Transposed Convolutions

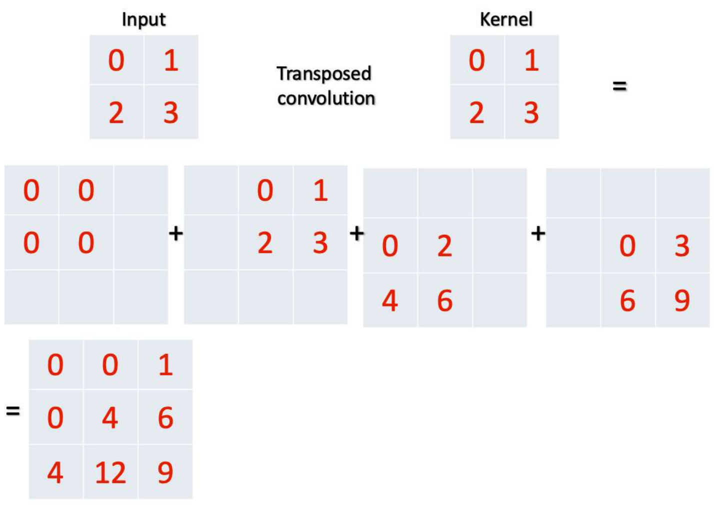
As we saw in the previous labs, Convolutional Neural Networks (CNNs) can be used to automatically extract features from images and videos for computer vision problems like image classification. CNNs use convolutional layers that convolve over our input images, find patterns, and update weights during the training procedure, thus allowing for learning to occur. The need for transposed convolutions comes in when we want the opposite of this to happen, that is, use a transformation going in the opposite direction of a normal convolution so we can invert the output of a convolutional layer and reconstruct the original input image. This is referred to as deconvolution in some sources, but refers to a different concept. Deconvolution reverts the process of a convolution, whereas transposed convolution carries out a regular convolution but reverts its spatial transformation.

Transposed convolutional layers, specifically `Conv2DTranspose` in the Keras API, can be used for this purpose. They learn a set of weights that can be used to reconstruct original inputs and can be trained jointly with convolutional layers during the training process. In this reading, we will cover transposed convolutions in detail and see how they work in Keras.

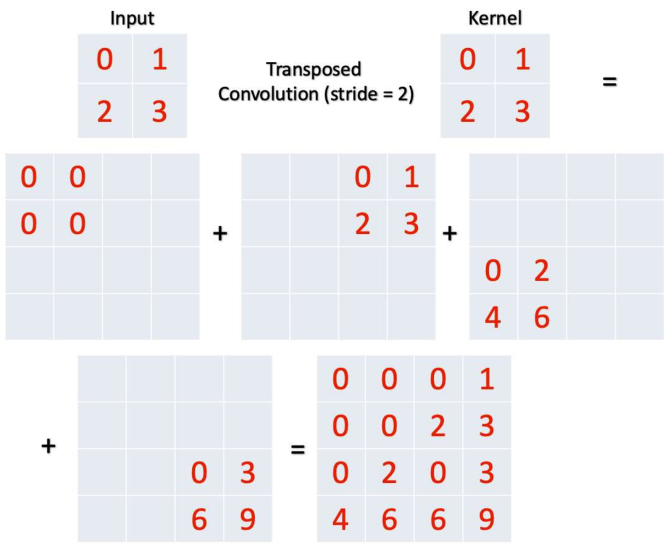
What are transposed convolutions?

Transposed convolutions are used for increasing (or upsampling) the input feature map to a desired output feature map. In other words, they are used for finding the original representation of a convolutional filter map by reversing the downsampling operations performed by the convolution.

The following image shows a basic transposed convolution operation with stride = 1 and padding = 0. Suppose that we are given a $n_h x n_w$ input image and a $k_h x k_w$ kernel. We start by simply sliding the kernel window with a stride of 1 for n_w times in each row and n_h times in each column. This would result in a total of n_h*n_w intermediate results, each of which would be a $(n_h+k_h-1)x(n_w+k_w-1)$ tensor with 0 initial values. We multiply each element in the input tensor by the kernel. This results in a $k_h x k_w$ which takes the position of the element in the intermediate tensor. Finally, we sum up all intermediate results to get our output.



Now let's walk through a slightly more complicated example. Here we use a stride of 2. In transposed convolutions, instead of specifying strides for inputs, we specify stride for the output or the intermediate tensors. As seen in the image below, when changing stride from 1 to 2, we increase both the n_h and n_k of intermediate tensors by 1. Given the same 2 x 2 input image and 2 x 2 kernel window, we start by sliding the kernel window over two rows and two columns and generating four intermediate result tensors of shape 4x4. We get our final output by summing up the intermediate tensors.



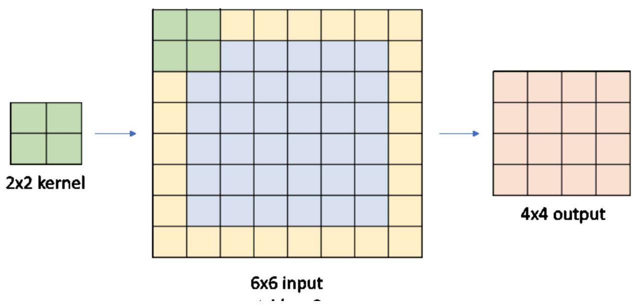
Examples of transposed convolutions

Now that we understand how transpose convolutions work and why they are used, let us walk through a few examples of convolution and transposed convolutions.

Example 1: Convolutions with Stride 2, Padding 1

We will start off with a convolution example, where we set stride to 2 and padding to 1; that is, we extend all image edges by 1 pixel (with values set to 0). This implies that both the height and width of the input image have grown by 2.

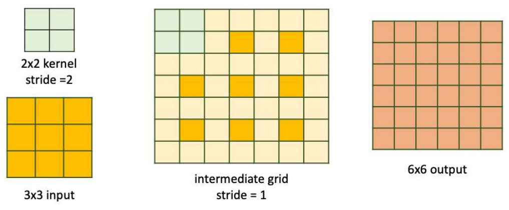
Now we will apply the 2x2 kernel to extend the 8x8 padded input image. Moving with a stride of 2, the kernel can take up 4 positions across the image horizontally and 4 positions vertically. This would result in a 4x4 output image.



Example 2: Transposed Convolutions with Stride 2, No Padding

The transposed convolution is the opposite of a normal convolution and is used to expand a tensor to a larger tensor. We first start by creating an intermediate grid that has the original input's cells spaces apart with a step size that is equal to the stride. The cells in between have zero values. To ensure the kernel in the top left covers one of the original cells, we extend the edge of the intermediate image with additional cells with a value of 0.

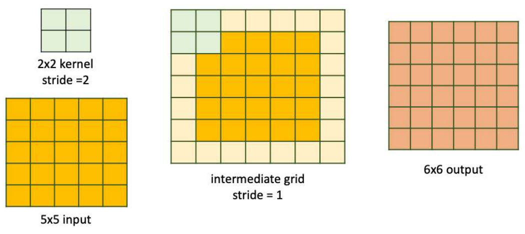
When it comes to transposed convolutions, the step size is always 1. The stride is just used to set how far apart the original cells are in the intermediate grid. On the intermediate grid, we shift the kernel by step sizes of 1. This results in a 6x6 output image, which is larger than the original 3x3 input.



Example 3: Transposed Convolutions with Stride 1, No Padding

In this example, we keep everything else the same as Example 2 but use a stride of 1. The transposed convolution process is exactly the same. This time the original cells are spaced apart without a gap in the intermediate grid as the step size is 1.

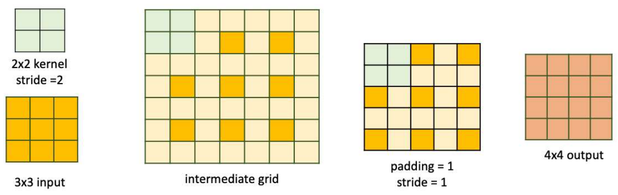
Just like we did before, we expand the intermediate grid with the maximum number of additional outer rings so that a kernel in the top left can still cover one of the original cells. Then we shift the kernel with a step size of 1 over this intermediate 7x7 grid to get a 6x6 output image.



Example 4: Transposed Convolutions with Stride 2, Padding 1

In this example, we use a padding of 1. Unlike normal convolution in Example 1, in transposed convolution, padding is used to expand the input image instead of reducing it. We set the 3x3 input image, a 2x2 kernel, a padding of 1, and a stride of 2.

Just like we did in Example 3, we create an intermediate grid where the original cells are spaced 2 steps apart, and the grid is expanded until the kernel covers the original cells. After setting padding to 1, we can remove one ring from around the intermediate grid, as seen in the image below. Applying the 2x2 kernel on the padded intermediate image results in a 4x4 output.



Conv2DTranspose in the Keras API

In this part of the reading, we will cover how to use 2D transposed convolutions in Keras to build an autoencoder for reconstructing MNIST hand-written digits images that have been encoded into the lower-dimensional state before. This is how Conv2DTranspose is represented within the Keras API, and the official Keras documentation on this layer is found here:

```
keras.layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='valid', output_padding=None, data_format=None, dilation_rate=(1, 1),
```

An autoencoder is a neural network that learns efficient data representations in an unsupervised manner.

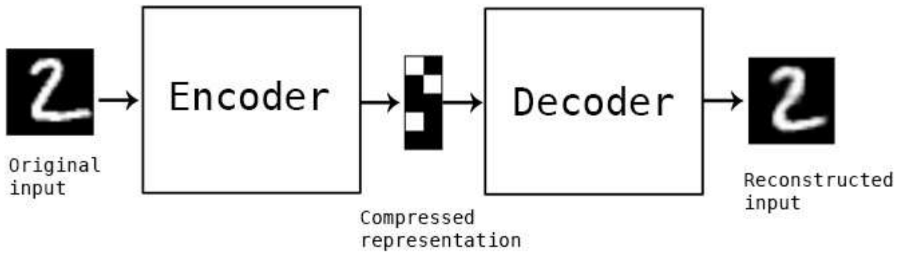


Image Source

On a high level, we feed our network an image that is fed through the encoder to obtain some encoded representation; that is, a lower-dimensional representation of our image. This embedding is fed to the decoder that attempts to reconstruct the original input image. We will build a convolution-based autoencoder using the `Conv2DTranspose` layer.

We will start by loading the MNIST data, reshaping and normalizing the data, converting the numbers to floats, and defining a few model configuration options, which we will explain throughout the reading.

▶ Click here to check out the code

Now, we will define our autoencoder model's architecture. We have an input `Sequential()` layer, followed by three `Conv2D` layers. These form the encoder part of our network. To upsample the encoded state back into a higher-dimensional format (that is, the 28x28 pixel data), we use three `Conv2DTranspose` layers that form the decoder of the network. The last `Conv2D` layer convolves over the upsampled data, reshaping the images to the original dimension of 28x28. Let us print out the model summary.

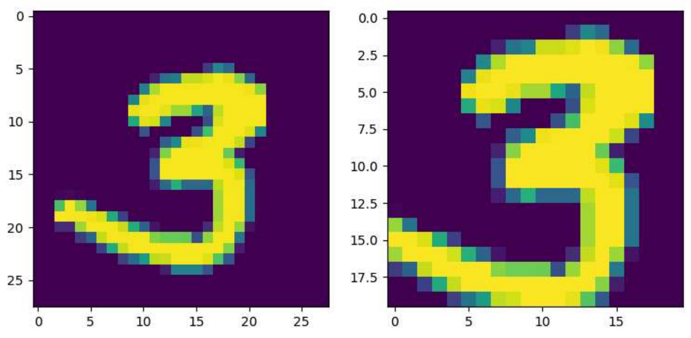
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 16)        4624
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 22, 22, 8)         1160
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 24, 24, 8)         584
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 26, 26, 16)        1168
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 28, 28, 32)        4640
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 1)         289
=================================================================
Total params: 12,785
Trainable params: 12,785
Non-trainable params: 0
_________________________________________________________________
```

▶ Click here to check out the code

Using the Adam optimizer and binary cross-entropy loss function, we can compile the model. We set a batch size of 1000, train our model over 25 epochs, use ten classes, and use 20% of the data will for validation. Using these model configuration options, we can train our model (this process should take ~30 minutes).

▶ Click here to check out the code

Once the model has finished training, we can check out the output to see how well the MNIST digits have been reconstructed.



As seen, using Conv2DTranspose for the decoder part worked well, and the images have been reconstructed pretty successfully. Transposed convolutions can also be used upscaling images to higher resolutions and for semantic segmentation tasks (such as from RGB input images to class-based visualization).

In this reading, we learned the theory behind transposed convolutions. We also went through an example that used convolutional layers for the encoder segment and transposed convolutions for the decoder segment of an autoencoder model.

References

Keras Blog. (n.d.). Building Autoencoders in Keras. Retrieved from https://blog.keras.io/building-autoencoders-in-keras.html

MachineCurve. (2019, September 29). Understanding transposed convolutions. Retrieved from https://www.machinecurve.com/index.php/2019/09/29/understanding-transposed-convolutions

Wikipedia. (2006, September 4). Autoencoder. Retrieved from https://en.wikipedia.org/wiki/Autoencoder

Keras Blog. (n.d.). Building Autoencoders in Keras. Retrieved from https://blog.keras.io/building-autoencoders-in-keras.html

Transposed convolutions. Retrieved from https://d2l.ai/chapter_computer-vision/transposed-conv.html

Authors

Kopal Garg

Kopal Garg is a Masters student in Computer Science at the University of Toronto.

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2022-07-17	0.1	Kopal Garg	Create Reading
2022-09-09	0.1	Steve Hord	QA pass edits