Module 4

Multichannel Convolutions

Most tutorials on convolutions use the example of a 2D image matrix applied to a 2D kernel filter. However Convolutional Neural Networks (CNN) generally operate on multi-channel images. When you have a "deep" network, you're stacking convolutional layers. When this happens you will deal with multi-channel images.

The input image may be an RGB image. This means you have a 3D volume: Height x Width x Channels. Let's say 100x100x3. A convolutional layer is intended to learn visual patterns relative to the kernel size. If we set our kernel size to 5x5. Then we are looking for small patterns that fit into a 5x5 kernel. This could be a curve, a straight line... or something else. However since our input image is 100x100x3, we're not dealing with 2D image matrices anymore. We're dealing with 3D volumes. This means our convolutional kernel filter also has to be 3D.

How does this work? The exact same way as before with a 2D matrix applied to 2D kernel filter. Element wise multiplication then reduce to a single number by summing. This is often called a dot product operation. Take a 3D slice of the 3D image, and do the dot product with the 3D kernel filter.

So for our RGB image 100x100x3, we would have a 5x5x3 convolutional kernel filter. In a CNN this means we are learning 75 weights for a single pattern. Here I'm using "weight" as a single number in a cell of a convolutional kernel filter.However we may want to learn more than just 1 pattern. So we increase the number of convolutional kernel filters. Let's say we think that there are 32 possible visual patterns with our RGB image. Then it's a simple matter of creating 32 convolutional kernel filters, each of which is of size 5x5x3.

This is why the Tensorflow conv2d actually allows you to specify input channels and output channels. The output channels is exactly equal to the number of kernel filters you want to learn.

The number of biases is exactly equal to the number of output channels.

As you stack up convolutional layers, you're creating different feature maps that is associated with each particular pattern you're trying to learn, which is represented by an output channel. Higher convolutional layers are then learning more composite patterns because each of their convolutional kernel filters has to learn weights for each of the input channels.

These numbers allow you to quickly estimate how much resources your CNN model requires. For an input image that is 100x100x3 to be applied to a 5x5x3 kernel to produce 32 output channels means there is 75 weights per output channel, with a total of 75 * 32 = 2400 weights overall.

Module 4

To calculate the width and height of the output feature map:

Oh = floor((Ih + 2*Ph - Kh / Sh) - 1)
Ow = floor((Iw + 2*Pw - Kw / Sw) - 1)
Where the P is padding, K is the kernel, S is the stride.

But what about Od?

The output depth is specified by the user. It's part of the function. It is equal to the "number" of filters that exists.

Now filters are not necessarily 2D, they can be 3D... etc. It depends on the input feature map.

So if the input feature map is 100x100x4. Then the kernel is KhxKwx4. Then there are 64 of these, if the the specified output feature map is 64. So the total dimensions of the weights is KhxKwx4x64.

This is why the size of the kernel varies as the size of the input and size of the output varies. So if we decide that the kernel height and width is 7 and 7. Then the input dimensions and output dimensions all affect how many weights there are.

Therefore for a Convolution Operation:

Iw = given
Ih = given
Id = given
Ph = given
Pw = given
Sh = given
Sw = given

Kw = specified
Kh = specified
Kd = Id
Kn = specified :"number of kernels"

B = Kn :"bias"

Oh = floor((Ih + 2*Ph - Kh / Sh) - 1)
Ow = floor((Iw + 2*Pw - Kw / Sw) - 1)
Od = Kn

ConvolutionWeights = (Kw * Kh * Kd * Kn) + B
ConvolutionByteSize = ConvolutionWeights * Bytes per Number
Memory consumption of a feature map is bytes per weight * number of weights. With a FP32 system, 4 bytes per weight. So the weights is the total number of numbers in the kernel.

Module 4