end{bmatrix}
\]

Output

\[
\begin{bmatrix}
2 & 7 & 4
\end{bmatrix}
\]

\[
W + w - 1
\]
Transposed convolution layer

Input

\[
\begin{bmatrix}
2 & 3 & 0 & -1
\end{bmatrix}
\]

\[W \quad w - 1\]

Kernel

\[
\begin{bmatrix}
1 & 2 & -1
\end{bmatrix}
\]

\[
\begin{bmatrix}
2 & 4 & -2 \\
3 & 6 & -3 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
-1 & -2 & 1
\end{bmatrix}
\]

Output

\[
\begin{bmatrix}
2 & 7 & 4 & -4 & -2 & 1
\end{bmatrix}
\]

\[W + w - 1\]
Transposed convolution layer

Input

\[
\begin{array}{cccc}
2 & 3 & 0 & -1 \\
\end{array}
\]

\[W\]

Output

\[
\begin{array}{cccccc}
2 & 7 & 4 & -4 & -2 & 1 \\
\end{array}
\]

\[W + w - 1\]
Transposed convolution layer

Input

```
2 3 0 -1
```

Kernel

```
1 2 -1
```

Output

```
2 7 4 -4 -2 1
```

\[ W + w - 1 \]
\texttt{F.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.

```python
>>> 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.]])
```

Francois Fleuret, Deep learning / 7.1. Transposed convolutions
\texttt{F.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.

```python
>>> 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.]])
```

\[
\ast
\]

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

\[
\ast
\]

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

\[
\ast
\]
The class `nn.ConvTranspose1d` embeds that operation into a `nn.Module`.

```python
>>> x = torch.tensor([[ 1., 0., 0., 0., -1.]]
>>> m = nn.ConvTranspose1d(1, 1, kernel_size=3)
>>> with torch.autograd.no_grad():
...    m.bias.zero_()
...    m.weight.copy_(torch.tensor([ 1, 2, 1 ]))
...  m.bias
Parameter containing:
tensor([0.], requires_grad=True)
>>> y = m(x)
>>> y
tensor([[ 1., 2., 1., 0., -1., -2., -1.]])
```
Transposed convolutions also have a *dilation* parameter that behaves as for convolution and expends 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 expends 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)

Input

\[
\begin{pmatrix}
2 & 3 & 0 & -1 \\
\end{pmatrix}
\]

Kernel

\[
\begin{pmatrix}
1 & 2 & -1 \\
\end{pmatrix}
\]

Output

\[
\begin{pmatrix}
s(W - 1) + w \\
\end{pmatrix}
\]
Transposed convolution layer (stride = 2)

Input

\[
\begin{array}{cccc}
2 & 3 & 0 & -1
\end{array}
\]

Kernel

\[
\begin{array}{ccc}
1 & 2 & -1 \\
2 & 4 & -2
\end{array}
\]

Output

\[
\begin{array}{cc}
2 & 4
\end{array}
\]

\[s(W - 1) + w\]
Transposed convolution layer (stride = 2)

\[ s(W - 1) + w \]

Input

\[
\begin{array}{cccc}
2 & 3 & 0 & -1 \\
\end{array}
\]

\[ W \]

\[
\begin{array}{ccc}
1 & 2 & -1 \\
\end{array}
\]

\[ s \]

\[
\begin{array}{ccc}
2 & 4 & -2 \\
3 & 6 & -3 \\
\end{array}
\]

\[ s \]

Output

\[
\begin{array}{cccc}
2 & 4 & 1 & 6 \\
\end{array}
\]

\[ s(W - 1) + w \]
Transposed convolution layer (stride = 2)

Input

\[
\begin{bmatrix}
2 & 3 & 0 & -1
\end{bmatrix}
\]

\(W\)

Kernel

\[
\begin{bmatrix}
1 & 2 & -1
\end{bmatrix}
\]

\(s\)

Output

\[
\begin{bmatrix}
2 & 4 & 1 & 6 & -3 & 0
\end{bmatrix}
\]

\(s(W - 1) + w\)
Transposed convolution layer (stride = 2)

\[
\text{Input} = \begin{bmatrix}
2 & 3 & 0 & -1 \\
\end{bmatrix}
\]

\[
\text{Kernel} = \begin{bmatrix}
1 & 2 & -1 \\
\end{bmatrix}
\]

\[
\text{Output} = \begin{bmatrix}
2 & 4 & 1 & 6 & -3 & 0 & -1 & -2 & 1 \\
\end{bmatrix}
\]

\[
\text{Formula: } s(W - 1) + w
\]
Transposed convolution layer (stride = 2)

Input

\[
\begin{align*}
2 & \quad 3 & \quad 0 & \quad -1 \\
\end{align*}
\]

Output

\[
\begin{align*}
2 & \quad 4 & \quad 1 & \quad 6 & \quad -3 & \quad 0 & \quad -1 & \quad -2 & \quad 1 \\
\end{align*}
\]

Kernel

\[
\begin{align*}
2 & \quad 3 & \quad 0 & \quad -1 \\
1 & \quad 2 & \quad 3 & \quad 0 & \quad -1 & \quad 2 & \quad 3 & \quad 0 & \quad -1 \\
\end{align*}
\]

\[
\begin{align*}
\text{Output} = s(W - 1) + w
\end{align*}
\]
Transposed convolution layer (stride = 2)

Input

\[
\begin{array}{cccc}
2 & 3 & 0 & -1 \\
\end{array}
\]

Kernel

\[
\begin{array}{ccc}
1 & 2 & -1 \\
\end{array}
\]

Output

\[
\begin{array}{cccccccc}
2 & 4 & 1 & 6 & -3 & 0 & -1 & -2 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
W & & & & & \\
\end{array}
\]

\[
\begin{array}{cccccc}
w & & & & & \\
\end{array}
\]

\[
\begin{array}{cccccc}
s(W - 1) + w & & & & & \\
\end{array}
\]
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

$$\exists q \in \mathbb{N}, \ W = w + s \cdot q.$$
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 functional \texttt{F.interpolate}.

```python
>>> x = torch.tensor([[[ 1., 2. ], [ 3., 4. ]]])

>>> F.interpolate(x, scale_factor = 3, mode = 'bilinear')
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 functional `F.interpolate`.

```python
>>> x = torch.tensor([[[ 1., 2. ], [ 3., 4. ]]]

>>> F.interpolate(x, scale_factor = 3, mode = 'bilinear')
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]]])

>>> F.interpolate(x, scale_factor = 3, mode = 'nearest')
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

```python
import torch.nn as nn

# Define the transposed convolution
k = 3
s = 2
p = 1
o_p = 1

tconv = nn.ConvTranspose2d(nic, noc, kernel_size = k, stride = s, padding = p, output_padding = o_p),

# Forward pass
y = tconv(x)
```

Alternatively, this can be achieved using a regular convolution followed by an upsampling:

```python
import torch.nn.functional as F

# Define the convolution and upsampling
conv = nn.Conv2d(nic, noc, kernel_size = k, padding = p)
u = F.interpolate(x, scale_factor = s, mode = 'bilinear')

# Forward pass
y = conv(u)
```
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

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

y = tconv(x)
```

can be replaced by

```python
conv = nn.Conv2d(nic, noc, kernel_size = 3, padding = 1)

u = F.interpolate(x, scale_factor = 2, mode = 'bilinear')

y = conv(u)
```
The end