Convolutional Neural Networks (CNNs)

Swapnoneel Kayal

15 June-21 June 2021

1 Introduction

1.1 The Convolution Operation

Let us consider the example of vertical edge detection. Consider this matrix:

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

Suppose the above 6×6 matrix is a small grayscale image in which we are trying to detect vertical edges. We use the following convolutional filter for the purpose of vertical edge detection:

1	0	-1
1	0	-1
1	0	-1

To obtain a matrix representing the vertical edges in the image, we convolve the filter over the image. The convolution operation superimposes the filter on different parts of the image, and creates a matrix of the weighted sums of the overlapping area of the image in each superposition. (The filter matrix acts as weights.) Thus, the convolution operation in this case gives:

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

In this result, the whiter pixels (cells having higher pixel values) are the vertical edges detected by the convolutional operation. This can be understood better by convolving over the following matrix:

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

Convolving the vertical edge detector on this gives us:

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0

As we can clearly see, the detector detects an edge along the central vertical region, and we know that to be true from the image itself. This gives us some intuition on the convolution operation.

1.2 Padding and Strides

The result of a convolution (if done as shown above) is smaller in size than the original image. Convolving an $f \times f$ filter over an $n \times n$ image results in an image of size $(n-f+1) \times (n-f+1)$. But often, we may want a result of the same size. To achieve this, we use padding.

In the padding technique, we pad the original image to make it larger, by filling in new pixels outside its border (the value is usually zero). The additional thickness (number of border layers added) is called the padding p. To achieve a result of the same size as the image, we use the value of padding $p = \dfrac{f-1}{2}$ (because $n+2p-f+1 = n$). Note that here we are assuming the convolutional filter to have odd size, which is usually the case. We cannot achieve 'same' padding with an even sized convolutional filter. 'Same' padding refers to padding such that the output has the same shape as the input, whereas 'valid' padding refers to not padding at all.

Sometimes, to save time or to achieve some specific purpose, we may not want to superimpose the convolutional filter in every possible place on the image. In that case, we use strided convolutions, where we slide the convolutional filter by some stride length s (in both horizontal and vertical directions) each time before writing the weighted sum into the result. The original case discussed before corresponds to stride $s = 1$.

1.3 Convolutions over Volume

In handling images with multiple channels (like RGB or HSV images) in our neural network, we have to convolve filters over volumes. In this technique, we convolve a filter with multiple channels (channels in filter = channels in image) onto the image and obtain a two-dimensional result by convolution. For example:

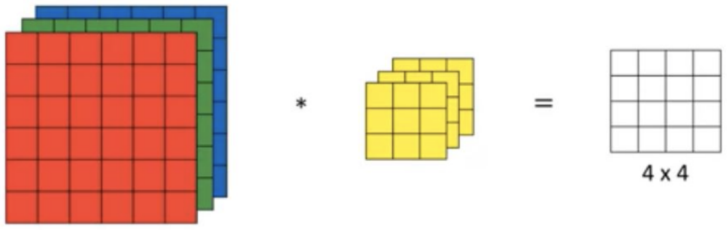

Figure 1: Convolution over a volume.

It's still just like before, only with 3D filters. Also, we may often convolve multiple filters over the image (like if we want to detect both vertical and horizontal edges). Suppose we have an image of shape $n \times n \times n_c$, and we convolve n'_c filters of shape $f \times f \times n_c$ over the image. Then, the result would have the shape $(n - f + 1) \times (n - f + 1) \times n'_c$.

In a way, this is exactly what a convolutional layer does in a neural network. Taking the activations of the previous layer, in the form of a 3D matrix, the convolutional layer convolves several filters over the volume (the filter weights are learned through gradient descent) and passes them on to the activation function in the form of another 3D matrix.

1.4 Pooling Layers

A pooling layer is quite similar to convolution, but in this case, we do not output filter-weighted sums of input values. In pooling layers, we output the maximum (max-pooling) or average (average pooling) value of the values overlapping with the filter. Here, we have an input of size $n \times n \times n_c$, a filter of size $f \times f$, padding p, and stride s. Then, the output shape of the pooling layer is going to be $\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times n_c$.

Note that there are no parameters to be learned in pooling layers. The backpropagation algorithm simply passes through this layer to its previous layers during gradient descent. In a CNN, convolutional layers are almost always followed by max-pooling layers.

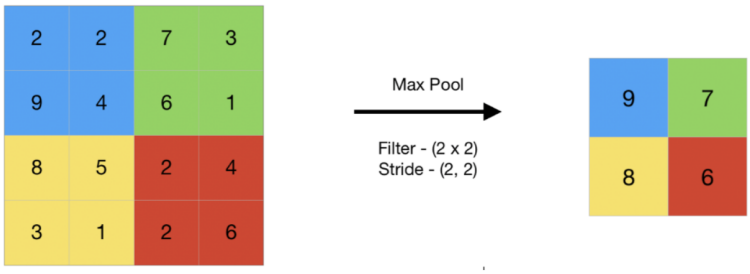

Figure 2: An example of a max-pooling layer.

An example of a complete CNN, with convolutional layers and max-pooling layers, and fully connected layers at the end, is shown here:

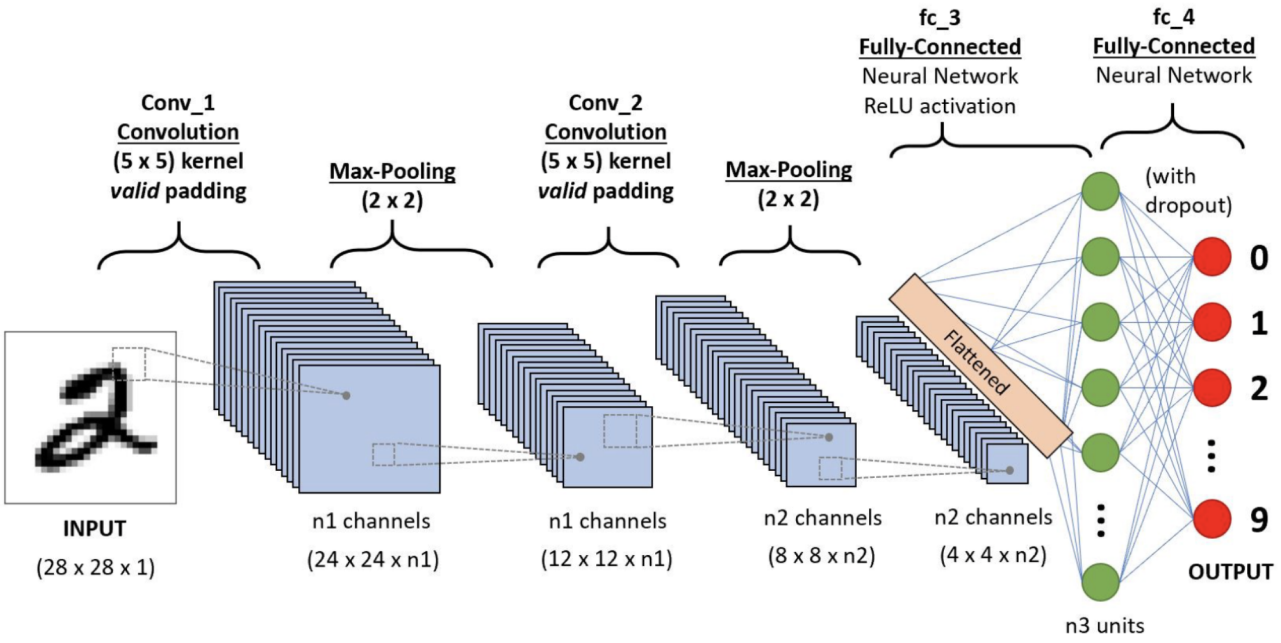

Figure 3: An example of a complete CNN.

This CNN takes as input a hand-written digits (from the MNIST dataset), and classifies it as a digit from 0 to 9. There are two pairs of convolutional and max-pooling layers, followed by two fully-connected (or dense) layers. As we can see, the CNN goes on decreasing the width and height of the data and increasing the number of channels, before finally flattening it out into a 1D array. This technique, used by CNNs, enables them to detect and classify things regardless of which part of the image they appear in.

1.5 Why Convolutions?

There are two main incentives for using convolutions:

1. Parameter Sharing: In the convolution operation, we move the same filter (of parameters) over the entire data (from the previous layer), and thus we use the same parameters over and over again. Thus, convolution works on the basis that a feature detector that is useful in one part of an image may be useful elsewhere too.

2. Sparsity of Connections: In a convolutional layer, each output value depends on only a small part of the input. Thus, convolutional layers are able to break down an image and analyze its parts independently.

Since computer vision tasks may have the target object in any arbitrary part of the image, convolutions are able to do quite well in such tasks, thanks to the above two characteristics.

2 CNN Architectures

2.1 The LeNet-5 Network Architecture

The LeNet-5, a classic CNN, has the following architecture:

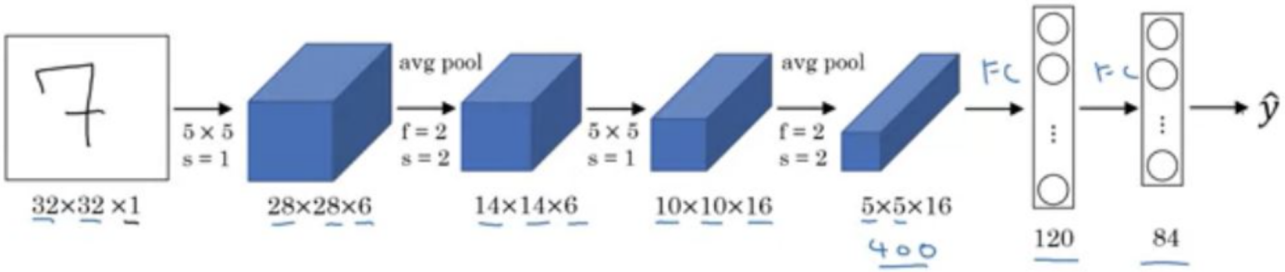

Figure 4: The LeNet-5 architecture.

The diagram shows its working on the MNIST dataset. There are two convolutional layers, two average-pooling layers, and two fully connected layers. It is clearly quite similar to Figure 14.3. It turns out that the LeNet-5 has close to 60 thousand parameters.

2.2 The AlexNet Network Architecture

The AlexNet, another classic CNN, has the following architecture:

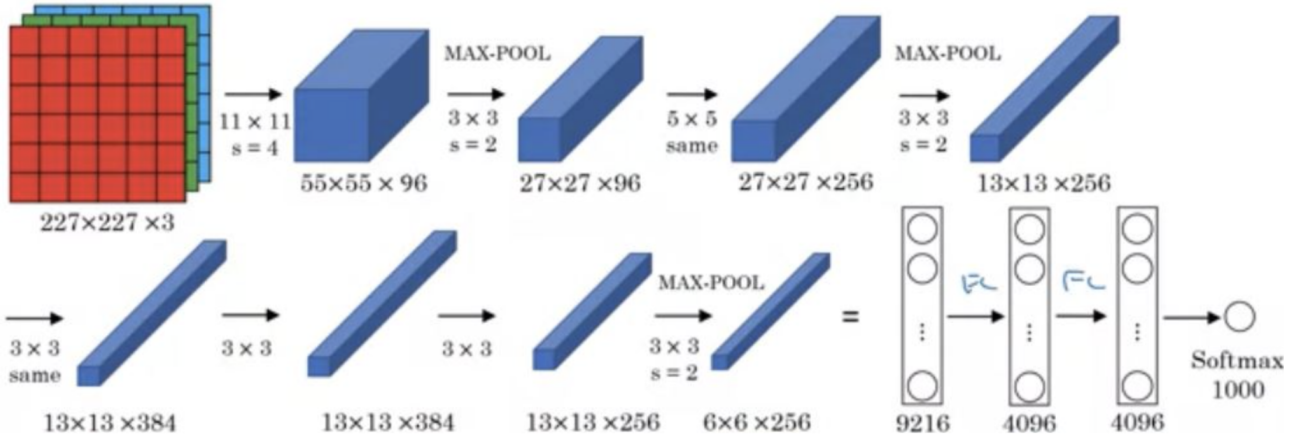

Figure 5: The AlexNet architecture.

AlexNet is quite similar to LeNet-5, just a lot bigger. AlexNet has close to 60 million parameters. Note that the original AlexNet had some other layers too, called Local Response Normalization (LRN) layers, which normalized all the channel values at each x and y. This was later found to be not very useful, so these layers have been left out.

2.3 The VGG-16 Network Architecture

Yet another classical CNN, called the VGG-16, has the following architecture:

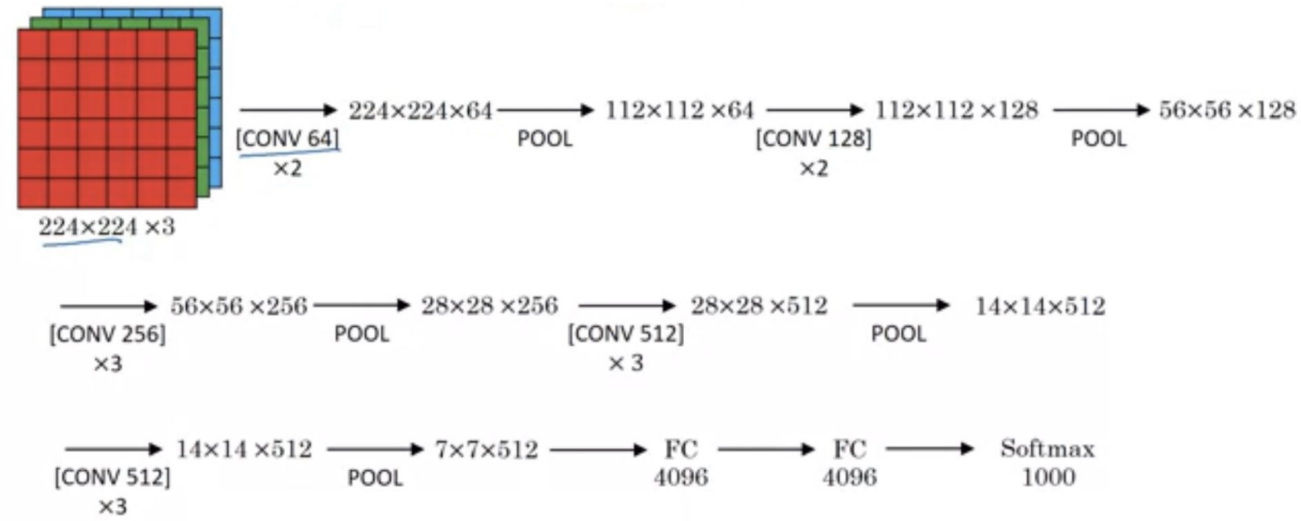

Figure 6: The VGG-16 architecture.

This also has a similar architecture to the networks above, but even larger. The VGG-16 network has nearly 138 million parameters.

2.4 Residual Networks

Residual networks, also called ResNets, differ from classical CNN architectures in that they are made up of residual blocks. Residual blocks were designed as a solution for the problem of vanishing or exploding gradients in very deep neural networks. A residual block looks like this:

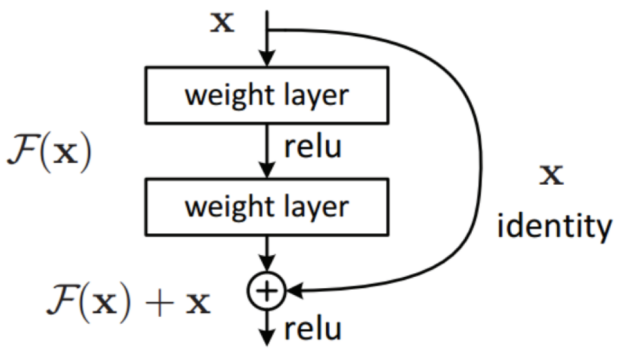

Figure 7: Diagram of a residual block.

Basically, in a residual block, we have:

$$
\begin{aligned}
z^{[l+1]} &= W^{[l+1]} a^{[l]} + b^{[l+1]} \\
a^{[l+1]} &= g(z^{[l+1]}) \\
z^{[l+2]} &= W^{[l+2]} a^{[l+1]} + b^{[l+2]} \\
a^{[l+2]} &= g(z^{[l+2]} + W_s a^{[l]})
\end{aligned}
$$

Here, W_s is an additional trainable parameter matrix for the skip connection. The main flow in Figure 15.4 is called the main path, whereas the side path is called the skip connection. Skip

connections allow activations of a layer to pass to another layer deeper in the network, and thus, due to shorter paths, exploding/vanishing gradients are avoided. A ResNet looks like this:

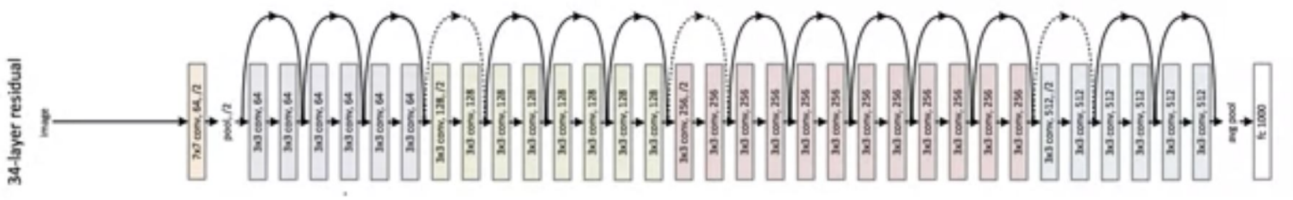

Figure 8: Diagram of a residual network.

2.5 Inception Networks

An inception network combines different kinds of convolutions for a single layer of a network. In a single transition from one layer to another, several different kinds of operations are performed (with 'same' padding to ensure that the results have the same width and height), such as convolution with a number of 1×1, 3×3, 5×5 filters, as well as pooling filters. All the results are concatenated together along the depth dimension before passing them on to the next layer.

However, there is one drawback of the above technique; the computational cost is very high this way. Making each layer's computations more complex would result in inefficient models. To solve this problem, we use a 1×1 convolution over the activations of the previous layer, to squeeze down the data, before applying a 3×3 or a 5×5 convolution.

We can understand this through an example. Suppose we have activations of size $28 \times 28 \times 192$. To get a result of size $28 \times 28 \times 32$, we apply 32 filters of size $5 \times 5 \times 192$ with 'same' padding. The number of computations we have in this case is $(5 \times 5 \times 192) \times (28 \times 28 \times 32) \approx 120$ million. On the other hand, if we first convolve 16 filters of size $1 \times 1 \times 192$ over the activations, and then convolve 32 filters of size $5 \times 5 \times 16$ on that, the number of computations we must do is $(1 \times 1 \times 192) \times (28 \times 28 \times 16) + (5 \times 5 \times 16) \times (28 \times 28 \times 32) \approx 12.4$ million. Thus we see a drastic improvement in computational efficiency with this trick.

An inception module thus looks like this:

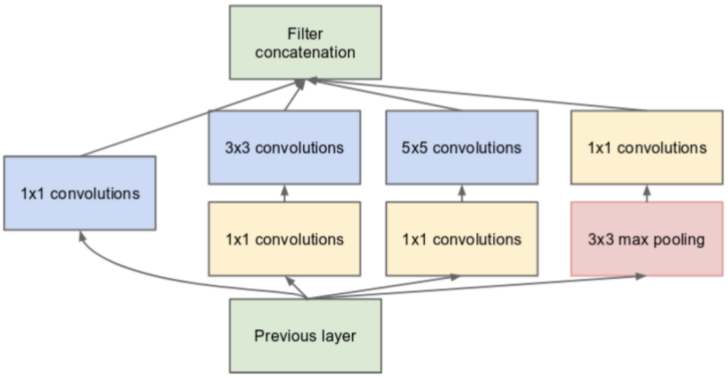

Figure 9: Diagram of an inception block.

Note that the 1×1 convolution after max pooling is for the purpose of decreasing the depth of the outcome of max pooling, because this outcome would be as deep as the previous layer activations, and a lot deeper than the outcomes of other convolutions in the inception block. A complete inception network looks like this:

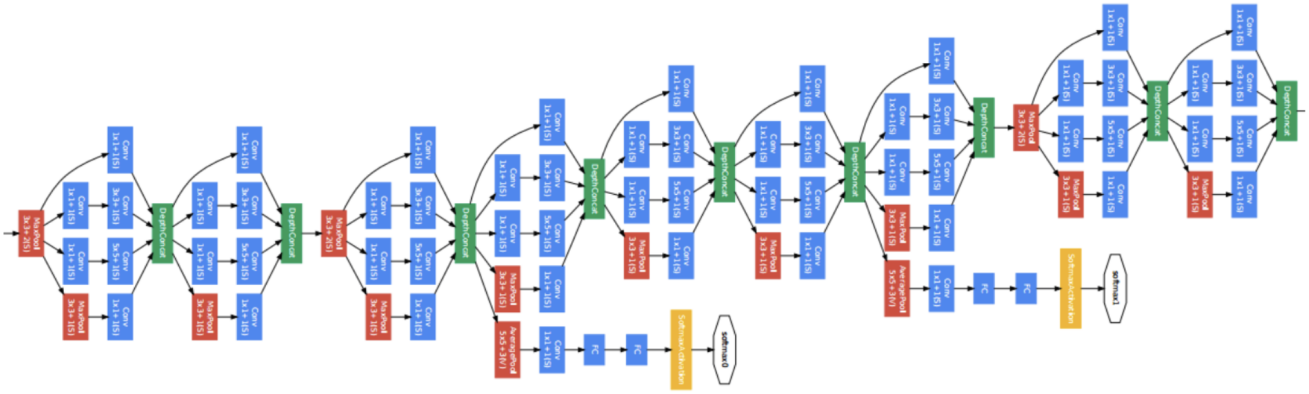

Figure 10: Diagram of an inception network.