Deep learning

7.1. Transposed convolutions

François Fleuret
https://fleuret.org/dlc/
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 (e.g. lecture 9.4. “Optimizing inputs”) The same can be done in the forward pass with transposed convolution layers whose forward operation corresponds to a convolution layer’s backward pass.

Notes

The convolution layers that we have seen until now usually reduce the size of the signal:

- either because the filter size (with no additional padding) reduces the tensor on the outside, or
- because the stride is greater than 1.

Hence they are useful to go from a high dimensional signal (e.g. image, sound sample) to a smaller one (e.g. vector of class scores).

The transposed convolution layers provides a way of increasing the size of the signal, which is necessary for generative tasks.
Consider a 1d convolution with a kernel $\kappa$:

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

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.

Notes

Since $x$ influences $\ell$ only through $y$, we have

$$\frac{\partial \ell}{\partial x_u} = \sum_i \frac{\partial \ell}{\partial y_i} \frac{\partial y_i}{\partial x_u}.$$  

We see that

$$\sum_u x_u \kappa_{u-i+1}$$

is very similar to

$$\sum_i \frac{\partial \ell}{\partial y_i} \kappa_{u-i+1},$$

except that

- in the first case, the filter $\kappa$ and the signal $x$ are visited in the same order, as indexed by $u$, and
- in the second case, the derivative $\frac{\partial \ell}{\partial y_i}$ and the filter $\kappa$ are visited in opposite directions, as indexes by $i$. The filter is “flipped” in this case.
This is actually the standard convolution operator from signal processing. If \( * \) 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

\[
y = x \circledast \kappa
\]

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**.

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

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

This convolution can be re-written as the following matrix product

\[
\begin{bmatrix}
1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 2 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
1 \\
4 \\
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
= 
\begin{bmatrix}
9 \\
0 \\
1 \\
3 \\
-5 \\
-3 \\
6
\end{bmatrix}
\]
Notes

This transposed convolution can be formulated as a matrix multiplication as follows:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 \\
-1 & 2 & 1 & 0 \\
0 & -1 & 2 & 1 \\
0 & 0 & -1 & 2 \\
0 & 0 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
2 \\
7 \\
4 \\
-4 \\
-2 \\
1
\end{bmatrix}
= 
\begin{bmatrix}
27 \\
7 \\
4 \\
-4 \\
-2 \\
1
\end{bmatrix}
\]

from which we can interpret as a weighted sum of kernels.

And we also notice that the output dimension is larger than the input one.
\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.

We can compare on a simple 1d example the results of a standard and a transposed convolution:

```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.]])
```

\[
\begin{array}{cccccc}
\hline
\text{Input} & \text{Filter} & \quad & \text{Output} \\
\hline
1 & 2 & 3 & \leftrightarrow & 3 & 2 & 1 & 0 & 0 \\
\end{array}
\]

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

\[
\begin{array}{cccccc}
\hline
\text{Input} & \text{Filter} & \quad & \text{Output} \\
\hline
1 & 2 & 3 & \leftrightarrow & 0 & 0 & 1 & 2 & 3 & 0 & 0 & 0 & 0 \\
\end{array}
\]

**Notes**

The transposed convolution increases the signal size and does not flip the filter shape.

So a standard convolution computes at every location of a tensor the responses of linear filters, and a transposed convolution computes at every location a linear combination of kernels.
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]))
...
Parameter containing:
tensor([0.], requires_grad=True)
Parameter containing:
tensor([[1., 2., 1.]], requires_grad=True)
>>> y = m(x)
>>> y
tensor([[1., 2., 1., 0., -1., -2., -1.]], grad_fn=<SqueezeBackward1>)
```
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

\[
\begin{array}{ccc}
2 & 3 & 0 \\
\end{array}
\]

\[W\]

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

\[s\]

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

\[s\]

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

\[s\]

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

Output

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

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

---

**Notes**

The level of gray of each square is proportional to the number of filters that cover that location. Darker is more visited.
An alternative is to use an analytic up-scaling, implemented in the PyTorch functional `F.interpolate`.

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

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

```python
>>> 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
  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)
```