Transposed Conv as Matrix Multiplication explained

Raymond Kwok · Follow

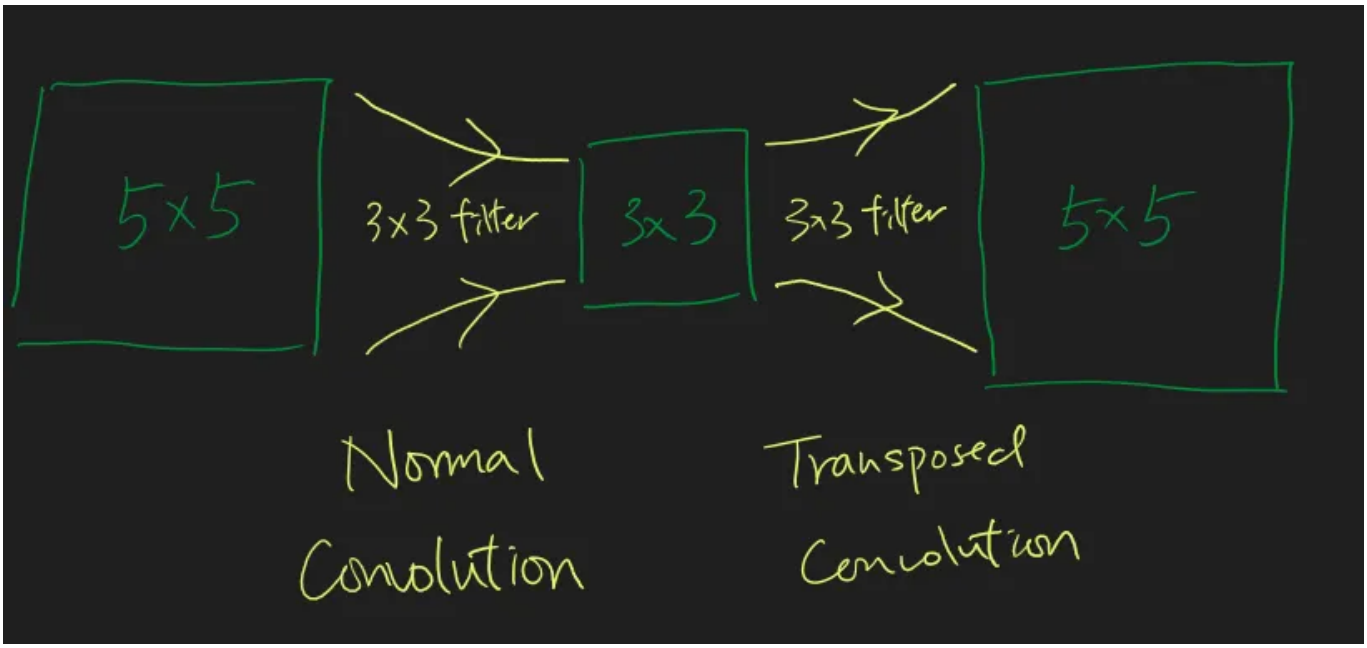
8 min read · Jan 15

Explained and implemented transposed Convolution as matrix multiplication in numpy. Why it is called transposed convolution, and comparisons with Tensorflow and Pytorch are covered.

This article assumes prior knowledge of neural network convolutational kernels. My implementation is in Numpy. The comparisons with Tensorflow and PyTorch are going to show when these implementations act the same and when differently.

Introduction

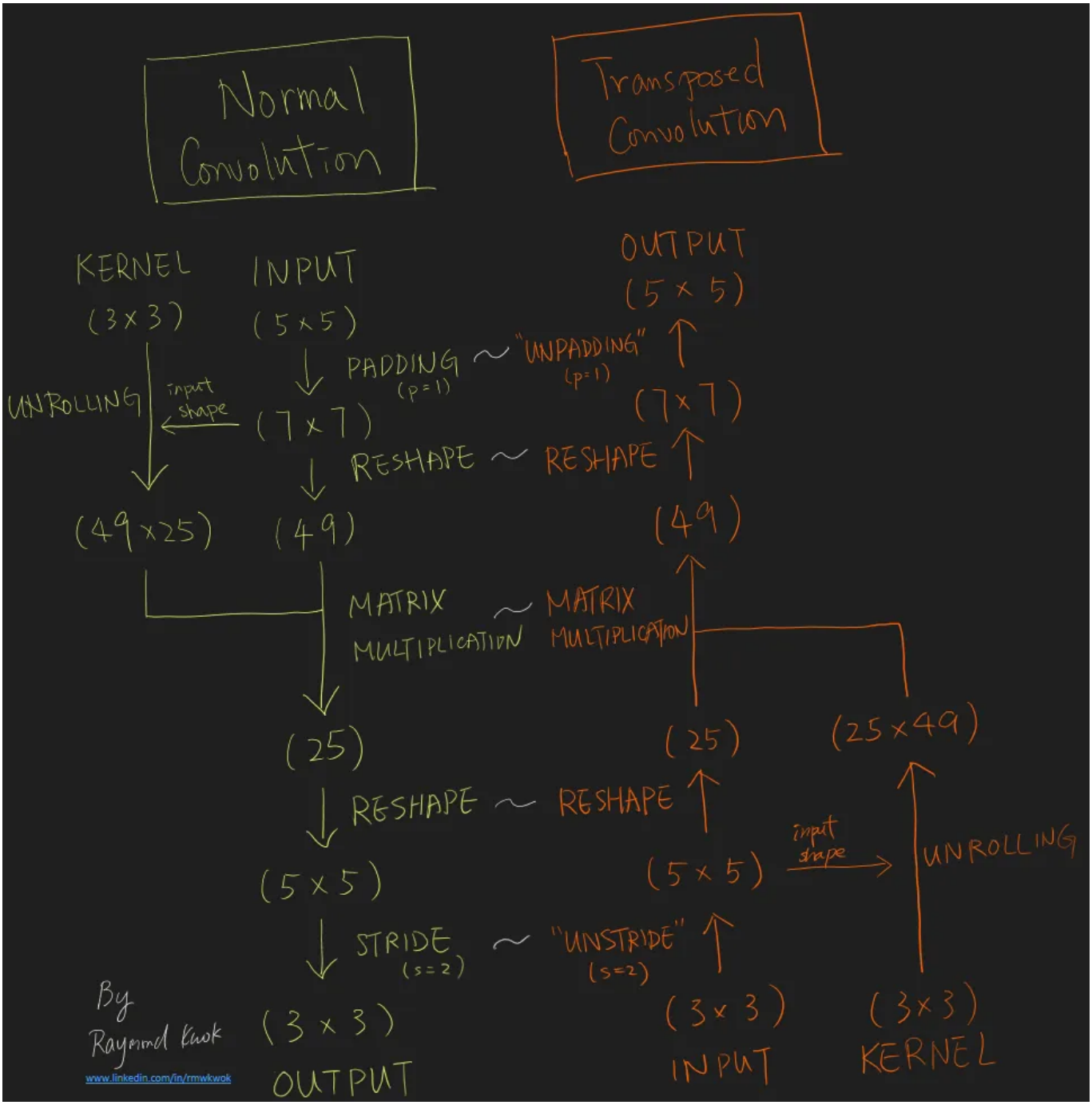
I do not find many articles explaining transposed convolution as a matrix multiplication and comparing with results from Tensorflow's or Pytorch's implementations, so I have written this.



I think the idea behind the Transposed Convolution is as simple as an operation that can up-sample a down-sampled (by normal convolution) matrix back to its original shape. The key here is about getting back the ***same shape*** as before, and with this key functionality, we can learn an autoencoder that encodes with normal convolution and decodes with transposed convolution. Getting back the same values as before, however, is not the concern because down-sampling is not always lossless which means it is not always reversible at all.

Comparing Normal and Transposed Convolutions

Here is how I would break down the two operations to implement (in NumPy) and compare them:



Breakdown of normal and transposed convolutions. This is not the most efficient way (in terms of performance) of breaking it down but it does show all the components that I want to show.

As shown in the flow charts, the two operations are pretty much the reverse operations of each other, however, I guess it is called the transposed convolution and not “reversed convolution” probably because on the one hand, it only recovers the shape of the pre-convoluted matrix instead of their values, and on the other hand, their unrolled kernels are the ***transpose*** of one

another. This can be seen from their shapes that in the normal case, it is (49 × 25) whereas in the transposed case, it is (25 × 49).

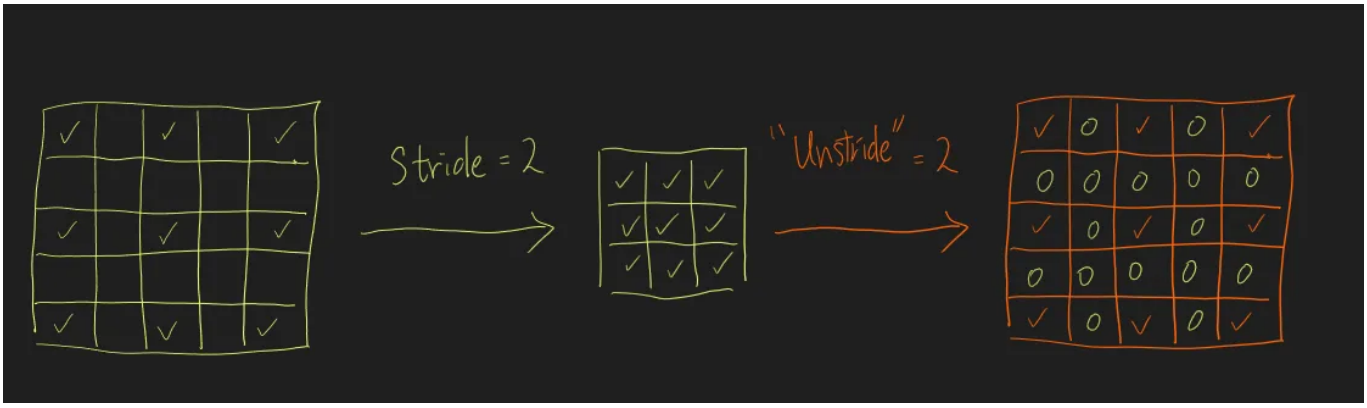
Until now...

I think we have seen two important features of the transposed convolution, which are “shape recovery” and the “transpose relationship in the two unrolled kernels”. Next we will go a bit deeper into the implementation and discuss when the recovery of the output shape becomes ambiguous.

Stride and “un-stride”

I guess nowhere else other than this article calls it as “un-stride”, so be careful if you want to use this term ;).

A normal convolutional stride skips some convoluted pixels (or it skips convoluting some pixels), and in a transposed convolution, we insert back those pixels and fill them with zeros. This step makes sure the shape is right, not the values.

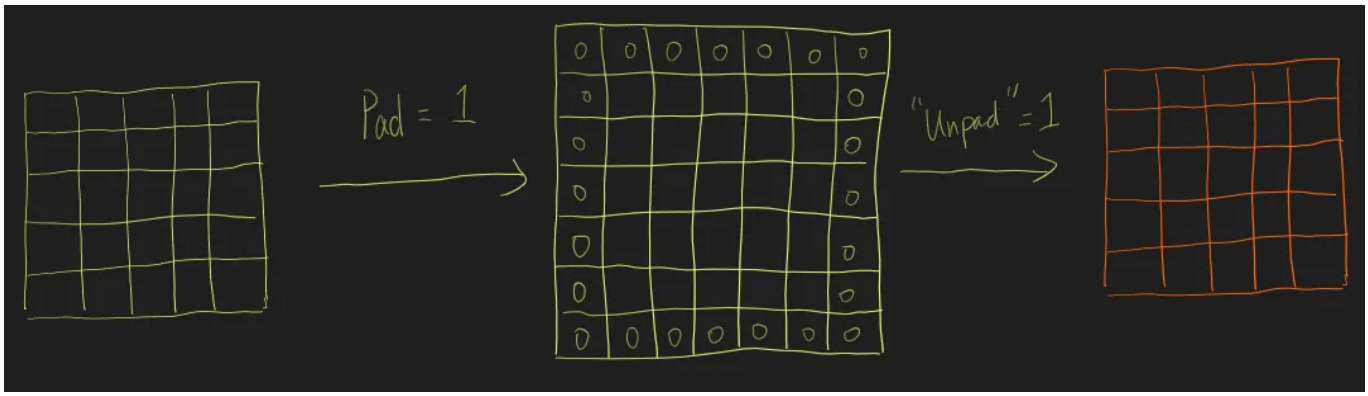



```
def _stride(inputs, strides, mode, output_padding=None):
    sh, sw = strides
    m, c, h, w = inputs.shape

    if mode == 'normal': # get every other N rows/columns
```

```
        outputs = inputs[:,:,::sh,::sw]

    elif mode == 'transposed': # add N rows/columns of 0 between two rows/column
        _h = h + (h - 1) * (sh - 1) + output_padding
        _w = w + (w - 1) * (sw - 1) + output_padding
        outputs = np.zeros((m,c,_h,_w), dtype=np.float32)
        outputs[:,:,::sh,::sw] = inputs

    return outputs
```

Padding and “Un-padding”

“Cropping” is probably a better name than “un-padding” to express the idea of taking away the extra pixels that we have added when normally convoluting. Again, this step keeps the shape.

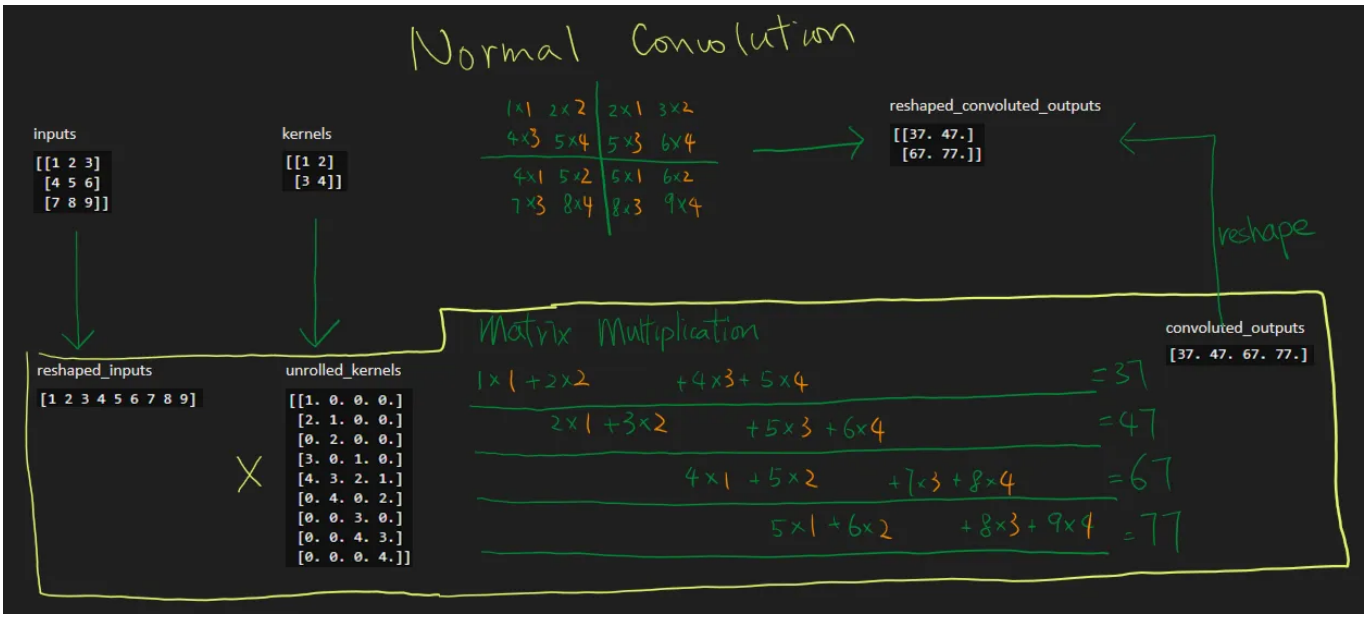



```
def _padding(inputs, padding, mode):
    p = padding
    m, c, h, w = inputs.shape

    if mode == 'normal': # do padding
        outputs = np.zeros((m, c, h+2*p, w+2*p), dtype=np.float32)
        outputs[:,:,p:h+p,p:w+p] = inputs

    elif mode == 'transposed': # do cropping
        outputs = inputs[:,:,p:h-p,p:w-p]
```

```
    return outputs
```

Kernel unrolling

This is perhaps the most important step, because it makes convolution a matrix multiplication. A convolution element-wise multiply the kernel with different parts of the input. Therefore, we can just create an unrolled kernel matrix, in which we "arrange" the values such that when it gets matrix-multiplied by the flatten version of the input, all those element-wise multiplications can be recovered in order.



Normal convolution. We can convolute a (2 × 2) kernel on a (3 × 3) input via the upper or the lower path. The upper path shifts the kernel to all of the four possible places, do the element-wise multiplications, and sum the results to get the (2 × 2) convoluted output. The lower path flattens the input and unrolls the kernels in a way that their matrix multiplication will maintain all element-wise multiplications in the right places such that the matrix multiplication product can be reshaped to recover the same convoluted output.

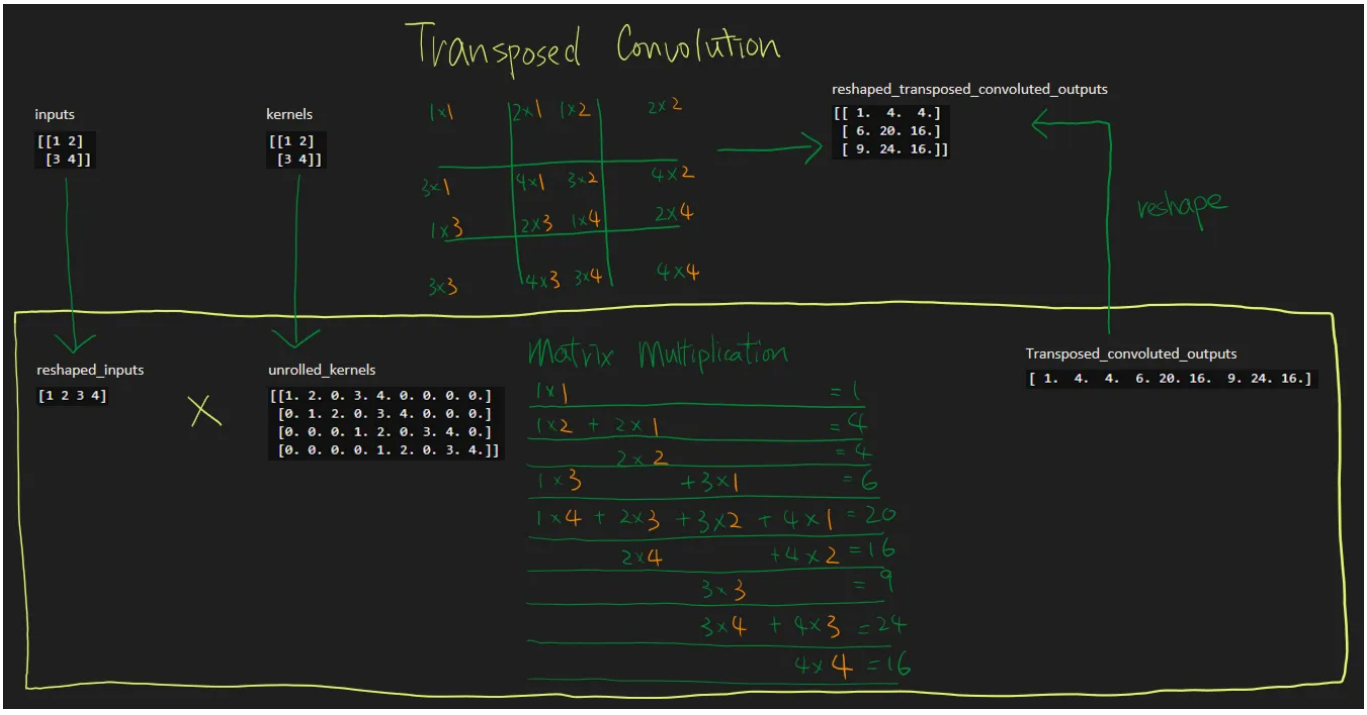
```
from conv2d_transpose_numpy import _unroll_kernel, _flatten, \
                                   compute_outputs_shape

inputs = np.arange(1, 10).reshape((1,1,3,3))
kernels = np.arange(1, 5).reshape((1,1,2,2))
```

```
    outputs_shape = compute_outputs_shape(inputs, kernels, 1, 'normal')

    reshaped_inputs = _flatten(inputs)
    unrolled_kernels = _unroll_kernel(inputs, kernels, outputs_shape, 'normal')
    convoluted_outputs = np.tensordot(reshaped_inputs, unrolled_kernels, ((1,2),(0,1
    reshaped_convoluted_outputs = convoluted_outputs.reshape(*outputs_shape)
```

Just as a (9 x 4) unrolled kernel can transform a (9) reshaped input into a (4) convoluted output as shown in the above chart, a (4 x 9) unrolled kernel can back-transform a (4) reshaped input into a (9) transposed-convoluted output as shown in the chart below.

You may refer to this Andrew Ng's video on how to compute the Transposed Convolution in the way I show as the upper path in the chart below.

We want to note that the unrolled kernels in the two charts are the transpose of one another.



Transposed convolution. We can transposed-convolute a (2 × 2) kernel on a (2 × 2) input via the upper or the lower path. The upper path multiply each element of the kernel with the whole inputs, put the results in the right places, and sum to get the (3 × 3) transposed-convoluted output. The lower path flattens the input and unrolls the kernels in a way that their matrix multiplication will maintain all element-wise multiplications in the right places such that the matrix multiplication product can be reshaped to recover the same transposed-convoluted output.

```
from conv2d_transpose_numpy import _unroll_kernel, _flatten, \
                                   compute_outputs_shape

inputs = np.arange(1, 5).reshape((1,1,2,2))
kernels = np.arange(1, 5).reshape((1,1,2,2))
outputs_shape = compute_outputs_shape(inputs, kernels, 1, 'transposed')

reshaped_inputs = _flatten(inputs)
unrolled_kernels = _unroll_kernel(inputs, kernels, outputs_shape, 'transposed')
transposed_convoluted_outputs = np.tensordot(reshaped_inputs, unrolled_kernels,
reshaped_transposed_convoluted_outputs = transposed_convoluted_outputs.reshape(*
```

Compute outputs shape

The `compute_outputs_shape()` function was used in the above code, but its definition is left to discuss in this section. Pre-computing the output shape is required for constructing the unrolled kernel in the right shape.

In a normal convolution, we know for sure that the output size is given by the following formula:

$$OutputSize = \left\lfloor \frac{InputSize - KernelSize + 2 \times PadSize}{StrideSize} \right\rfloor + 1$$

Therefore, for a transposed convolution, in order for it to recover the same shape given the same kernel size, pad size, and stride size, we can change the subject of the above formula to *InputSize* and because output becomes input in the transposed convolution, we can swap the two and the formula becomes:

$$OutputSize = (InputSize - 1) \times StrideSize + M + KernelSize - 2PadSize$$

$$\text{where } M = (OutputSize - KernelSize + 2PadSize) \bmod (StrideSize)$$

Note that because in the normal convolution formula we have the floor operation (⌊x⌋), consequently, after changing the subject, we will have the modulo operation (x mod y). We will see the ambiguity brought by these operations by first examining the following normal convolutions under *StrideSize= 2*, *PadSize = 1* and *KernelSize = 3*:

Normal Convolution

InputSize	*Output Size*
3	2
4	2

Now, due to the floor operation, even a different *InputSize* can give the same *OutputSize*. This makes transposed convolution ambiguous because its goal

is shape recovery: if we transposed-convolute a (2 x 2) input, should it produce a (3 x 3) output or a (4 x 4) output?

If we look again at the formula for transposed convolution, if $M = 0$, then $OutputSize = 3$, and if $M = 1$, then $OutputSize = 4$, so the modulo term M gives us the control to choose between the two, as expected.

However, if we check out the PyTorch documentation for its torch.nn.ConvTranspose2d or the Tensorflow documentation for its tf.keras.layers.Conv2DTranspose, we find neither of them has the modulo term M. Instead they have a parameter called `output_padding` for us to specify how many additional padding to add to one side of each dimension in the output shape.

As shown in my breakdown, I treat padding, stride, "unpadding", and "unstride" as individual steps, the output shapes that I need to compute does not have to consider their contributions. Consequently, after taking away M, $StrideSize$, and $PadSize$, my outputs shape formula for normal convolution becomes:

$$OutputSize = InputSize - KernelSize + 1$$

and that for transposed convolution becomes:

$$OutputSize = (InputSize - 1) \times StrideSize + KernelSize + OutputPadSize$$

```
def compute_outputs_shape(inputs, kernels, strides, mode, output_padding=None):
    sh, sw = strides
    im, ic, ih, iw = inputs.shape
    kn, kc, kh, kw = kernels.shape

    if mode == 'normal':
        outputs_shape = (
            im,
            kn,
            ih - kh + 1,
            iw - kw + 1)

    elif mode == 'transposed':
        outputs_shape = (
            im,
            kc,
            (ih - 1) * sh + kh + output_padding,
            (iw - 1) * sw + kw + output_padding)

    return outputs_shapeComparisons with Tensorflow's, and PyTorch's implementat
```

The first one comes the normal convolution. The check compares my implementation (denoted as "Numpy") with them using `np.allclose`.

```
from tests import test_normal_convolution

strides = (1, 1)
inputs_shape = (4, 3, 5, 5) #(Samples, N Channels, Height, Width)
kernels_shape = (2, 3, 3, 3) #(N Kernels, N Channels, Height, Width)

rng = np.random.default_rng(10)
inputs = rng.random(inputs_shape)
kernels = rng.random(kernels_shape)

test_normal_convolution(inputs, kernels, strides, print_arrays=False)

# Outputs:
# Normal Convolution tests
```

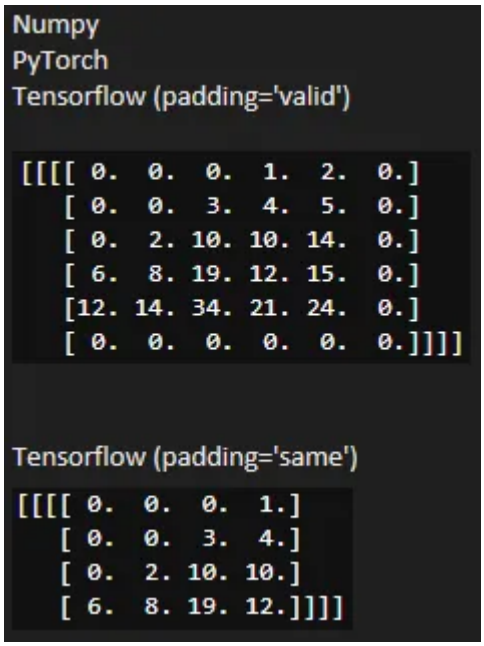
```
# AllClose Check: TensorFlow vs Numpy: Passed
# AllClose Check: Torch vs Numpy: Passed

# Output Shapes:
# Numpy: (4, 2, 3, 3)
# PyTorch: (4, 2, 3, 3)
# Tensorflow: (4, 2, 3, 3)
```

Next comes the transposed convolution. The last two rows of the output printouts show some shapes returnable by Tensorflow's implementation.

```
from tests import test_transposed_convolution

strides = (2, 2)
padding = 0
output_padding = 1

inputs_shape = (5, 3, 2, 2) #(Samples, N Channels, Height, Width)
kernels_shape = (3, 1, 3, 3) #(N Channels, N Kernels, Height, Width)

rng = np.random.default_rng(10)
inputs = rng.random(inputs_shape)
kernels = rng.random(kernels_shape)

test_transposed_convolution(inputs, kernels, strides, padding, output_padding,
                            print_arrays=False)

# Outputs
# Transposed Convolution tests

# AllClose Check: Torch vs Numpy: Passed

# Output Shapes:
# Numpy: (5, 1, 6, 6)
# PyTorch: (5, 1, 6, 6)

# Various Tensorflow Output Shapes:
# padding='valid' (5, 1, 6, 6)
# padding='same'  (5, 1, 4, 4)
```

In case of `padding=0`, we should have set `padding='valid'` for Tensorflow, and therefore all implementations produce the same output shape.

We can take a look at their outputs, but for us to easily compare, I replaced the randomized inputs and kernels with some integers, so that we can see that the smaller matrix is just a sub-matrix of a bigger one.



Note that the smaller matrix is a sub-matrix of the larger one.

Summary

We have broken down and compare the normal and the transposed convolutions to see their similarities and differences. We have seen the ability of the transposed convolution to recover shapes but there can also be ambiguity which is handled by the parameter `output_padding`. We have also gone through how to do both convolutions as matrix multiplications, and seen that their unrolled kernels are transpose to each other.

Lastly, my breakdown is not the best in terms of performance. For example, the stride process is placed after the matrix multiplication which means we

will first compute some multiplications but then drop their results due to the stride. If you are interested in the full code, please visit my Git Repository.

References

[1] Dumoulin, V., and Visin, F. (2016) A guide to convolution arithmetic for deep learning. arXiv:1603.07285

[2] Shi, W. et al. (2016) Is the deconvolution layer the same as a convolutional layer? arXiv:1609.07009

[3] Im D. J. et al. (2016) Generating images with recurrent adversarial

Written by Raymond Kwok

140 Followers

Data scientist (linkedin.com/in/rmwkwok)