Gentle Dive into Math Behind Convolutional Neural Networks

Mysteries of Neural Networks Part V

Piotr Skalski Follow
Apr 12, 2019 · 12 min read

Autonomous driving, healthcare or retail are just some of the areas where Computer Vision has allowed us to achieve things that, until recently, were considered impossible. Today the dream of a self driving car or automated grocery store does not sound so futuristic anymore. In fact, we are using Computer Vision every day — when we unlock the phone with our face or automatically retouch photos before posting them on social media. Convolutional Neural Networks are possibly the most crucial building blocks behind this huge successes. This time we are going to broaden our understanding of how neural networks work with ideas specific to CNNs. Be advise, the article will include quite complex math equations, but **don't be discouraged if you are not comfortable with linear algebra and differential calculus**. My goal is not to make you remember those formulas, but to provide you with the intuition of what is happening underneath.

Side notes: For the first time I decided to enrich my artwork with an audio version and I kindly invite you to listen to it. You will find a link to Soundcloud above. In this article I focus mainly on things typical to CNNs. If you are looking for more general information about deep neural networks I encourage you to read my other posts from this series. As usual, full source code with visualizations and comments can be found on my GitHub. Let's start!

Introduction

In the past we got to know the so-called densely connected neural networks. These are networks whose neurons are divided into groups forming successive layers. Each such unit is connected to every single neuron from the neighboring layers. An example of such an architecture is shown in the figure below.

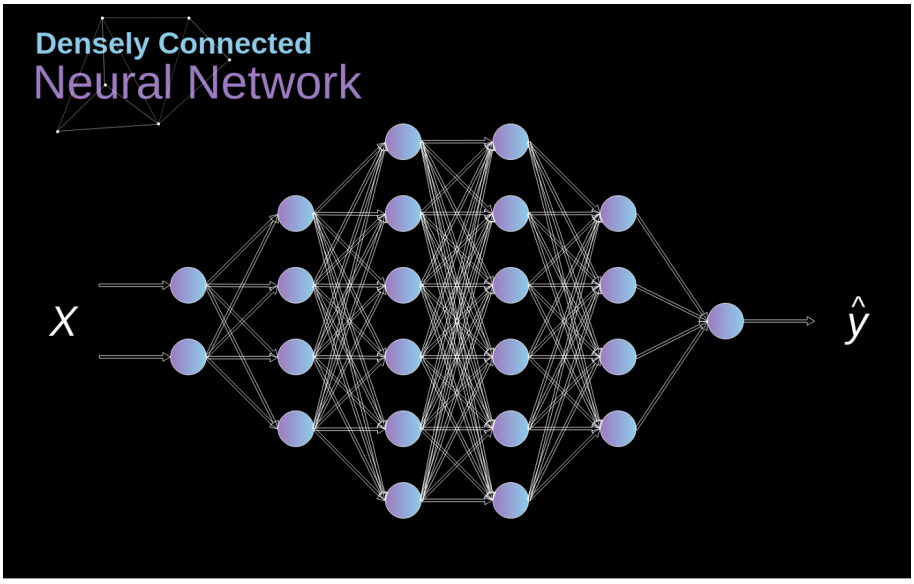


Figure 1. Densely connected neural network architecture

This approach works well when we solve classification problem based on a limited set of defined features — for example, we predict a football player's position based on the statistics he logs during games. However, the situation becomes more complicated when working with photos. Of course,

we could treat the brightness of each pixel as a separate feature and pass it on as an input to our dense network. Unfortunately, in order to make it work for a typical smartphone photo, our network would have to contain tens or even hundreds of millions of neurons. On the other hand, we could scale our photo down, but we would lose valuable information in the process. Right away we see that a traditional strategy does nothing for us—we need a new clever way to use as much data as possible, but at the same time reduce the number of necessary calculations and parameters. That's when CNNs comes into play.

Digital photo data structure

Let's start by taking a minute to explain how digital images are stored. Most of you probably realize that they are actually huge matrices of numbers. Each such number corresponds to the brightness of a single pixel. In the RGB model, the colour image is actually composed of three such matrices corresponding to three colour channels — red, green and blue. In black-and-white images we only need one matrix. Each of these matrices stores values from 0 to 255. This range is a compromise between the efficacy of storing information about the image (256 values fit perfectly in 1 byte) and the sensitivity of the human eye (we distinguish a limited number of shades of the same colour).

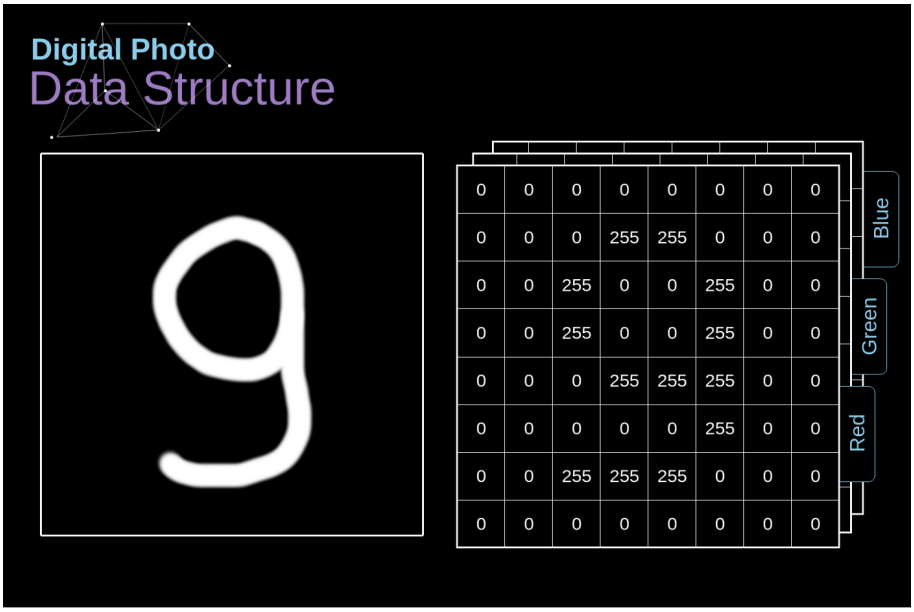


Figure 2. Data structure behind digital images

Convolution

Kernel convolution is not only used in CNNs, but is also a key element of many other Computer Vision algorithms. **It is a process where we take a small matrix of numbers (called kernel or filter), we pass it over our image and transform it based on the values from filter.** Subsequent feature map values are calculated according to the following formula, where the input image is denoted by f and our kernel by h. The indexes of rows and columns of the result matrix are marked with m and n respectively.

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

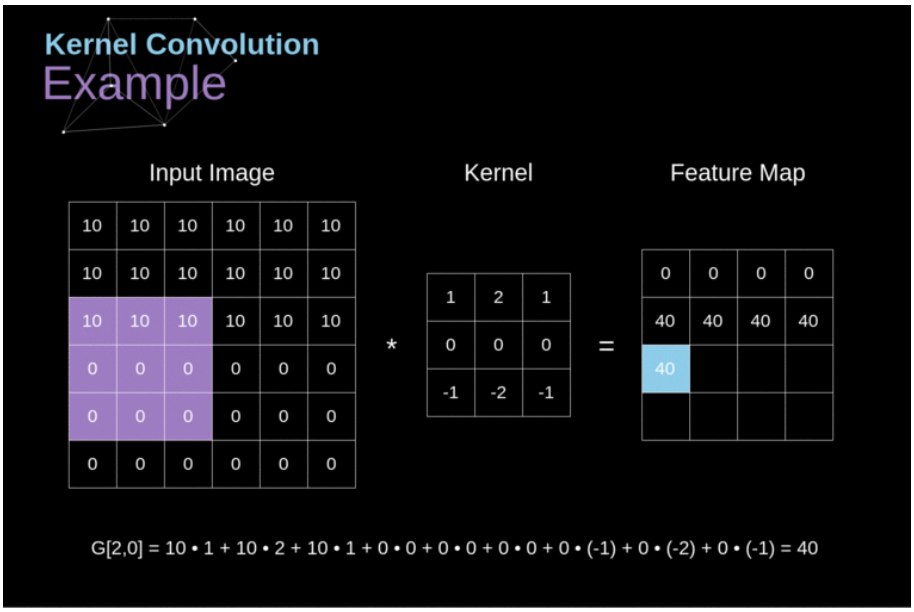


Figure 3. Kernel convolution example

After placing our filter over a selected pixel, we take each value from kernel and multiply them in pairs with corresponding values from the image. Finally we sum up everything and put the result in the right place in the output feature map. Above we can see how such an operation looks like in micro scale, but what is even more interesting, is what we can achieve by performing it on a full image. Figure 4 shows the results of the convolution with several different filters.

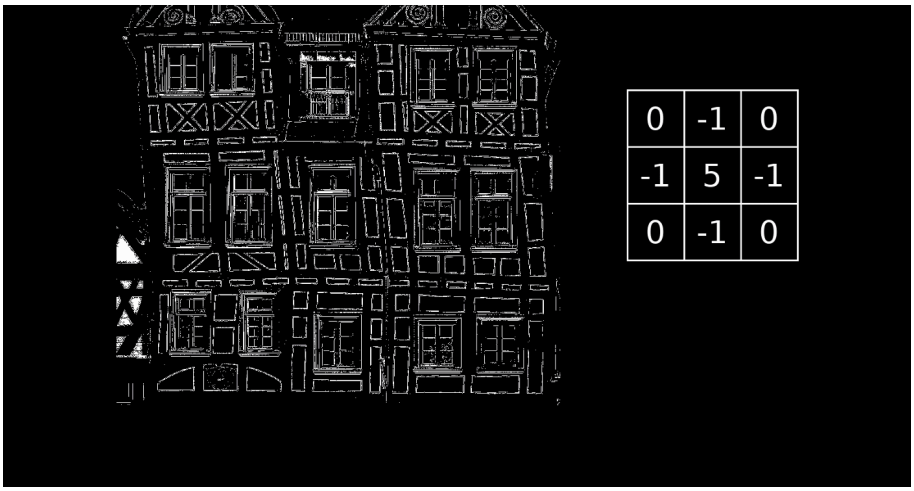


Figure 4. Finding edges with kernel convolution [Original Image]

Valid and Same Convolution

As we have seen in Figure 3, when we perform convolution over the 6x6 image with a 3x3 kernel, we get a 4x4 feature map. This is because there are only 16 unique positions where we can place our filter inside this picture. **Since our image shrinks every time we perform convolution, we can do it only a limited number of times, before our image disappears completely. What's more, if we look at how our kernel moves through the image we see that the impact of the pixels located on the outskirts is much smaller than those in the center of image.** This way we lose some of the information contained in the picture. Below you can see how the position of the pixel changes its influence on the feature map.

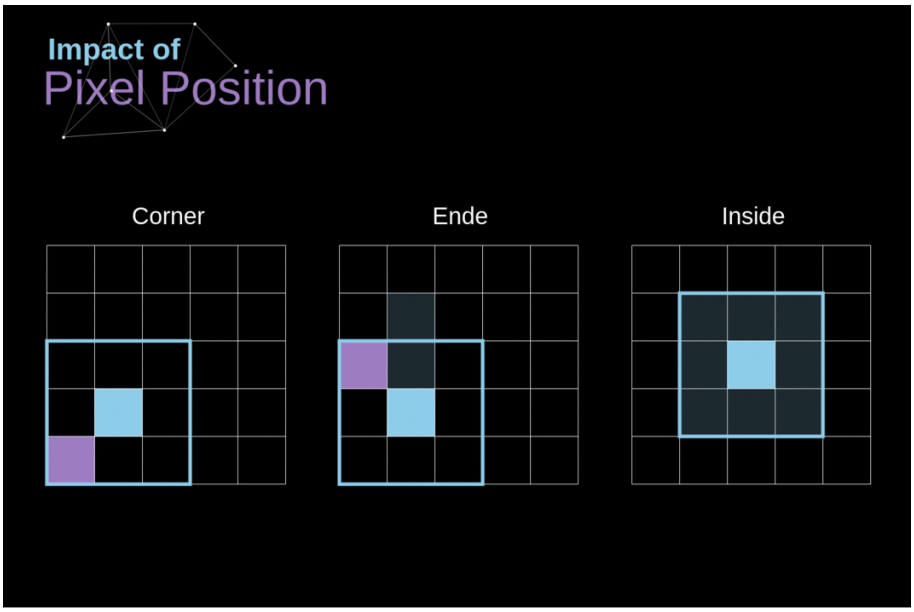


Figure 5. Impact of pixel position

To solve both of these problems we can pad our image with an additional border. For example, if we use 1px padding, we increase the size of our photo to 8x8, so that output of the convolution with the 3x3 filter will be 6x6. Usually in practice we fill in additional padding with zeroes. Depending on whether we use padding or not, we are dealing with two types of convolution — Valid and Same. Naming is quite unfortunate, so for the sake of clarity: **Valid — means that we use the original image, Same — we use the border around it**, so that the images at the input and output are the same size. In the second case, the padding width, should meet the following equation, where *p* is padding and *f* is the filter dimension (usually odd).

$$p = \frac{f - 1}{2}$$

Strided Convolution

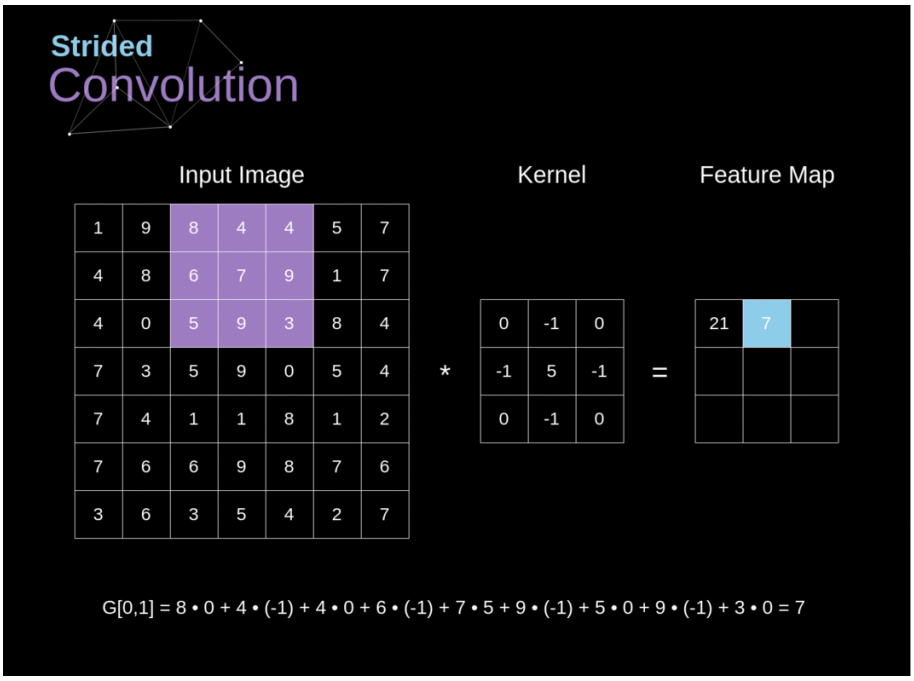


Figure 6. Example of strided convolution

In previous examples, we always shifted our kernel by one pixel. However, step length can also be treated as one of convolution layer hyperparameters. In Figure 6, we can see how the convolution looks like if we use larger stride. When designing our CNN architecture, we can decide to increase the step if we want the receptive fields to overlap less or if we want smaller spatial dimensions of our feature map. The dimensions of the output matrix - taking into account padding and stride - can be calculated

Top highlight

using the following formula.

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - f}{s} + 1 \right\rfloor$$

The transition to the third dimension

Convolution over volume is a very important concept, which will allow us not only to work with color images, but even more importantly to apply multiple filters within a single layer. **The first important rule is that the filter and the image you want to apply it to, must have the same number of channels.** Basically, we proceed very much like in the example from Figure 3, nevertheless this time we multiply the pairs of values from the three-dimensional space. **If we want use multiple filters on the same image, we carry out the convolution for each of them separately, stack the results one on top of the other and combine them into a whole.** The dimensions of the received tensor (as our 3D matrix can be called) meet the following equation, in which: *n* — image size, *f* — filter size, *nc* — number of channels in the image, *p* —used padding, *s* — used stride, *nf* — number of filters.

$$[n, n, n_c] * [f, f, n_c] = \left[\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, n_f \right]$$

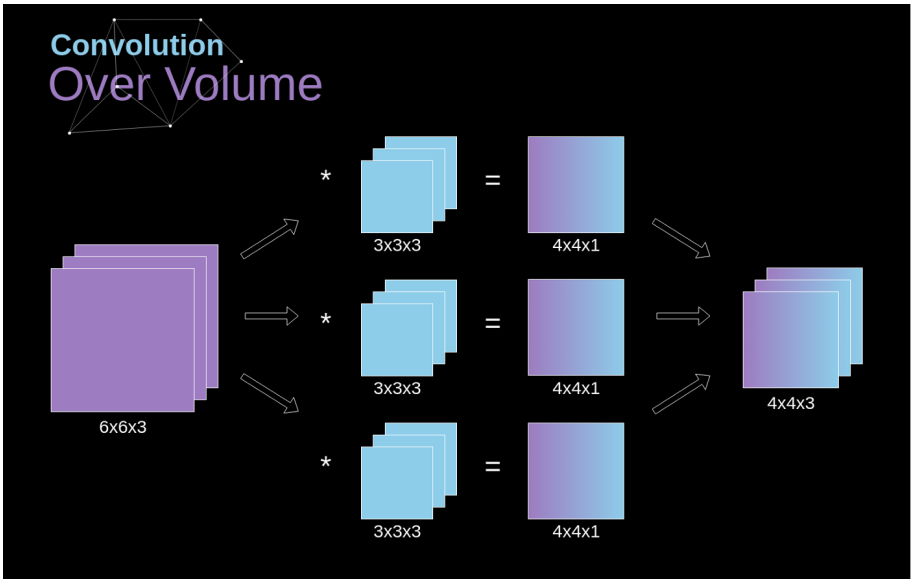


Figure 7. Convolution over volume

Convolution Layers

The time has finally come to use everything we have learned today and to build a single layer of our CNN. Our methodology is almost identical to the one we used for densely connected neural networks, the only difference is that instead of using a simple matrix multiplication, this time we will use the convolution. Forward propagation consists of two steps. The first one is to calculate the intermediate value **Z**, which is obtained as a result of the convolution of the input data from the previous layer with **W** tensor (containing filters), and then adding bias **b**. The second is the application of a non-linear activation function to our intermediate value (our activation is denoted by g). Fans of matrix equations will find appropriate mathematical formulas below. If any of the operations in question is not clear to you, I highly recommend my previous article, in which I discuss in detail what is happening inside densely connected neural networks. By the way, on illustration below you can see a small visualization, describing the dimensions of tensors used in equation.

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \qquad \mathbf{A}^{[l]} = g^{[l]}(\mathbf{Z}^{[l]})$$

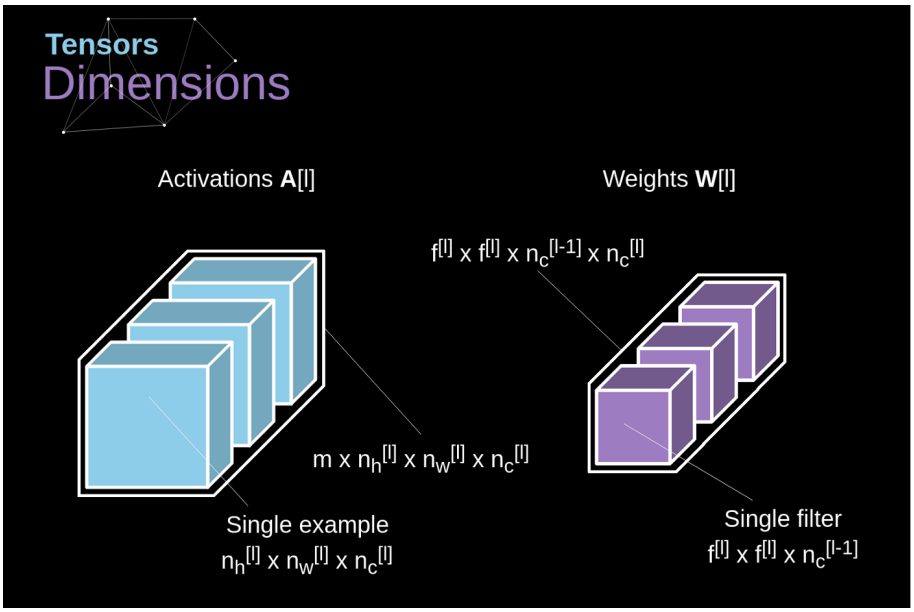


Figure 8. Tensors dimensions

Connections Cutting and Parameters Sharing

At the beginning of the article I mentioned that densely connected neural networks are poor at working with images, due to the huge number of parameters that would need to be learned. Now that we understand what convolution is all about, let's consider how it allows us to optimize the calculations. On the Figure below, the 2D convolution has been visualized in a slightly different way — neurons marked with numbers

1–9 form the input layer that receives brightness of subsequent pixels, while units A-D denotes calculated feature map elements. Last but not least, I-IV are the subsequent values from kernel — these must be learned.

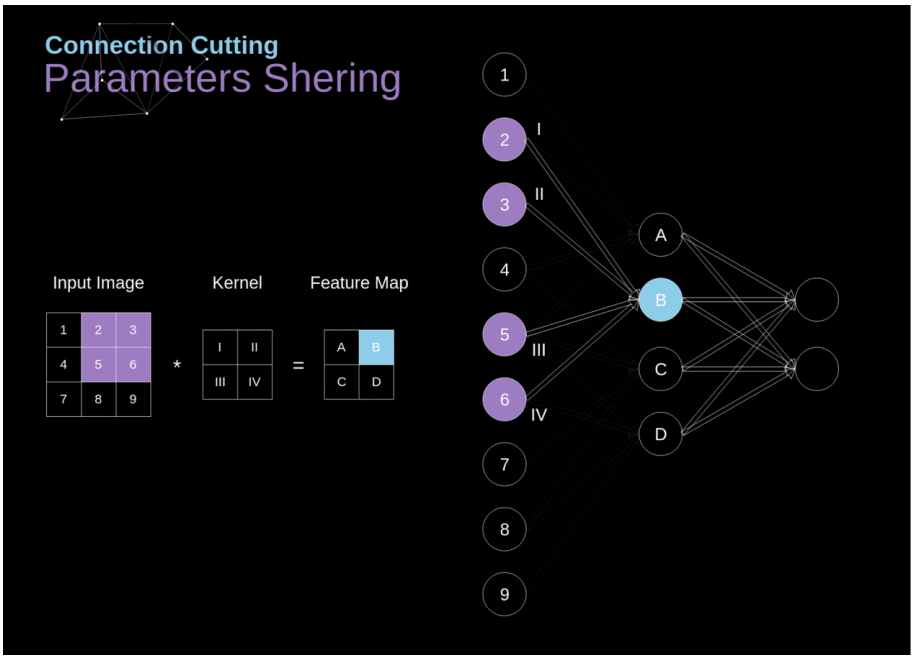


Figure 9. Connections cutting and parameters sharing

Now, let's focus on the two very important attributes of convolution layers. **First of all, you can see that not all neurons in the two consecutive layers are connected to each other.** For example, unit 1 only affects the value of A. **Secondly, we see that some neurons share the same weights.** Both of these properties mean that we have much less parameters to learn. By the way, it is worth noting that a single value from the filter affects every element of the feature map — it will be crucial in the context of backpropagation.

Convolutional Layer Backpropagation

Anyone who has ever tried to code their own neural network from scratch knows, that forward propagation is less than half the success. The real fun starts when you want to go back. Nowadays, we don't need to bother with backpropagation — deep learning frameworks do it for us, but I feel it's worth knowing what's going on under the hood. **Just like in densely connected neural networks, our goal is to calculate derivatives and later use them to update the values of our parameters in a process called gradient descent.**

In our calculations we will use a chain rule — which I mentioned in

previous articles. **We want to assess the influence of the change in the parameters on the resulting features map, and subsequently on the final result.** Before we start to go into the details, let us agree on the mathematical notation that we will use — in order to make my life easier, I will abandon the full notation of the partial derivative in favour of the shortened one visible below. But remember, that when I use this notation, I will always mean the partial derivative of the cost function.

$$\mathbf{dA}^{[l]} = \frac{\partial L}{\partial \mathbf{A}^{[l]}} \quad \mathbf{dZ}^{[l]} = \frac{\partial L}{\partial \mathbf{Z}^{[l]}} \quad \mathbf{dW}^{[l]} = \frac{\partial L}{\partial \mathbf{W}^{[l]}} \quad \mathbf{db}^{[l]} = \frac{\partial L}{\partial \mathbf{b}^{[l]}}$$

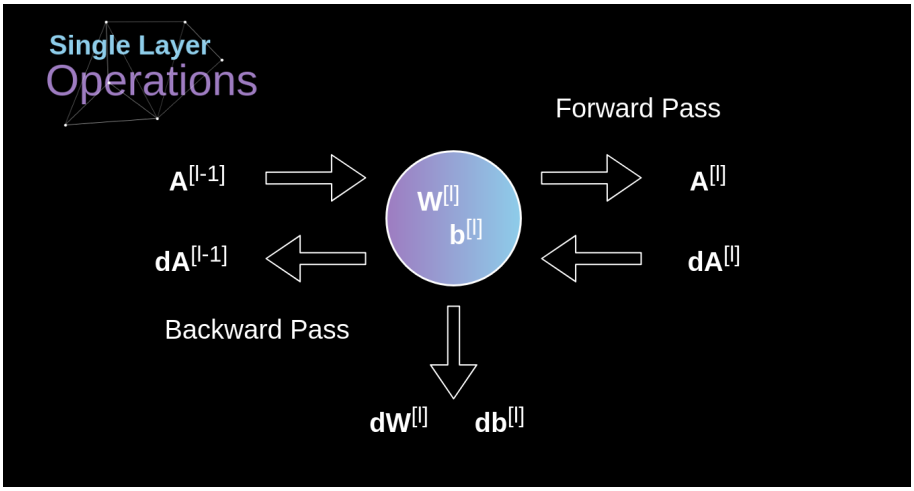


Figure 10. Input and output data for a single convolution layer in forward and backward propagation

Our task is to calculate **dW[l]** and **db[l] -** which are derivatives associated with parameters of current layer, as well as the value of **dA[l -1]** - which will be passed to the previous layer. As shown in Figure 10, we receive the **dA[l]** as the input. Of course, the dimensions of tensors **dW** and **W**, **db** and **b** as well as **dA** and **A** respectively are the same. The first step is to obtain the intermediate value **dZ[l]** by applying a derivative of our activation function to our input tensor. According to the chain rule, the result of this operation will be used later.

$$\mathbf{dZ}^{[l]} = \mathbf{dA}^{[l]} * g'(\mathbf{Z}^{[l]})$$

Now, we need to deal with backward propagation of the convolution itself, and in order to achieve this goal we will utilise a matrix operation called full convolution — which is visualised below. Note that during this process we use the kernel, which we previously rotated by 180 degrees. This operation can be described by the following formula, where the filter is denoted by **W**, and **dZ[m,n]** is a scalar that belongs to a partial derivative obtained from

the previous layer.

$$\mathbf{dA+} = \sum_{m=0}^{n_h} \sum_{n=0}^{n_w} \mathbf{W} \cdot \mathbf{dZ}[m, n]$$

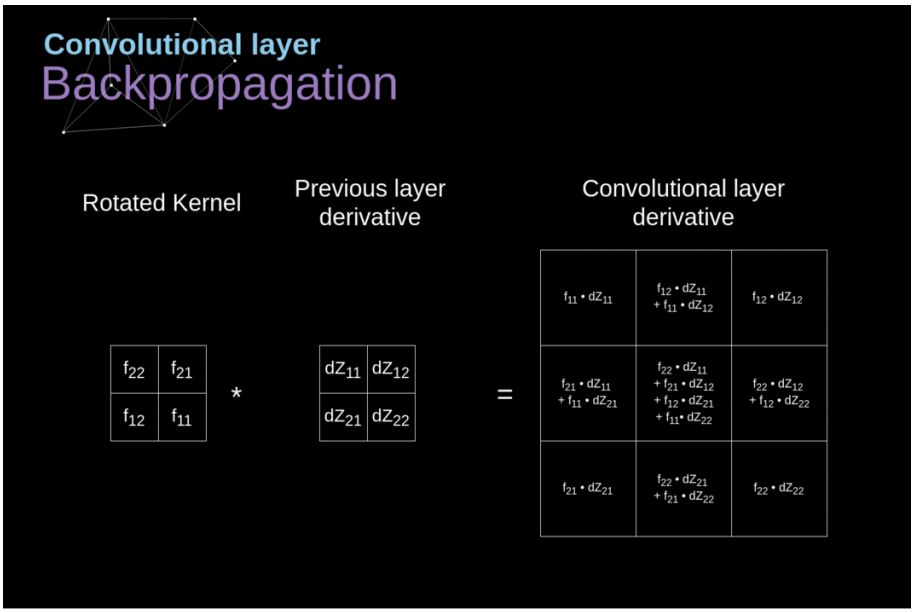
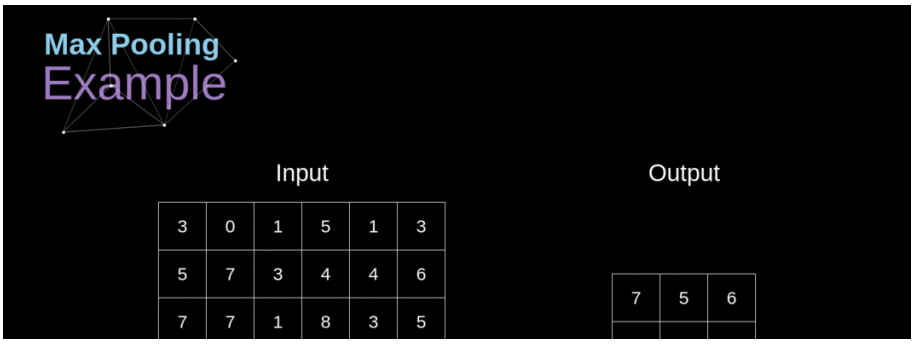


Figure 11. Full convolution

Pooling Layers

Besides convolution layers, CNNs very often use so-called pooling layers. They are used primarily to reduce the size of the tensor and speed up calculations. This layers are simple - we need to divide our image into different regions, and then perform some operation for each of those parts. For example, for the Max Pool Layer, we select a maximum value from each region and put it in the corresponding place in the output. As in the case of the convolution layer, we have two hyperparameters available — filter size and stride. Last but not least, if you are performing pooling for a multi-channel image, the pooling for each channel should be done separately.



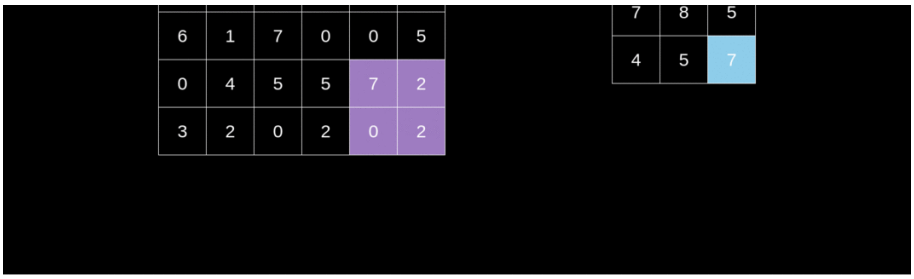


Figure 12. Max pooling example

Pooling Layers Backpropagation

In this article we will discuss only max pooling backpropagation, but the rules that we will learn — with minor adjustments — are applicable to all types of pooling layers. Since in layers of this type, we don't have any parameters that we would have to update, our task is only to distribute gradiwents appropriately. As we remember, in the forward propagation for max pooling, we select the maximum value from each region and transfer them to the next layer. It is therefore clear that during back propagation, the gradient should not affect elements of the matrix that were not included in the forward pass. In practice, this is achieved by creating a mask that remembers the position of the values used in the first phase, which we can later utilize to transfer the gradients.

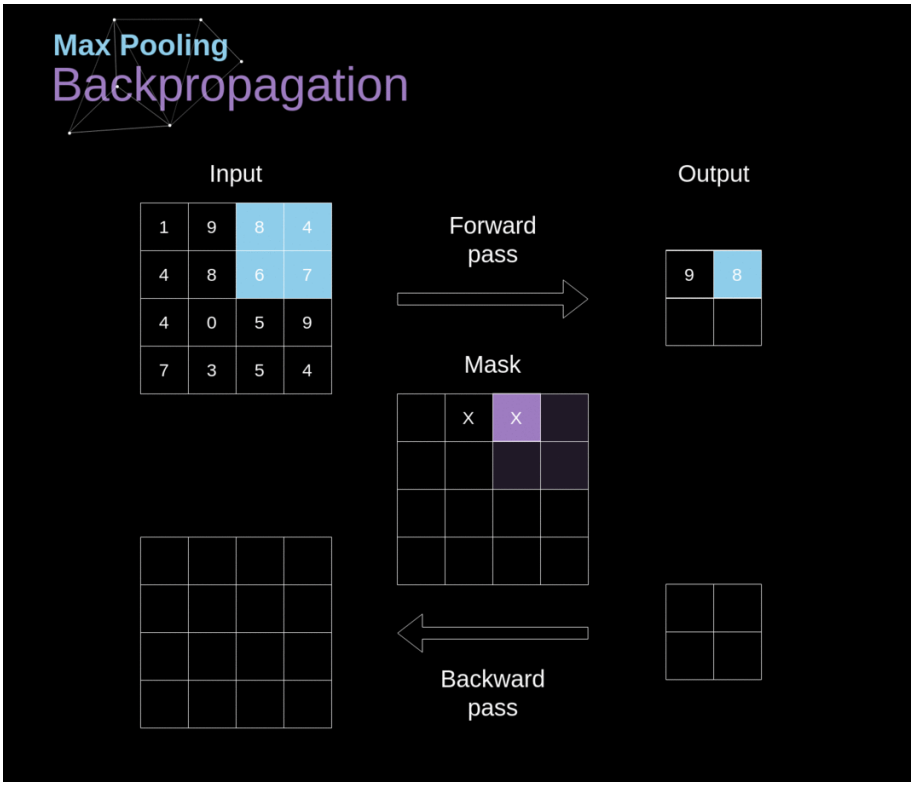


Figure 13. Max pooling backward pass

Conclusion

Congratulations if you managed to get here. Big thanks for the time spent reading this article. If you liked the post, consider sharing it with your friend, or two friends or five friends. If you have noticed any mistakes in the way of thinking, formulas, animations or code, please let me know.

This article is another part of the “Mysteries of Neural Networks” series, if you haven’t had the opportunity yet, read the other articles. Also, if you like my job so far, follow me on Twitter and Medium and see other projects I’m working on, on GitHub and Kaggle. Stay curious!

. . .