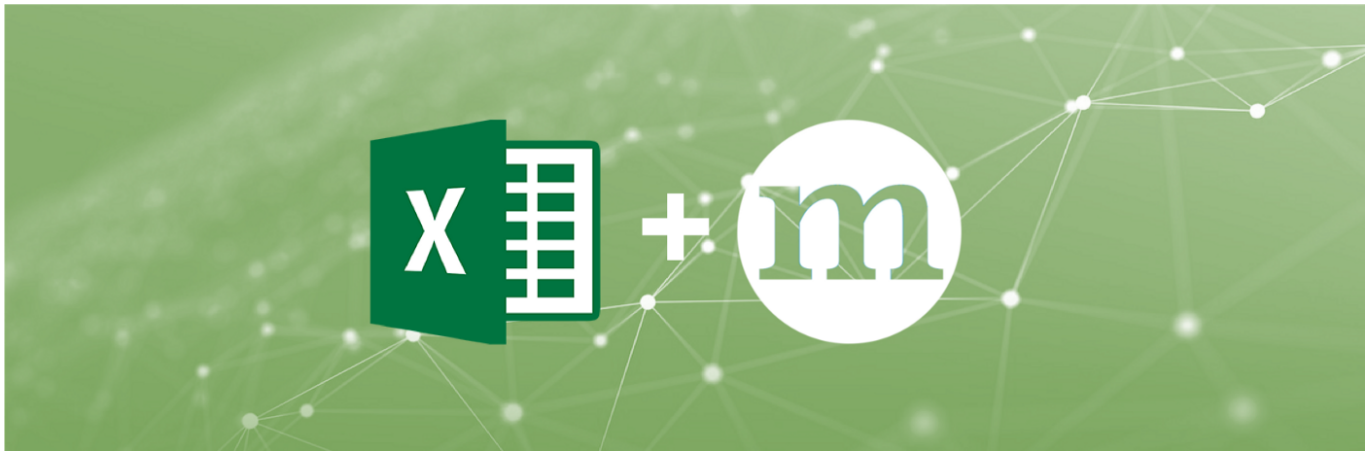
1D & 3D Convolutions explained with... MS Excel!

Thom Lane Follow
Oct 17, 2018 · 7 min read



Welcome back to the blog post series on convolutions using MS Excel and Apache MXNet Gluon. Just incase you missed the first post you can learn all about 2D Convolutions here. Our next instalment in the series will take us into another dimension! We're going to look at 1D and 3D Convolutions.

. . .

1D Convolutions

We started with 2D Convolutions in the first post because convolutions gained significant popularity after successes in the field of Computer Vision. Since tasks in this domain use images as inputs, and natural images usually have patterns along two spatial dimensions (of height and width), it's more common to see examples of 2D Convolutions than 1D and 3D Convolutions.

More recently though, there have been a huge number of successes in applying convolutions to natural language processing tasks too. Since tasks in this domain use text as inputs, and text has patterns along a single spatial dimension (i.e. time), *1D Convolutions* are a great fit! We can use them as more efficient alternatives to traditional recurrent neural networks (RNNs) such as LSTMs and GRUs. Unlike RNNs, they can be run in parallel for really fast computations.

Another domain that benefits from 1D Convolutions is time series modelling. As before, we have a single spatial dimension on our input data (i.e. time), and we want to pick up patterns over periods of time. We often call this type of data 'sequential' because it can be viewed as a sequence of values. And the spatial dimension we often call the 'temporal' dimension. Using 1D Convolutions before RNNs is quite common too: the LSTNet model is an example of this and its predictions for traffic forecasting are shown below.

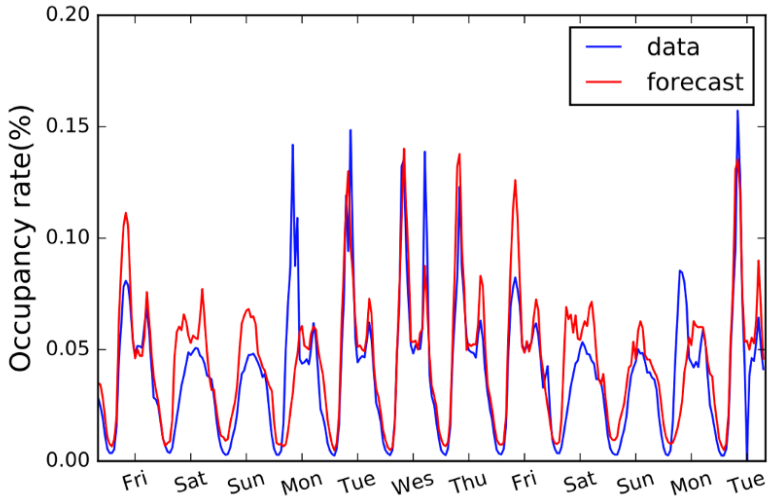
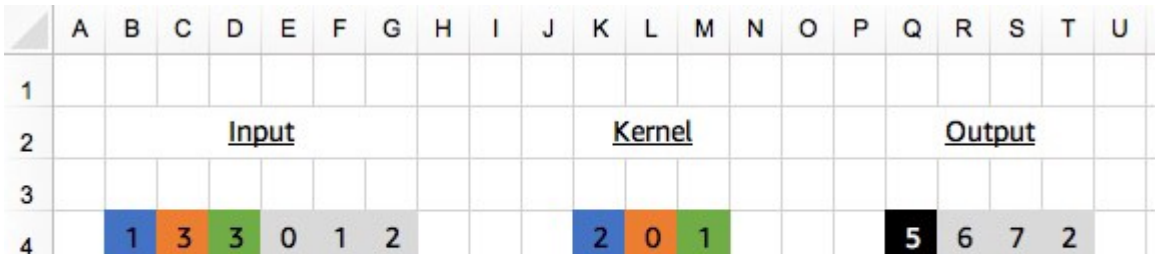


Figure 1: LSTNet using 1D Convolution for time series forecasting of traffic (source)

A, B, C. It's easy as (1,3,3) dot (2,0,1) = 5.

With naming conventions clarified, let's now take a closer look at the arithmetic. We're in luck because 1D Convolutions are actually just a simplified version of the 2D Convolution!



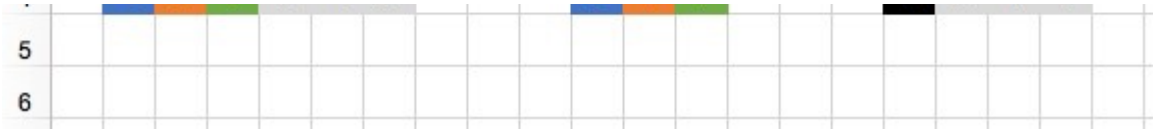


Figure 2: A 1D Convolution with kernel of size 3, applied to a 1x6 input matrix to give a 1x4 output.

Working through the calculation of a single output value, we can apply our kernel of size 3 to the equivalently sized region on the left hand side of our input array. We calculate the 'dot product' of the input region and the kernel, which to recap is just the element-wise product (i.e. multiply the colour pairs) followed by a global sum to obtain a single value. So in this case:

$(1*2) + (3*0) + (3*1) = 5$

We apply the same operation (with the same kernel) to the other regions of the input array to obtain the complete output array of (5,6,7,2); i.e. we slide the kernel across the whole input array. Unlike 2D Convolutions, where we slide the kernel in two directions, for 1D Convolutions we only slide the kernel in a single direction; left/right in this diagram.

> *Advanced: a 1D Convolution is not the same as a 1x1 2D Convolution.*

Coding things up

Unsurprisingly, we'll need a `Conv1D` Block in Gluon for our 1D Convolution. We define the kernel shape as 3, and since we're just working with a single kernel in these example, we'll specify `channels=1`.

```
import mxnet as mx

conv = mx.gluon.nn.Conv1D(channels=1,
                          kernel_size=3)
```

We can now pass our input into the `conv` block using a predefined kernel.

```
input_data = mx.nd.array((1,3,3,0,1,2))
kernel = mx.nd.array((2,0,1))
```

```
# see appendix for definition of `apply_conv`
output_data = apply_conv(input_data, kernel, conv)
print(output_data)

# [[[5. 6. 7. 2.]]]
# <NDArray 1x1x4 @cpu(0)>
```

What changes with padding, stride and dilation?

We saw the effect of padding, stride and dilation in the last post on 2D Convolutions, and it's very similar for 1D Convolutions. With padding, we only pad along a single dimension (the temporal dimension) this time. Our diagram represents time across the horizontal axis, so we pad to the left and right of the input data (and not above and below as in the 2D Convolution example). With stride and dilation, we only apply them along the temporal dimension too. So an example with padding and stride would look as follows:

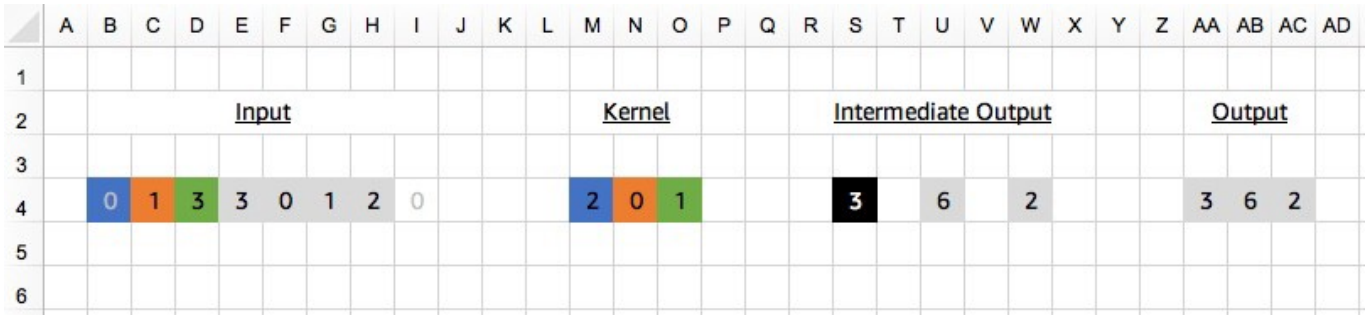


Figure 3: A 1D Convolution with kernel of size 3, padding of 1 and stride of 2, applied to a 1x6 input matrix to give a 1x3 output.

We add two additional arguments in the code: `padding=1` and `strides=2`.

```
conv = mx.gluon.nn.Conv1D(channels=1, kernel_size=3,
                          padding=1, strides=2)

output_data = apply_conv(input_data, kernel, conv)
print(output_data)

# [[[3. 6. 2.]]]
# <NDArray 1x1x3 @cpu(0)>
```

> *Advanced: the dimensions of input data must conform to a particular order called the layout. By default the 1D convolution expects to be applied to input data of the format 'NCW', i.e. batch size (N) * channels (C) * width/time (W). And for 2D Convolutions the default is NCHW, i.e. batch size (N) * channels (C) * height (H) * width (W). Check out the* `layout` *argument of* `Conv1D` *and* `Conv2D`*.*

. . .

3D Convolutions

With 1D and 2D Convolutions covered, let's extend the idea into the next dimension! A 3D Convolution can be used to find patterns across 3 *spatial* dimensions; i.e. depth, height and width. One effective use of 3D Convolutions is object segmentation in 3D medical imaging. Since a 3D model is constructed from the medical image slices, this is a natural fit. And action detection in video is another popular research area, where multiple image frames are concatenated across a temporal dimension to give a 3D spatial input, and patterns are found across frames too.

Figure 4: Stack of frames as potential input for a 3D Convolution (source)

> *Advanced: Spatial dimensions are distinct from the channels dimension. We'll find out more about this in the next post. Although colour images have 3 channels, there are still only 2 spatial dimensions (of height and width), making 2D Convolutions more appropriate in this case. And RGB videos will still have 3 spatial dimensions (of time, height and width), making 3D Convolutions ideal.*

Our kernel shape for a 3D Convolution is specified along 3 dimensions. In the example below we see a 2x2x2 kernel. Ordering is depth, height and width unless specified otherwise. When thinking about the convolution operation in terms of a kernel sliding

across a multidimensional input array, for a 3D Convolution, the kernel slides in 3 directions: in Figure 5 this is left/right, up/down and forward/backwards. At every step the dot product is calculated, giving us a 3D output too. Clearly MS Excel is going to struggle a little with this, so let's take a look at an example with three.js first.

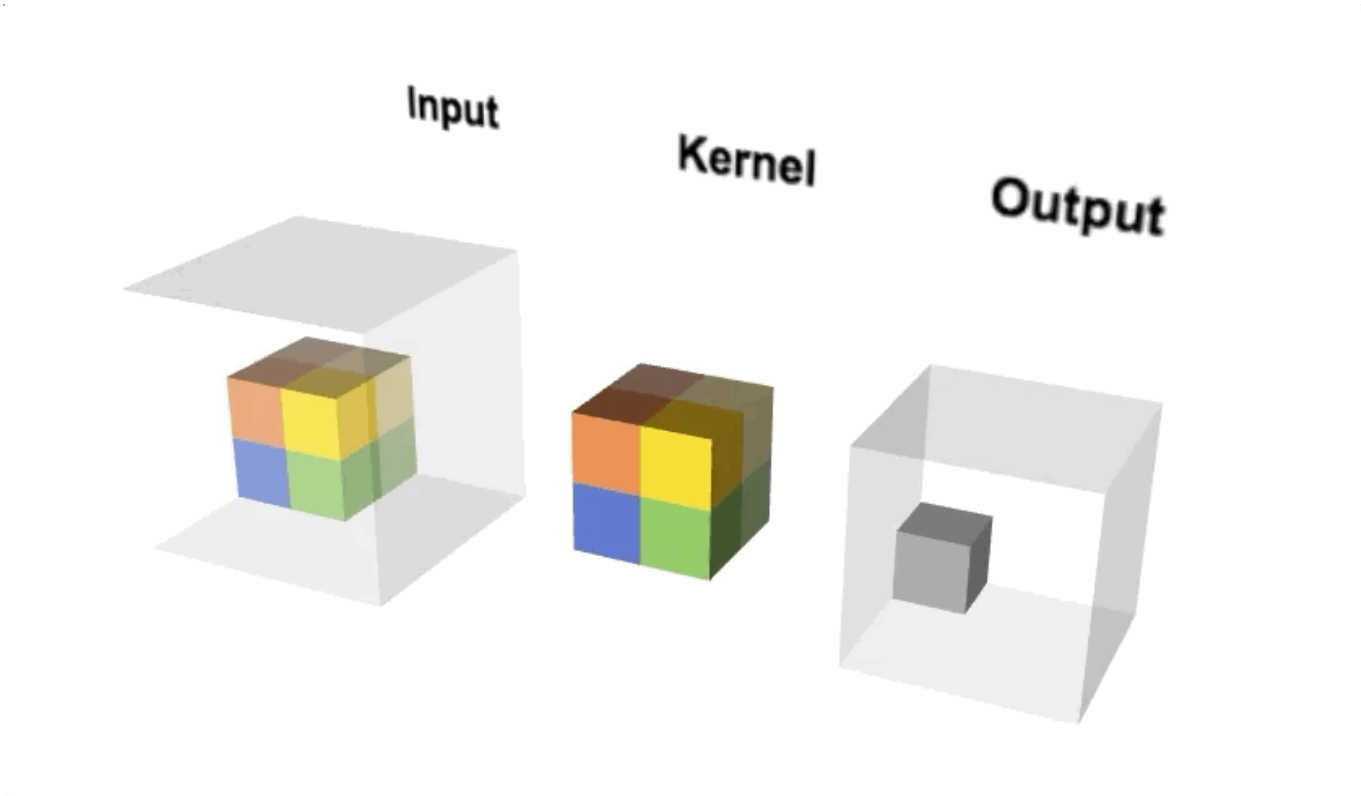
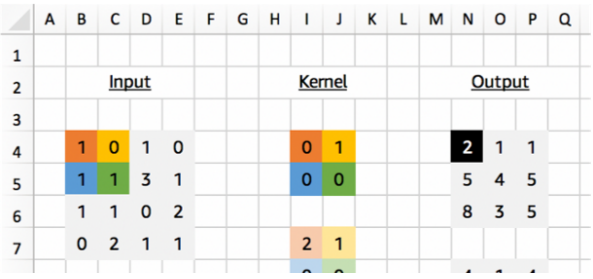


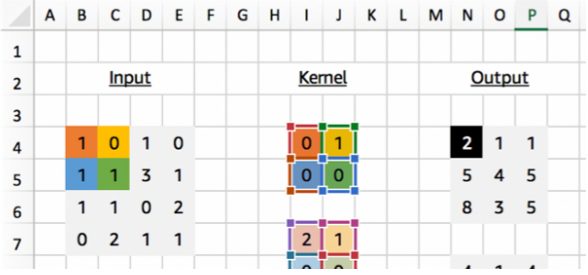
Figure 5: A 3D Convolution with kernel of size 2x2x2, applied to a 4x4x4 input matrix to give a 3x3x3 output. (source)

Check out this link for an interactive version of the diagram above. Well, slightly interactive, you can change the viewing angle at least! We're now going to venture where nobody has ever dared venture before… we're going to implement a 3D Convolution in MS Excel. So if I'm being honest I forgot about 3D Convolutions before deciding the theme of these posts: but don't worry, because I deliver on my promises.

Closest top left corner



Furthest bottom right corner



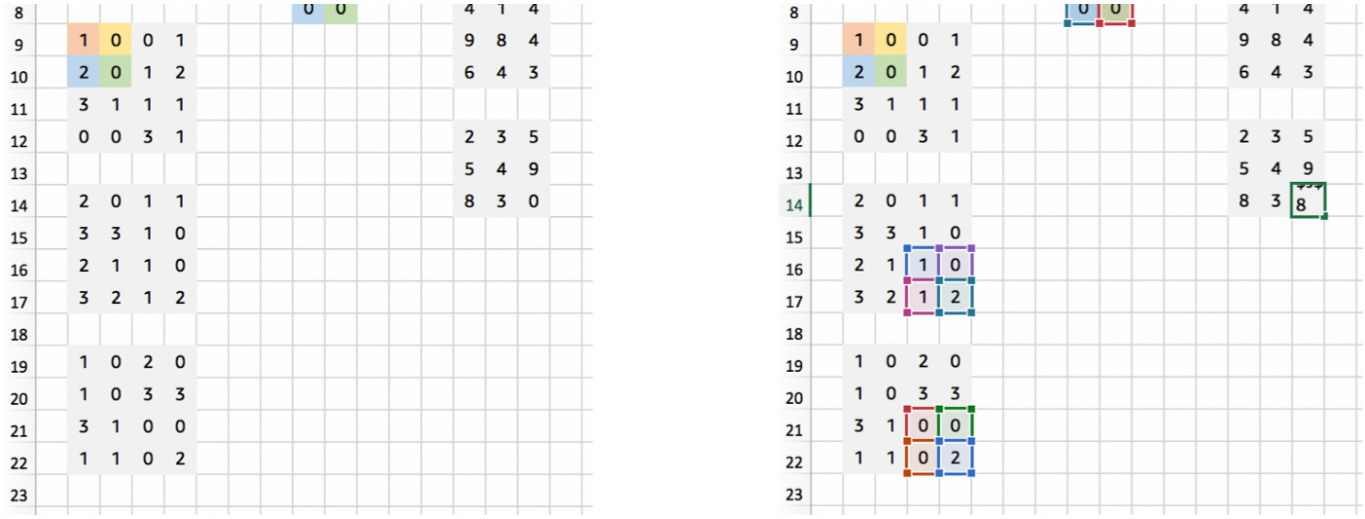


Figure 6: A 3D Convolution with kernel of size 2x2x2, applied to a 4x4x4 input matrix to give a 3x3x3 output.

Using dummy input data and kernel parameters, we can calculate the furthest bottom right corner of the output array using the following formula:

Figure 7: Excel formula for cell P14

Okay, let's move quickly onto the code before we lose our minds. Thankfully things are much simpler in the world of MXNet Gluon. We just need to create a `Conv3D` Block and we're good to go!

```
conv = mx.gluon.nn.Conv3D(channels=1, kernel_size=(2,2,2))
```

> *Advanced: although we're being explicit about the shape of our kernel here, and use* `(2,2,2)`*, we could just specify* `kernel_size=2` *as a shorthand in this case.*

```
# define input_data and kernel as above
# input_data.shape is (4, 4, 4)
# kernel.shape is (2, 2, 2)

output_data = apply_conv(input_data, kernel, conv)
print(output_data)

# [[[[2. 1. 1.]
#    [5. 4. 5.]
```

```
#     [8. 3. 5.]]

#    [[4. 1. 4.]
#     [9. 8. 4.]
#     [6. 4. 3.]]

#    [[2. 3. 5.]
#     [5. 4. 9.]
#     [8. 3. 0.]]]]]
# <NDArray 1x1x3x3x3 @cpu(0)>
```

Stride, padding and dilation are as you'd expect, and operate along each of the 3 spatial dimensions. We'd pad on all sides of our cube in Figure 5 (unless we're using uneven padding), and we'd stride and dilate the kernel in each direction.

Get experimental

All the examples shown in this blog post can be found in these MS Excel Spreadsheets for Conv1D and Conv3D (or on Google Sheets here and here respectively). Click on the cells of the output to inspect the formulas and try different kernel values to change the outputs. And after replicating your results in MXNet Gluon, I think you can officially add 'convolutional ninja' as a title on your LinkedIn profile!

Up next

We've actually been working with a slightly simplified version of the truth (sorry to break the news), but in the next post we'll uncover the reality of channels and multiple convolution kernels. We'll then be able to work with multivariate time series, sequences of word embeddings, and RGB images!

Appendix

Thanks to Simon Corston-Oliver.

Machine Learning | Convolutional Network | Mxnet | Computer Vision | Naturallanguageprocessing