Introduction

In this paper we are going to give an introduction to the mechanics of *convolutional* neural networks. Our first obejective is to outline the most important concepts in machine learning. Building upon these concepts, we are going to show the underlying mathematical equations which govern the operation of a convolutional nerual network. Included in that section will be and answer as to why it is that convolutional neural networks are named so. Finally we show and implimentation of a convolutional neural network from scratch using nothing but Python with the NumPy library.

Table of contents

1 Machine learning concepts

First of all we need to answer the question 'what is a neural network?' from a conceptual level. Conceptualy, a nerual network is a large network of numerous 'neurons'. These neruons are connected to one another through so called 'channels' of data streams.

A neuron in a neural network is supposed to *look* for certain *features* in the input data it recives from upstream. These features that the neuron is looking for are those that it has been trained to look for. Neurons in a neural network are not told explicitly what features they are supposed to look for. If the neuron *thinks* that the data it is looking at possess the features it is looking for—it *fires* or *activates*. Which in simple terms means that it sends some new data downstream.

The structure of a neural network as a whole—is a layerd one. Neurons are grupped together in specific layers. Neruons in the same layer possess the same internal structure and thus process the same kinds of input data. Of course this also means that they produce the same kind of output data. The structure of neural networks bears some resemblance to actuall neural networks as studied in neuroscience. Figure 1 shows the overall structure of a nerual network. The figure also illustrates the common terminology of calling the layer inbetween the input and output layer for *hidden* layers.

In addition to layers of neurons, nerual networks can also incorporate special 'non-thinking' layers. Layers in which there are no neurons. These layers may serve a different purposes but a commonly, such layers are used to reduce the computational complexity of a nerual network. Furthermore, these layers may protect the network from the non-desirable phenomena of 'overfitting' data. Having too many parameters than what is justifible for the problem that the network is treating.

This leads us to the question: what is the kinds of problems that neural networks treat? A neural network's job is to process some input data and produce a final 'ativation'. This final activation represents what the nerural network 'thinks' about the input it has processed. This output of the network, its final activation, can be used for many different purposes such opening a door or turning the wheel of a self-driving car. Perhaps one of the simplest, and a very common application, of nerual networks ses them used to classify large amounts of data. In this context, the output of the network is interpreted as a probability. The probability represents how confident the network is that one piece of data belongs to a particular class. Networks used for classification purposes often output several probabilites which is interpreted as the networks confidence that a particular input data belongs to any one of the classes that the network knows about.

The convolutional neural network that we are going to be using as an example in this paper, serves such a classification purpose.

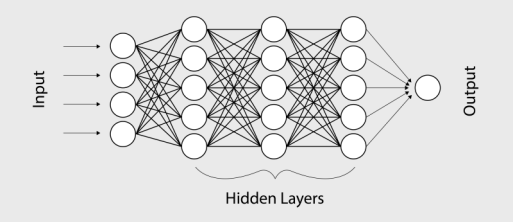

Figure 1 Overview of a nerual network structure [4]

1.1 Traning a neural network with a cost function

The major idea which is the reason why neural networks perform at so high levels is the idea of training them using a cost function. The cost function is a function that we insert at the end of a neural network. The cost function takes the output of the network as its input and, together with a predetermined set of correct values, it determines the score of the network. The score or *the cost* of a neural network is a single number that describes how well the network performed for a given input. The better the network performs the lower the cost will be.

We use cost functions to train neural network by giving them several thousand inputs and calculating the average cost of all these inputs. This average cost is then used to enhance the parameters of the network by adjusting them according to the networks performance.

The idea of the cost function lies at the heart of why neural networks work. But it is worth noting that it is also somewhat limiting. For us to have a cost function availiable, we need to have a dataset with not only suitable inputs for a neural network but also, correct values or *labels* for each of these inputs. More often than not, humans have to manually provide these labels which somewhat restricts what data we can train neural networks on.

The cost function is also callled the loss function. In this paper we use the term loss function to refer to the function which calculates the *cost* of a network for a single output. While the cost function we use to refer to the average of all the loss function outputs.

1.2 Optimization

When we train a neural network we calculate its loss for several thousand inputs, average it to get the cost and use it to enhance the network. The way in which we enhance the network with this information is with various optimization algorithms. Note that the cost function is nothing more than a real valued non-negative function. Which we can use the tools of analysis to minimize.

The most commonly used algorithm used to minimize nerual network cost functions is gradient descent (see Figure **2**). The gradient of the cost function tells us which of the parameters in the network the cost function is most sensitive to. This tells us which parameters of the network we should adjust—namely the most sensitive ones—in order to minimize the cost function as much as possible. In geometric terms, the gradient shows which direction in the space of the networks parameters that would cause the highets increase in the value of the cost function. Incidently, the opposite direction is the direction in this space which lowers the cost function as much as possible. This is why we calculate the gradient and then use its negative inverse to adjust the networks parameters.

The gradient descent algorithm consists, in large parts, of the following:

1. Calculating the cost of the network for a large amount of input data.
2. Adjusting the parameters of the network with the negative gradient of the cost function.
3. Repating steps **1** through **2** enough times until the cost of the network is sufficiently low.

An inherent limitation of the gradient descent algorithm is that the propoerties of the cost function's extrema are not known a priori. The cost function could have, and probably does have, local minima which the gradient descent algorithim will converge towards and then get stuck in. Luckily though, in practice it turns out that the local minima of the cost functions used to train neural networks are sufficiently low so as to not cause and problems.

In the context of training a neural network, we have yet to introduce a way to calculate this very important gradient of the cost function. To do so, we first need to determine a mathematical

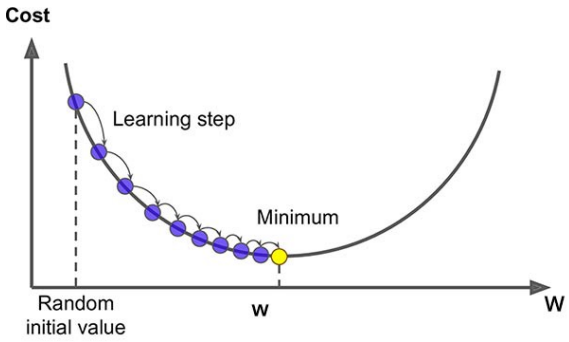

Figure 2 Visualization of gradient descent [5].

expression of the neural network itself. In fact, the network typically consists of more than just linear functions.

1.3 Forward and backward propoagation

Now that we have a clear concept of what we do to train neural networks, we need to undersand their two primary *modes* of operation.

Forward propagation refers to the regular operation of a neural network. The network process some input and returns some output which is then passed to a cost function.

Backward propagation refers to ensuing mode of operation. After the cost has been determined, the network now propagets backwards through itself in order to determine its gradient. It determines its gradient this way because of how the chain rule works. A neural network can be seen as a function with hundreds if not thousands of individual parameters. But the real observation, is that a nerual network can be seen as a *nested* function of functions. In such a hierarchy, each function represents the output of some layer that is then passed onto the next function in the hierarchy—the next layer. The chain rule tells us that in order to calculate the derivative of such a nested function it, roughly speaking, suffices to calculate the derivate of each function in the nesting and multiplying the results. Which can be thought of as *unwinding* the nested functions.

The unwinding starts at the end of the network with the function at the top of the nesting hierarchy. The next function to unwind is the one before the last one, which in other terms is the second to last layer of the neural network. This procedure of unwinding the networks nested function from back to front is why we call this operation backward propagation.

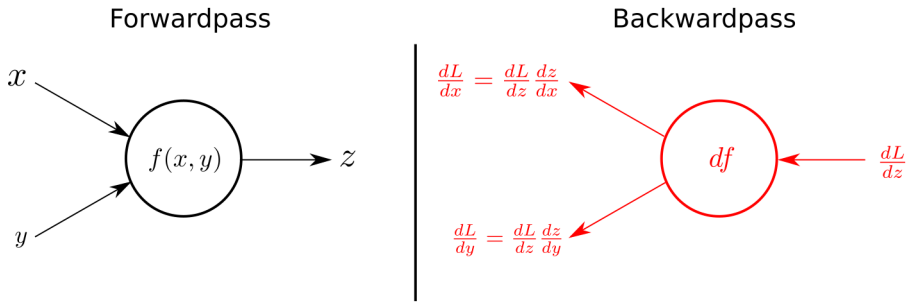

Figure 3 Forward propagation and backward propagation [6].

2 Layers of convolutional neural networks

In this section we are going to look at the fundamental equations which govern the operation of each layer of a convolutional neural network. The equations we are going to be looking at are primarily the 'lowest level' ones, relating each number from the previous layer to each number in the next layer.

Terminology and notation

Before we get started we need to introduce some terms and notation. Scalars will be represented with a lowercase letter (e.g. x). Vectors will be represented with a bold lowercase letter (e.g. $\boldsymbol{x}$). Matricies will be represented with an uppercase letter (e.g. X). Vectors of matricies—vectors where each entry is a matrix—will be represented with a bold uppercase letter (e.g. $\boldsymbol{X}$). This scheme has the added benefit that for the most part, aggregates of a layer will be represented in bold while data in individual channels will be represented in normal style letters.

Speaking of layers we need some symbols for the various parts that make up a layer. Firstly we will need symbols for the two principle components of a neuron. Namely its weights and its bias. Weights will be represented with the letter 'w' while the bias will be represented with the letter 'b'. The preliminary output of a neruon (or layer of neurons), before it is passed through an activation function, will be represented with the letter 'z'. The output after activation will be represented with the letter 'a'. Where does one layer end and the next one begin? Everything after the previous layer's activation up until and including the current layer's activation constitues the layer in question.

We need the ability to reference specific positions in each layer and for that we need *a lot* of indexes. Since we want our mathematics to match our code, to a certain extent, all the indices used in this paper will start at 0. The layer index which tells us which layer we are looking at will be written using a parenthetical superscript (e.g. $\boldsymbol{a}^{(l)}$ is the activation vector of layer l). The channel index which tells us which channel we are looking at will be written using a parenthetical subscript (e.g. $a^{(l)}_{(c)}$ is the scalar activation of layer l in channel c). The output of each neuron in a layer becomes a channel in the next layer. To reference both the input and output channel, a double parenthetical subscript will be used (e.g. $w^{(l)}_{(c,c')}$ is the scalar weight in layer l connecting channel c from layer l to channel c' in layer $l-1$). To reference positions within a matrix we will use regular xy-coordinates (e.g. $a(x,y)$ is the scalar in row x at column y). Equations which relate a previous layer to the next one will generally use blue color to makr indicies belonging to the previous layer. Red color will be used to mark indices beloning to the next layer.

We will often need to know the *dimensions* of layer. How many channels does it have and what are the sizes of the matrices in that layer. These dimensions will be written using the greek letter eta (η) with a layer index and a subscript to identify which dimention is expressed. A 'c' subscript means it is the number of channels in the layer *minus one*. A 'x' and 'y' subscript means it is row and column length *minus one* respectivly. We use the letter η instead of n to remind the reader that all dimensions are substracted 1, because we want all our indices to start at zero. As an example, if layer 3 has eight individual channels then $\eta^{(3)}_c = 7$. In addition to the η dimentions we are going to need the dimentions of various weight matricies (kernels). These will use the same kind of notation but with k instead of η.

The inital input to a network can be though of as the first activation to the network. So we are going to denote the inital input using the appropriate 'a' letter with a zero in parenthetical subscript.

2.1 Fully-connected layer

The fully-connected layer of a convolutional network is a classic neural network layer as seen in classical neural network. Which are also known as multi-layer precepetrons.

Recall the concept of how neurons in a layer fire and such, here is how this concept is implemented in the fully-connected layer. The layer takes as its input a multi-channel scalar stream of data. Each such stream is connected to each neuron in the layer. The neurons in the layer all have a vector of scalar weights equal in length to the number of input streams that the neurons have. In addition, each neuron has its own bias. The data in the input streams are multiplied with their corresponding weights in each neurons weight vector. Subsequently, the bias is added onto this weighted sum. This forms the preliminary output of the neurons (z) which is then sent through some activation function responsible for determining if the neuron fires or not. Figure 4 shows a diagram representing this kind of layer.

Since the data this layer is working with is scalar values there are no η_x or η_y to speak of. Furthermore, the number of output streams is determined by the number of neurons in the layer. Meaning that the number of channels in the previous layer is completely independent from the number of channels in the next layer.

Let l be the index of a fully-connected layer. The equation governing the forward operation for these kinds of layers is:

$$z_{(c)}^{(l)} = \sum_{c'=0}^{\eta_c^{(l-1)}} \left(w_{(c,c')}^{(l)} \, a_{(c')}^{(l-1)} \right) + b_{(c)}^{(l)} \tag{1}$$

The final output is obtained by sending z through the choosen activation function. Now that we know how to move forwards through a fully-connected layer we need to be able to propagate backwars through it as well. We can use equation 1 to calculate the equations governing the backward operation of these layers. The backpropagation through this layer starts at its output with a gradient vector. Recall that we denote the loss of the network by ℓ so we write this gradient vector as:

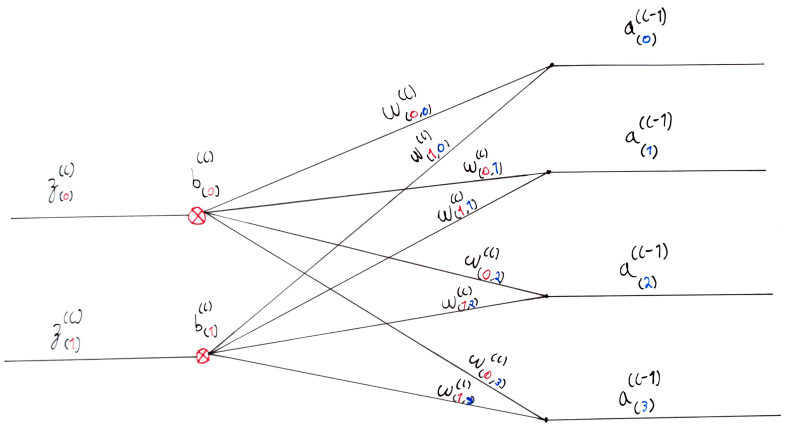

Figure 4 A diagram of a fully-connected layer.

$$\frac{\partial \ell}{\partial \boldsymbol{z}^{(l)}} = \begin{pmatrix} \frac{\partial \ell}{\partial z_0^{(l)}} \\ \frac{\partial \ell}{\partial z_1^{(l)}} \\ \vdots \\ \frac{\partial \ell}{\partial z_{\eta_c^{(l)}}^{(l)}} \end{pmatrix}$$

The partial derivative which relates each z to an a in the previous layer is:

$$\frac{\partial z_{(c)}^{(l)}}{\partial a_{(c'')}^{(l-1)}} = \frac{\partial}{\partial a_{(c'')}^{(l-1)}} \left(\sum_{c'=0}^{\eta_c^{(l-1)}} \left(w_{(c,c')}^{(l)} \, a_{(c')}^{(l-1)} \right) + b_{(c)}^{(l)} \right) = w_{(c,c'')}^{(l)}$$

Using the chain rule for partial derivatices we then obtain:

$$\frac{\partial \ell}{\partial a_{(c'')}^{(l-1)}} = \sum_{c=0}^{\eta_c^{(l)}} \frac{\partial \ell}{\partial z_{(c)}^{(l)}} \frac{\partial z_{(c)}^{(l)}}{\partial a_{(c'')}^{(l-1)}} = \sum_{c=0}^{\eta_c^{(l)}} \frac{\partial \ell}{\partial z_{(c)}^{(l)}} w_{(c,c'')}^{(l)} \tag{2}$$

We also need to calculate the partial derivaties with respect to all the weights and the bias in the layer.

$$\frac{\partial z_{(c)}^{(l)}}{\partial w_{(c,c'')}^{(l)}} = \frac{\partial}{\partial w_{(c,c'')}^{(l)}} \left(\sum_{c'=0}^{\eta_c^{(l-1)}} \left(w_{(c,c')}^{(l)} \, a_{(c')}^{(l-1)} \right) + b_{(c)}^{(l)} \right) = a_{(c'')}^{(l-1)}$$

$$\frac{\partial z_{(c)}^{(l)}}{\partial b_{(c)}^{(l)}} = \frac{\partial}{\partial b_{(c)}^{(l)}} \left(\sum_{c'=0}^{\eta_c^{(l-1)}} \left(w_{(c,c')}^{(l)} \, a_{(c')}^{(l-1)} \right) + b_{(c)}^{(l)} \right) = 1$$

Applying the chain rule again we get:

$$\frac{\partial \ell}{\partial w_{(c,c'')}^{(l)}} = \frac{\partial \ell}{\partial z_{(c)}^{(l)}} \frac{\partial z_{(c)}^{(l)}}{\partial w_{(c,c'')}^{(l)}} = a_{(c'')}^{(l-1)} \frac{\partial \ell}{\partial z_{(c)}^{(l)}} \tag{3}$$

$$\frac{\partial \ell}{\partial b_{(c)}^{(l)}} = \frac{\partial \ell}{\partial z_{(c)}^{(l)}} \frac{\partial z_{(c)}^{(l)}}{\partial b_{(c)}^{(l)}} = \frac{\partial \ell}{\partial z_{(c)}^{(l)}} \tag{4}$$

We can with the following scheme 'scale up' equation **1** to the 'layer level'.

$$\boldsymbol{z}^{(l)} = \begin{pmatrix} z_{(1)}^{(l)} \\ z_{(2)}^{(l)} \\ \vdots \\ z_{(\eta_c^{(l)})}^{(l)} \end{pmatrix}$$

$$W^{(l)} = \begin{pmatrix} w_{(0,0)}^{(l)} & w_{(0,1)}^{(l)} & \cdots & w_{(0,\eta_c^{(l-1)})}^{(l)} \\ w_{(1,0)}^{(l)} & w_{(1,1)}^{(l)} & \cdots & w_{(1,\eta_c^{(l-1)})}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{(\eta_c^{(l)},0)}^{(l)} & w_{(\eta_c^{(l)},1)}^{(l)} & \cdots & w_{(\eta_c^{(l)},\eta_c^{(l-1)})}^{(l)} \end{pmatrix}$$

$$\boldsymbol{a}^{(l-1)} = \begin{pmatrix} a^{(l-1)}_{(1)} \\ a^{(l-1)}_{(2)} \\ \vdots \\ a^{(l-1)}_{(\eta_c^{(l-1)})} \end{pmatrix}$$

$$\boldsymbol{b}^{(l)} = \begin{pmatrix} b^{(l)}_{(0)} \\ b^{(l)}_{(1)} \\ \vdots \\ b^{(l)}_{(\eta_c^{(l)})} \end{pmatrix}$$

Now we can write equation 1 as:

$$\boldsymbol{z}^{(l)} = W^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{5}$$

2.2 Convolutional layer

The convolutional layer is in many ways an extension of the regular fully-connected layer. One of the issues with regular fully-connected layers is that these kinds of layers only accept input that is in the form of a vector. This means that for applications where it is not natural for the input to be in a vector format, say image recognition, the input first has to be translated to a vector format. Usually this results in a loss of information contained in the input. In the typical case of image recongition, the input is in the form of one or more arrays of two dimensions. For a multilayer percepetron to treat this input, the images has to be 'flattend' into a vector of one dimention before it can be passed on to the network. This procedure eliminates some of the pixel relations in the image. To deduce this, consider the process of reconstructing a flattend image. If the image's dimensions prior to being flattend is not known, it is impossible, without the aid of pattern recognition, to reconstruct the image and be sure the reconstruction is equal to the original image.

To fix this problem, we can use a 'simple' solution. Instead of having the neuron contain a vector of weights, let it have an array of weights. These weight arrays are often called the neurons *kernel* or *filter*. If we change the neuron's vector of weights into an array, we also need to change the operation that is used to combine the weights with the input (which in a neuron of a multilayer percepetron is the dot product). There are two things to consider here. The purpose of the weights is to look for features or *patterns* in the input—by emphasizing or deemphasizing certain aspects—and the operation must reflect this purpose of the weights. Furthermore, the result of the operation should be a single number which, in a sense, representes the neurons 'initial' confidence that the feature it is looking for is present in the input. The operation which does both of these things is the *Hadamard product*. The Hadamard product can be seen as an extension of the dot product for two dimentional arrays. It combines two arrays—of the same dimensions—by multiplying corresponding entries together and summing the results. Which is precisly what the dot product does with two vectors.

Another thing we need to take into account is the dimentions of our new weight array. If we were to proceed analogically to how a multilayer percepetron works, the array should have the same size as the input to the neuron. 'Connecting' each value in the input to an individual weight in the neuron. But this approach means that each neuron, in principle, looks for a single feature in the entire input at once. A more refined approach, is to let the neuron's weight array be smaller in size than its input. Instead of applying the weight array (using the Hadamard product) on the entire

input at once, we apply it (still using the Hadamard product) to portions of the input seperately. Intuitivly, this means that the weight arrays are 'scanned' across the entire input image. Figure 5 illustrates this idea of the weight array scanning the input matrix to produce the output matrix.

A weight array that is smaller than the input matrix allows the neurons to be trained to look for a single small feature, such as a 'sharp edge' or a 'round corner'. Which in can look for in multiple areas of the input. As figure 5 clearly shows, the output of the neurons is no longer be a single number representing how 'initialy' confident the neuron is that a particular feature is present in the input ***as a whole***. Rather, the result becomes a *feature map*. Another two dimentional array which representes how 'initialy' confident the neuron is that a particular feature is present in ***specific locations*** of the input.

Of course each neuron must be able to handle more than one input data stream each with its own input data matrix. A typical example of a case where multiple streams are needed is the treatment of RGB images. An RGB image consists of three arrays of pixel values (numbers) that desribe how red, green and blue an image is in each pixel. Such an input would be split into three channels before being passed to a convolutional layer. The neurons handle multiple data streams by simple adding together the feature maps produces by each data stream. Figure 6 gives an overview of a convolutional layer. The channels are drawn as pipes to emhpasize that they contain matrix data rather than scalar values.

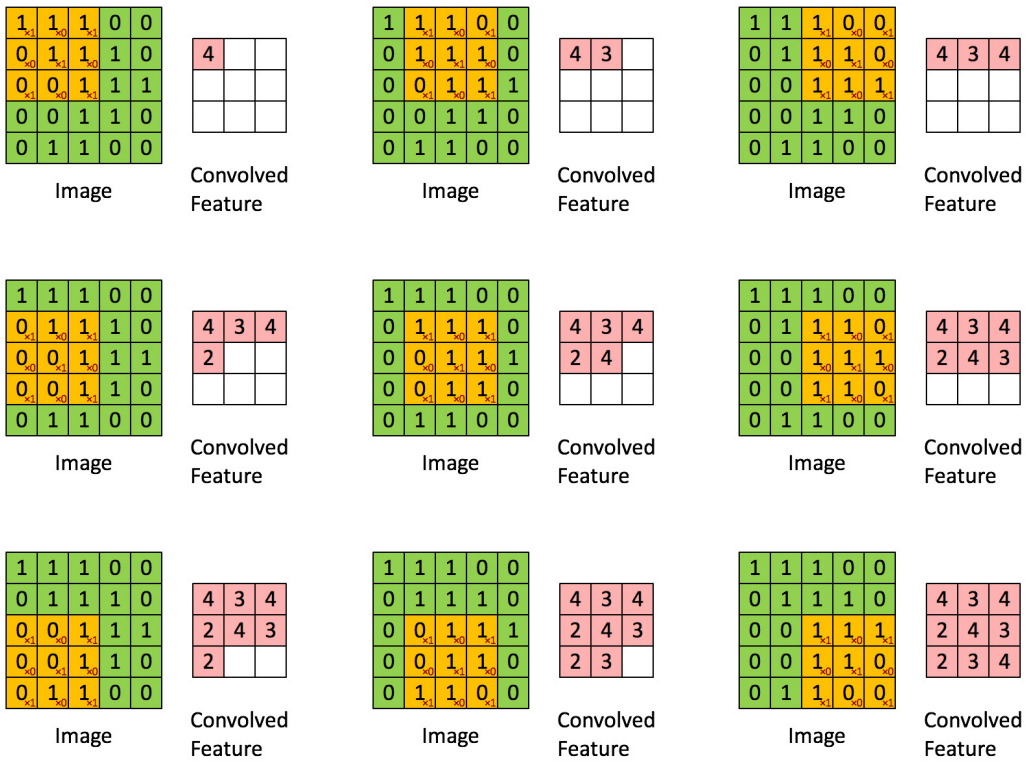

Figure 5 The basic forward operation of a convolutional layer [2].

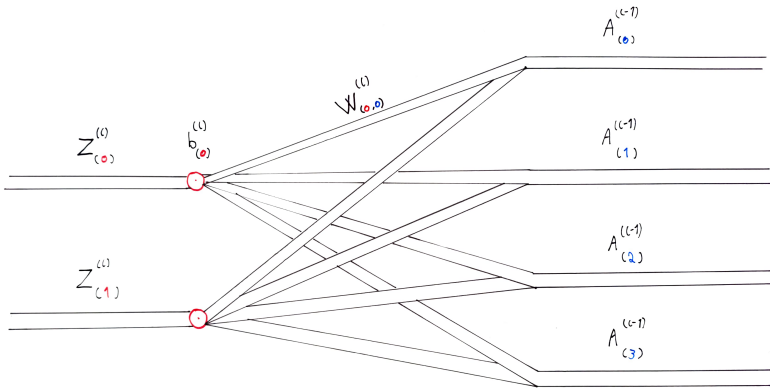

Figure 6 A diagram of a convolutional layer

2.2.1 Strides and zero padding

There are two other aspects of convolutional layers that we have yet to discuss: strides and zero padding. The stride is the number of rows/columns that the kernel moves over the input matrix between each scanning. Figure **5** shows a scenario where the stride is equal to one.

What purpose does altering the stride serve? Increasing the stride reduces the size of the resulting feature map since it is computed at fewer areas. Which has the net effect that the *computational complexity* of the neuron is reduced. However, it also means that the neuron studies its input less which can result in vital information in the input being overlooked. This is not a big problem as long as the features of the input is judged to be larger than the kernel that the neuron uses. So long as this is the case, increasing the stride of a neuron can significantly reduce its computational complexity without resulting in it no longer surving its purpose.

Let us now further develop Eq. (**22**) to take into account variable strides. It is as simple as multiplying the red indices with the strides.

$$F(x,y) = \text{activation}\left(\sum_y \sum_x \left(\sum_c \Big(K(x,y,c)\cdot M(xs_x + x, ys_y + y, c)\Big)\right) + b\right) \tag{6}$$

2.2.2 Zero padding

If we increase a neurons stride we can shrink the size of the resulting feature map. But it does not allow us to increase the size of the feature map! To do this, a techinque known as *zero padding* is used. Instead of working with the input *as is*, one or more extra outer rows and columns of zeros are added onto it before the kernel is scanned across it. This allows us to increase the size of the resulting feature map.

This techinque is predominantly used to make the feature map have the same dimentions as the input image in applicatons where this is desirable.

For the remainder of this section l is the index of a convolutional layer.

The dimentions of the input matricies and the kernel directly determine the dimentions of the output matricies.

$$\begin{aligned} \eta_x^{(l)} &= \eta_x^{(l-1)} - k_x^{(l)} + 1 \\ \eta_y^{(l)} &= \eta_y^{(l-1)} - k_y^{(l)} + 1 \end{aligned} \tag{7}$$

The equation governing the forward operation for this layer at the level of the individual neuron is:

$$z_{(c)}^{(l)}(x,y) = \sum_{c'}^{\eta_c^{(l-1)}} \left(\sum_{x'}^{k_x^{(l)}} \sum_{y'}^{k_y^{(l)}} \Big(w_{(c,c')}^{(l)}(x',y')\, a_{(c)}^{(l-1)}(x+x', x+y') \Big) \right) + b_{(c)}^{(l)} \tag{8}$$

This equation has so much detail in it that scaling it up to the layer level necessarly removes most of the information in it. But with it is possible with the following scheme which makes use of the intermediary h variables.

$$h^{(l)}_{(c,c')}(x,y) = \sum_{x'}^{k_x^{(l)}} \sum_{y'}^{k_y^{(l)}} \Big(w^{(l)}_{(c,c')}(x',y')\, a^{(l-1)}_{(c')}(x+x', x+y') \Big)$$

$$h^{(l)}_{(c)}(x,y) = \sum_{c'}^{\eta_c^{(l-1)}} h^{(l)}_{(c,c')}(x,y)$$

$$H^{(l)}_{(c)} = \begin{pmatrix} h^{(l)}_c(0,0) & h^{(l)}_c(0,1) & \dots & h^{(l)}_c(0,\eta^{(l)}_y) \\ h^{(l)}_c(1,0) & h^{(l)}_c(1,1) & \dots & h^{(l)}_c(1,\eta^{(l)}_y) \\ \vdots & \vdots & \ddots & \vdots \\ h^{(l)}_c(\eta^{(l)}_x,0) & h^{(l)}_c(\eta^{(l)}_x,1) & \dots & h^{(l)}_c(\eta^{(l)}_x,\eta^{(l)}_y) \end{pmatrix}$$

$$Z^{(l)}_{(c)} = \begin{pmatrix} z^{(l)}_c(0,0) & z^{(l)}_c(0,1) & \dots & z^{(l)}_c(0,\eta^{(l)}_y) \\ z^{(l)}_c(1,0) & z^{(l)}_c(1,1) & \dots & z^{(l)}_c(1,\eta^{(l)}_y) \\ \vdots & \vdots & \ddots & \vdots \\ z^{(l)}_c(\eta^{(l)}_x,0) & z^{(l)}_c(\eta^{(l)}_x,1) & \dots & z^{(l)}_c(\eta^{(l)}_x,\eta^{(l)}_y) \end{pmatrix}$$

$$B^{(l)}_{(c)} = \begin{pmatrix} b^{(l)}_c & b^{(l)}_c & \dots & b^{(l)}_c \\ b^{(l)}_c & b^{(l)}_c & \dots & b^{(l)}_c \\ \vdots & \vdots & \ddots & \vdots \\ b^{(l)}_c & b^{(l)}_c & \dots & b^{(l)}_c \end{pmatrix}$$

$$\boldsymbol{H}^{(l)} = \begin{pmatrix} H^{(l)}_{(0)} \\ H^{(l)}_{(1)} \\ \dots \\ H^{(l)}_{(\eta^{(l)}_c)} \end{pmatrix}$$

$$\boldsymbol{Z}^{(l)} = \begin{pmatrix} Z^{(l)}_{(0)} \\ Z^{(l)}_{(1)} \\ \dots \\ Z^{(l)}_{(\eta^{(l)}_c)} \end{pmatrix}$$

$$\boldsymbol{B}^{(l)} = \begin{pmatrix} B^{(l)}_{(0)} \\ B^{(l)}_{(1)} \\ \dots \\ B^{(l)}_{(\eta^{(l)}_c)} \end{pmatrix}$$

With this scheme we can write equation **8** as:

$$\boldsymbol{Z}^{(l)} = \boldsymbol{H}^{(l)} + \boldsymbol{B}^{(l)} \tag{9}$$

Now that we know how to move forwards through a convolutional layer we need to be able to propagate backwars through it as well. The backpropagation through this layer starts at its output with a gradient vector of matricies. We are not going to try and write this vector of matricies down directly, it suffices to note that we have access to all partial derivatives of the form:

$$\frac{\partial \ell}{\partial z^{(l)}_{(c)}(x,y)}$$

Using equation 8 we can calculate the equations governing backpropagation.

$$\begin{aligned}
\frac{\partial z^{(l)}_{(c)}(x,y)}{\partial a^{(l-1)}_{(c'')}(x'',y'')} &= \frac{\partial}{\partial a^{(l-1)}_{(c'')}(x'',y'')}\left(\sum_{c'=0}^{\eta_c^{(l-1)}}\left(\sum_{x'=0}^{k_x^{(l)}}\sum_{y'=0}^{k_y^{(l)}}\left(w^{(l)}_{(c,c')}(x',y')\,a^{(l-1)}_{(c')}(x+x',x+y')\right)\right)+b^{(l)}_{(c)}\right)\\
&= \frac{\partial}{\partial a^{(l-1)}_{(c'')}(x'',y'')}\left(\sum_{x'=0}^{k_x^{(l)}}\sum_{y'=0}^{k_y^{(l)}}\left(w^{(l)}_{(c,c'')}(x',y')\,a^{(l-1)}_{(c'')}(x+x',x+y')\right)\right)\\
&= w^{(l)}_{(c,c'')}(x''-x,y''-y)
\end{aligned}$$

With the chain rule we then obtain:

$$\frac{\partial \ell}{\partial a^{(l-1)}_{(c'')}(x'',y'')} = \sum_{c=0}^{\eta_c^{(l)}}\sum_{x=0}^{\eta_x^{(l)}}\sum_{y=0}^{\eta_y^{(l)}}\frac{\partial \ell}{\partial z^{(l)}_{(c)}(x,y)}\frac{\partial z^{(l)}_{(c)}(x,y)}{\partial a^{(l-1)}_{(c'')}(x'',y'')} \tag{10}$$

Equation 8 also lets us calculate the partial derivaties with respect to all the weights and the bias in the layer.

$$\begin{aligned}
\frac{\partial z^{(l)}_{(c)}(x,y)}{\partial w^{(l)}_{(c,c'')}(x'',y'')} &= \frac{\partial}{\partial w^{(l)}_{(c,c'')}(x'',y'')}\left(\sum_{c'=0}^{\eta_c^{(l-1)}}\left(\sum_{x'=0}^{k_x^{(l)}}\sum_{y'=0}^{k_y^{(l)}}\left(w^{(l)}_{(c,c')}(x',y')\,a^{(l-1)}_{(c')}(x+x',x+y')\right)\right)+b^{(l)}_{(c)}\right)\\
&= \frac{\partial}{\partial w^{(l)}_{(c,c'')}(x'',y'')}\left(\sum_{x'=0}^{k_x^{(l)}}\sum_{y'=0}^{k_y^{(l)}}\left(w^{(l)}_{(c,c'')}(x',y')\,a^{(l-1)}_{(c'')}(x+x',x+y')\right)\right)\\
&= a^{(l-1)}_{(c'')}(x+x'',x+y'')
\end{aligned}$$

With the chain rule we then obtain:

$$\frac{\partial \ell}{\partial w^{(l)}_{(c,c'')}(x'',y'')} = \sum_{x=0}^{\eta_x^{(l)}}\sum_{y=0}^{\eta_y^{(l)}}\frac{\partial \ell}{\partial z^{(l)}_{(c)}(x,y)}\frac{\partial z^{(l)}_{(c)}(x,y)}{\partial w^{(l)}_{(c,c'')}(x'',y'')} \tag{11}$$

Repating the same calculation for the bias.

$$\begin{aligned}
\frac{\partial z^{(l)}_{(c)}(x,y)}{\partial b^{(l)}_{(c)}} &= \frac{\partial}{\partial b^{(l)}_{(c)}}\left(\sum_{c'=0}^{\eta_c^{(l-1)}}\left(\sum_{x'=0}^{k_x^{(l)}}\sum_{y'=0}^{k_y^{(l)}}\left(w^{(l)}_{(c,c')}(x',y')\,a^{(l-1)}_{(c')}(x+x',x+y')\right)\right)+b^{(l)}_{(c)}\right)\\
&= 1
\end{aligned}$$

With the chain rule we then obtain:

$$\frac{\partial \ell}{b^{(l)}_{(c)}} = \frac{\partial \ell}{\partial z^{(l)}_{(c)}(x,y)} \tag{12}$$

A convolutional neural network is an evolution of a classical multilayer percepetron network. Recall the basic principle underpinning how a normal neuron in a neural network is supposed to work. The neuron is supposed to 'look' for features in its input data. If the neuron 'thinks' that those features are present in the input data it 'fires'. Otherwise the neuron does not fire.

In a classical multilayer percepetron network this is implemented in the following way. Each neuron contains a vector of weights, a bias and an activation function. The input to the neuron—which must be a vector of equal length to the neuron's own weight vector—is combined with the neuron's weight vector using the dot product. The neuron's bias is added onto the result which in turn is passed to the activation function which finally determines if the neuron 'fires' or not.

Using several layers of neurons one can achive quite remarkable results using this implementation of a neural network. However, a multilayer percepetron is inherently limited. The major problem is that neurons in these kinds of networks only accept input that is in the form of a vector. This means that for applications where it is not natural for the input to be in a vector format, say image recognition, the input first has to be translated to a vector format. Usually this results in a loss of information contained in the input. In the typical case of image recongition, the input is in the form of one or more arrays of two dimentions. For a multilayer percepetron to treat this input, the images has to be 'flattend' into a vector of one dimention before it can be passed on to the network. This procedure eliminates some of the pixel relations in the image. To deduce this, consider the process of reconstructing a flattend image. If the image's dimentions prior to being flattend is not known, it is impossible, without the aid of pattern recognition, to reconstruct the image and be sure the reconstruction is equal to the original image.

To fix this problem, we can use a 'simple' solution. Instead of having the neuron contain a vector of weights, let it have an array of weights. If we change the neuron's vector of weights into an array, we also need to change the operation that is used to combine the weights with the input (which in a neuron of a multilayer percepetron is the dot product). There are two things to consider here. The purpose of the weights is to look for features or *patterns* in the input—by emphasizing or deemphasizing certain aspects—and the operation must reflect this purpose of the weights. Furthermore, the result of the operation should be a single number which, in a sense, represents the neurons 'initial' confidence that the feature it is looking for is present in the input. The operation which does both of these things is the *Hadamard product*. The Hadamard product can be viewed as an extension of the dot product to two dimentional arrays. It combines two arrays—of the same dimentions—by multiplying corresponding entries together and summing the results. Which is precisly what the dot product does with two vectors.

The Hadamard product (Hp) of two matricies A, B with entries $a_{i,j}, b_{i,j}$ of equal dimentions is

$$\mathrm{Hp}(A, B) = \sum_j \sum_i a_{i,j} \cdot b_{i,j} \tag{13}$$

Another thing we need to take into account is the dimentions of our new weight array. If we were to proceed analogically to how a multilayer percepetron works, the array should have the same size as the input to the neuron. 'Connecting' each value in the input to an individual weight in the neuron. But this approach means that each neuron, in principle, looks for a single feature in the entire input at once. A more refined approach, is to let the neuron's weight array be smaller in size than its input. Instead of applying the weight array (using the Hadamard product) to the entire input at once, we apply it (still using the Hadamard product) to portions of the input seperately. Intuitivly, this means that the weight arrays are 'scanned' across the entire input image. This allows the neuron to be trained to look for a single feature, such as a 'sharp edge' or a 'round corner', in multiple areas of the input. The result of this method will no longer be a single number representing how 'initialy' confident the neuron is that a particular feature is present in the input ***as a whole***. Rather, the result becomes a *feature map*. Another two dimentional array which represents how 'initialy' confident the neuron is that a particular feature is present in ***specific locations*** of the input.

Going one step further, we can allow the neuron to treat inputs of not only a single two dimentional array, but several two dimentional arrays. A typical example where the input would

consists of several interlinked two dimentional arrays is RGB images. An RGB image consists of three arrays of pixel values (numbers) that desribe how red, green and blue an image is in each pixel. The number of interlinked two dimentional arrays present in the input, is known as the number of *channels* that the input has. In order for our neuron to treat inputs with more than one channel we let the neuron have as many channels as the input. That is to say, we equip the neuron with a weight array for each channel in the input. For each channel the weight arrays are applied to the input (using Hadamard) and the result in each channel is combined to form a final single feature map.

A neural network which makes use of layers of neurons of this kind, is a convolutional neural network. The multi-channel two dimentional arrays of weights inside each such neuron is called the neuron's *kernel* or *filter*. Why are these neural networks called convolutional neural networks? That will be explained in the next section.

2.3 The convolutional neuron

Let us start this section with a simple convolutional neuron. The neuron's kernel consists of a single weight matrix (one channel), some bias and some activation function. Consequently, the input that this neuron accepts is any one channel two dimentional array of size greater than its kernel.

The process of calcuating the neurons 'initial' confidence that the feature it is looking for is present in the input, is illustrated by Figure **5** on page **10**. Figure **5** shows how the kernel 'scans' the entire input array to compute the feature map. Let us construct the general formula for this feature map.

We will denote the feature map, kernel and input as F, K and M respectivly. They are two dimentional arrays and their individual entries will be denoted as $F(x, y)$ where x is the column index and y the row index. The indexes will all start at zero (e.g. $M(1, 2)$ is the entry in the second column, third row of M). There are many lengths involved in working with three seperate arrays, various lengths will here be denoted by n. A subscript, either x or y, will indicate if it is a horisontal or vertical length and a subsequent argument will indicate which array the length belongs to. So for example, $n_x(M)$ is the horisontal length of the input array—the number of columns or x's in M.

If the input represents a typical MNIST image with dimentions 28x28, then $n_x(M) = 28$ and $n_y(M) = 28$. In this scenario, the indices of M range in $[0\,..\,n_x(M) - 1] \times [0\,..\,n_y(M) - 1]$.

Let us say that the kernel scans its input one column and one row at a time. In this scenario, the output feature map will have dimentions

2.4 The convolutional layer

Okay this is where I will start developing the mathematics. First we need to define the various variables used and some terminology. A network has several layers and each layer has its own specific attributes. A supersricpt encolsed in parantheses will be used as the layer index.

$$\begin{aligned} \delta x_Z^{(l+1)} &= \delta x_W^{(l)} - \delta x_Z^{(l)} + 1 \\ y_Z^{(l+1)} &= \delta y_W^{(l)} - \delta y_Z^{(l)} + 1 \end{aligned} \tag{14}$$

$$\begin{aligned} x &\in [0\,..\,n_x(F) - 1] \qquad y \in [0\,..\,n_y(F) - 1] \\ x &\in [0\,..\,n_x(K) - 1] \qquad y \in [0\,..\,n_y(K) - 1] \end{aligned} \tag{15}$$

The product in layer l before activation, here denoted Z is given by the following formula Regular formula

$$Z^{(l+1)}_{x,y,c} = \sum_{c'=0}^{\delta c_Z^{(l)}} \sum_{y'=0}^{\delta y_Z^{(l)}} \sum_{x'=0}^{\delta x_Z^{(l)}} \left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \tag{16}$$

Derivative with respect to Z. $c^* \in [c \,..\, c + \delta c_W^{(l)}]$. $y^* \in [y \,..\, y + \delta y_W^{(l)}]$. $x^* \in [x \,..\, x + \delta x_W^{(l)}]$.

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{x,y,c}}{\partial Z^{(l)}_{x^*,y^*,c^*}} &= \frac{\partial}{\partial Z^{(l)}_{x^*,y^*,c^*}} \left(\sum_{c'=0}^{\delta c_W^{(l)}} \sum_{y'=0}^{\delta y_W^{(l)}} \sum_{x'=0}^{\delta x_W^{(l)}} \left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\delta c_W^{(l)}} \sum_{y'=0}^{\delta y_W^{(l)}} \sum_{x'=0}^{\delta x_W^{(l)}} \frac{\partial}{\partial Z^{(l)}_{x^*,y^*,c^*}} \left(\left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \right) \\ &= W^{(l,c)}_{x',y',c'} \end{aligned} \tag{17}$$

Derivative with respecto to W. $c^* \in [0 \,..\, \delta c_W^{(l)}]$. $y^* \in [0 \,..\, \delta y_W^{(l)}]$. $x^* \in [0 \,..\, \delta x_W^{(l)}]$.

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{x,y,c}}{\partial W^{(l,c)}_{x^*,y^*,c^*}} &= \frac{\partial}{\partial W^{(l,c)}_{x^*,y^*,c^*}} \left(\sum_{c'=0}^{\delta c_W^{(l)}} \sum_{y'=0}^{\delta y_W^{(l)}} \sum_{x'=0}^{\delta x_W^{(l)}} \left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \right) \\ &= \sum_{c'=0}^{\delta c_W^{(l)}} \sum_{y'=0}^{\delta y_W^{(l)}} \sum_{x'=0}^{\delta x_W^{(l)}} \frac{\partial}{\partial W^{(l,c)}_{x^*,y^*,c^*}} \left(\left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \right) \\ &= Z^{(l)}_{x^*+x',y^*+y',c^*} \end{aligned} \tag{18}$$

Derivative with respect to b

$$\begin{aligned} \frac{\partial Z^{(l+1)}_{x,y,c}}{\partial b^{(l,c)}} &= \frac{\partial}{\partial b^{(l,c)}} \left(\sum_{c'=0}^{\delta c_W^{(l)}} \sum_{y'=0}^{\delta y_W^{(l)}} \sum_{x'=0}^{\delta x_W^{(l)}} \left(W^{(l,c)}_{x',y',c'} Z^{(l)}_{x+x',y+y',c'} \right) + b^{(l,c)} \right) \\ &= 1 \end{aligned} \tag{19}$$

If we add a bias b to this filter, the formula becomes

$$F_*(x,y) = \sum_y \sum_x \Big(K(x,y) \cdot M(x+x, y+y) + b \Big) \tag{20}$$

Sending this formula throug an activation function such as ReLU or the sigmoid function, we obtain the final output feature.

$$F_*(x,y) = \text{activation} \left(\sum_y \sum_x \Big(K(x,y) \cdot M(x+x, y+y) + b \Big) \right) \tag{21}$$

2.4.1 Multiple channels

Let us return to Eq. (**21**) for a minute and ask: what happens if we add some channels to the input? Let us denote the number of channels by n_c. The input and kernel are now multi-channel

arrays, their individual entries will be denoted as $K(i, j, c)$ with c ranging in $[0\,..\,n_c-1]$ To calculate the feature map when multiple channels are present, we have to take into account the contribution of each individual channel. We do this by simply adding them together. Since we only want to add the bias of the filter once per entry in the feature map, the resulting modified formula becomes

$$F(x, y) = \text{activation}\left(\sum_y \sum_x \left(\sum_c \Big(K(x, y, c) \cdot M(x + x, y + y, c)\Big)\right) + b\right) \tag{22}$$

2.4.2 Where is the convolution?

In mathematics, the convolution operation is denoted by $*$ and the convolution s (itself a function) of two functions k and m is defined as

$$s(t) = (k * m)(t) = \int_{-\infty}^{+\infty} k(x) \cdot m(t - x)\,\mathrm{d}x \tag{23}$$

$$s(x) = (k * m)(x) = \sum_{x=-\infty}^{+\infty} k(x) \cdot m(x - x) \tag{24}$$

For continuous (Eq. (**23**)) and discrete (Eq. (**24**)) functions respectivly. On a computer, we are in practice always working with discrete convolutions so for us the convolution of interest is Eq. (**24**). If we make the functions k and m two dimentional—they take two arguments as their input—their convolution also becomes two dimentional.

$$s(x, y) = (k * m)(x, y) = \sum_{y=-\infty}^{+\infty} \sum_{x=-\infty}^{+\infty} k(x, y) \cdot m(x - x, y - y) \tag{25}$$

Let us known say that k and m are two functions which each index an array of two dimentions. Meaning that the functions take two indexes as their input arguments. Let us further assume that k and m's arrays are zero at any index but for those contained in a small area. For k that area is $[0\,..\,n_x(k) - 1] \times [0\,..\,n_y(k) - 1]$ and for m it is $[0\,..\,n_x(m) - 1] \times [0\,..\,n_y(m) - 1]$. In this case, the convolution of k and m reduces to

$$s(x, y) = (k * m)(x, y) = \sum_{y=0}^{n_y(k)-1} \sum_{x=0}^{n_x(k)-1} k(x, y) \cdot m(x - x, y - y) \tag{26}$$

Where $x \in [0\,..\,n_x(m) - 1]$ and $y \in [0\,..\,n_y(m) - 1]$.

The resemblance to Eq. (**16**) is already apparent but there is a subtle difference. Eq. (**26**) subtracts the blue indices from the red ones while Eq. (**16**) adds them together. The effect of subtracting instead of adding renders the convolution operation $(*)$ commutative.

$$\begin{gathered}(k * m)(x, y) = (m * k)(x, y) \\ \sum_x \sum_y k(x, y) \cdot m(x - x, y - y) = \sum_x \sum_y m(x, y) \cdot k(x - x, y - y)\end{gathered} \tag{27}$$

So what is the effect of replacing the pluses in Eq. (**16**) with minuses? Well at first glance it seems that you cannot do it as you would end up with indexes that are out of bounds for the input image M. But let us assume that M behaves as m meaning that it is zero everywhere where the indices are out of bounds. What you find if you visualize the computation of F using

Eq. (26), is that it is effectivly the same as using Eq. (16) with the kernel flipped about its horisontal and vertical axes. Which in turn means that flipping the kernel relative to the input, renders the computation of the feature map commutative. But this is a property that is usefull in mathematical contexts and less so for computing neural networks. It is Eq. (16) that is commonly used in macine learning applications and the associated mathematical operations is actually what is called the *cross-correlation*. Despite this, neural networks that implement these kinds of neurons are called convolutional even if the underlying mathematical operations is more often than not a close relative of the proper convolution operation.

2.5 Downsampling layer

The downsampling layer of a convolutional neural network is a special 'non-thinking' layer. The prupose of the downsampling is to reduce the data flowing through the network. Potentially reducing the redudancy present in the data. A downsampling layer also offers a significant speed improvment and reduced memory footprint of the network on the computer hardware.

There are two common methods of performing downsampling: average pooling and max pooling. Max pooling is in almost all cases the better alternative and the one used in our example network later. So here we are only going to be focusing on downsampling by max pooling.

A downsampling layer takes as its input a multi-channel data stream of scalar matricies. In maxpooling, the downsampling is performed by scanning a predetermined size of the input matrices at a time, selecting the largets value and placing it in an output matrix.

Like the kernel in the convolutional layer the 'downsampler' can use strides larger than 1 while scanning their input matrix. Figure 7 illustrates a maxpooling operation with a stride of two.

The size of the downsampling region is for the designer of the neural network to decide. Let δ_x and δ_y denote this size. Let s_x and s_y be the strides in the xy-directions. Let l be the index of a downsampling layer. With these notations, we can write the relationship between the input and output layers dimentions.

$$\begin{aligned} \eta_c^{(l)} &= \eta_c^{(l-1)} \\ \eta_x^{(l)} &= \left\lfloor \frac{\eta_x^{(l-1)} - \delta_x}{s_x} \right\rfloor + 1 \\ \eta_y^{(l)} &= \left\lfloor \frac{\eta_y^{(l-1)} - \delta_y}{s_y} \right\rfloor + 1 \end{aligned} \tag{28}$$

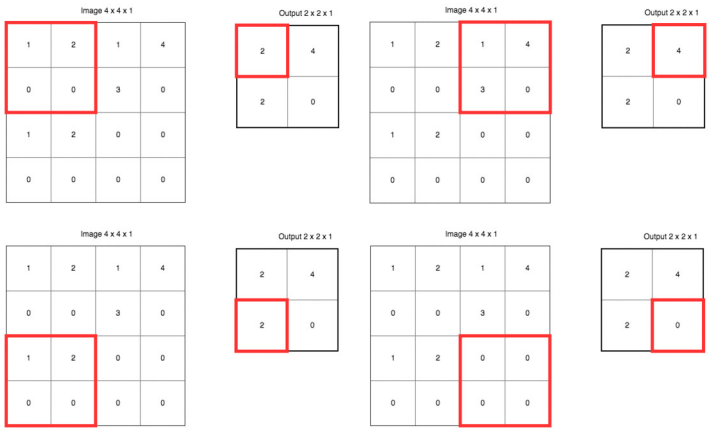

Figure 7 Downsampling accross an input matrix [2].

The equation which governs the forward operation of this layer is:

$$a_{(c)}^{(l)}(x,y) = \text{maxpool}\left(a_{(c)}^{(l-1)}(x,y)\right) = \max_{\substack{r_x\in[0\,..\,\delta_x]\\ r_y\in[0\,..\,\delta_y]}} \left(a_{(c)}^{(l-1)}(xs_x + r_x, ys_y + r_y)\right) \tag{29}$$

The use of the a variables alert the reader that the downsampling layer does not need an activation function hence we skip the z directly.

The equation which governs backpropagation can be calculated using equation **29**.

$$\frac{\partial a_{(c)}^{(l)}(x,y)}{\partial a_{(c)}^{(l-1)}(x,y)} = \begin{cases} 1 & \text{if } a_{(c)}^{(l-1)}(x,y) = \text{maxpool}\left(a_{(c)}^{(l-1)}(x,y)\right) \\ 0 & \text{otherwise} \end{cases}$$

By the chain rule we get:

$$\frac{\partial \ell}{\partial a_{(c)}^{(l-1)}(x,y)} = \begin{cases} \frac{\partial \ell}{\partial a_{(c)}^{(l)}(x,y)} & \text{if } a_{(c)}^{(l-1)}(x,y) = \text{maxpool}\left(a_{(c)}^{(l-1)}(x,y)\right) \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

2.6 The activation function

Let us now discuss for a brief moment the activation functions of our network. The ones responsible for turning our z's into a's. In principle, any suitable non-linear function can be used as an activation function. It has to be non-linear becasue the combination of two linear functions remains a linear function. Meaning that a layer directly followed by another layer without an activation function would in essence function as a single layer—a single linear function.

When designing a network one can chose between several different possible activation functions. The sigmoid function $\phi(x) = \frac{1}{1+e^{-x}}$ and the arctan functions are possible candidates for activation functions. The one our example network uses, and the one shown to be more efficient for optimization [1], is the rectified linear unit (ReLU) activation function. Our network also makes use of the softmax activation function at the very last layer.

The ReLU fucntion is extremly simple, being nothing more than a max and a zero.

$$\text{ReLU}(z) = \max(0, z) \tag{31}$$

The ReLU function is technicaly not derivable at 0 but for nerual network applications this can safely be ignored. We simple set the derivative of it to be zero at zero most of the time.

$$\frac{\mathrm{d}}{\mathrm{dz}}\Big(\text{ReLU}(z) = \max(0,z)\Big) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{32}$$

The softmax activation function is interesting because it does not behave like a regular activation function. Its purpose is to map the activations of the previous layer into a vector where the entries sum to 1. As such, it depends on all the preliminary outputs per activation. Letting $a_{(c)}^{(l)}$ be the activation of a softmax function in layer l we have

$$a_{(c)}^{(l)} = \frac{\exp\left(z_{(c)}^{(l)}\right)}{\sum_{c^*=0}^{\eta_c^{(l)}} \exp\left(z_{(c^*)}^{(l)}\right)} \tag{33}$$

The partial derivatives of the softmax function are:

$$\begin{aligned}
\frac{\partial a^{(l)}_{(c)}}{z^{(l)}_{(c)}} &= \frac{\exp\left(z^{(l)}_{(c)}\right)\sum_{c^*=0}^{\eta^{(l)}_c}\exp\left(z^{(l)}_{(c^*)}\right) - \left(\exp\left(z^{(l)}_{(c)}\right)\right)^2}{\left(\sum_{c^*=0}^{\eta^{(l)}_c}\exp\left(z^{(l)}_{(c^*)}\right)\right)^2} = a^{(l)}_{(c)} - \left(a^{(l)}_{(c)}\right)^2 = a^{(l)}_{(c)}(1-a^{(l)}_{(c)}) \\
\frac{\partial a^{(l)}_{(c)}}{z^{(l)}_{(c')}} &= -\frac{\exp\left(z^{(l)}_{(c)}\right)\exp\left(z^{(l)}_{(c')}\right)}{\left(\sum_{c^*=0}^{\eta^{(l)}_c}\exp\left(z^{(l)}_{(c^*)}\right)\right)^2} = -a^{(l)}_{(c)}a^{(l)}_{(c')}
\end{aligned} \tag{34}$$

Where $c' \neq c$.

2.7 Output layer

The output layer is not really a layer at all. Here we instead find the loss function of the network (asuming it is undergoing traning). There are a multitude of loss functions available which one can use at the end of a neural network.

In the network that we are using for our example, the function in use is the *cross category entropy* loss function. Let l be the last 'proper' layer of the network. The cross category entropy loss function with output here denoted by ℓ is defined as:

$$\ell = -\sum_{c=0}^{\eta^{(l)}_c} y_{(c)} \log\left(a^{(l)}_{(c)}\right) \tag{35}$$

Where $y_{(c)}$ is the correct value for the data in channel c. $a^{(l)}_{(c)}$ is the last activation of the network. Which we here interprete as the probability the network assigned to $y_{(c)}$ being the correct answer.

We can use equation **35** to calculate the inital partial derivaties of the backward propagation operation. They turn out to be:

$$\frac{\partial \ell}{\partial a^{(l)}_{(c')}} = \frac{\partial}{\partial a^{(l)}_{(c')}}\left(-\sum_{c=0}^{\eta^{(l)}_c} y_{(c)} \log\left(a^{(l)}_{(c)}\right)\right) = -\frac{y_{(c')}}{a^{(l)}_{(c')}} \tag{36}$$

3 Python Implementation

Knwon that we have taken a look at the theory behind convolutional neural networks, let us look at an implementation of one. The nerual network we are going to be looking at is one written by Alejandro Escontrela. It is written in Python using the NumPy library and is publicly avaliable at `https://github.com/Alescontrela/Numpy-CNN.git`

The network is going to tackle the classic neural network problem. Categorizing the handwritten digits in the MNIST database. Compared to other problems convolutional neural networks are commonly faced with, categorizing the MNIST images is rather simple. So the network will use a comperativly simple architecture as shown in Figure **8** on page **22**. The network uses no zero padding. The first two layer consists of two indentical convolutional layers. They are in turn follwed by a single max-polling layer which notably, uses a stride of 2. The output is then flattened and passed to a dense layer with 128 neurons. The output of this layer is passed to the final dense layer, which necessarly consists of 10 neurons.

3.1 Convolution function

Here is the python code for the forward convolution operation of a given layer.

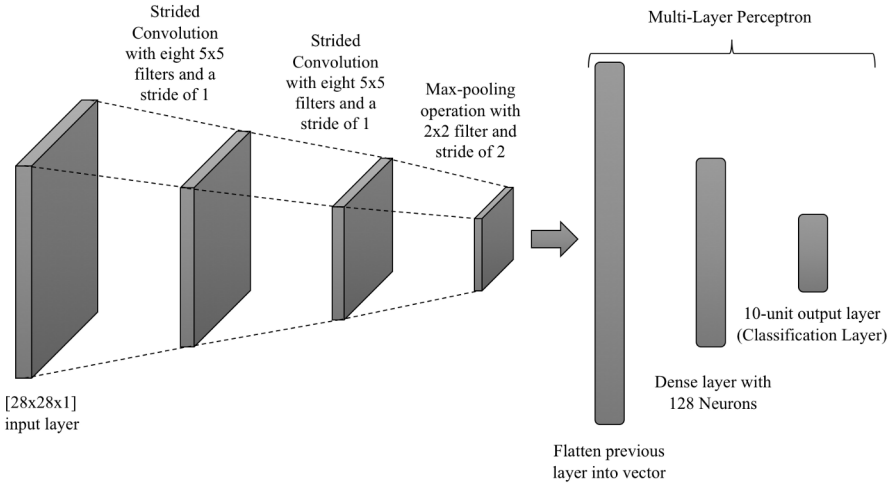

Figure 8 The architecture of the network

```
def convolution(image, filt, bias, s=1):
    '''
    Confolves `filt` over `image` using stride `s`
    '''
    (n_f, n_c_f, f, _) = filt.shape # filter dimensions
    n_c, in_dim, _ = image.shape # image dimensions

    out_dim = int((in_dim - f)/s)+1 # calculate output dimensions

    assert n_c == n_c_f, "Dimensions of filter must match dimensions of input
image"

    out = np.zeros((n_f,out_dim,out_dim))

    # convolve the filter over every part of the image, adding the bias at
each step.
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= in_dim:
            curr_x = out_x = 0
            while curr_x + f <= in_dim:
                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] *
image[:,curr_y:curr_y+f, curr_x:curr_x+f]) + bias[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return out
```

3.2 Max pooling

Here is the python code for the forward and backward max-pooling operation of the network

```
def maxpool(image, f=2, s=2):
    '''
    Downsample `image` using kernel size `f` and stride `s`
    '''
    n_c, h_prev, w_prev = image.shape

    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1

    downsampled = np.zeros((n_c, h, w))
    for i in range(n_c):
        # slide maxpool window over each part of the image and assign the max
value at each step to the output
        curr_y = out_y = 0
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            while curr_x + f <= w_prev:
                downsampled[i, out_y, out_x] = np.max(image[i,
curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled
```

```
def maxpoolBackward(dpool, orig, f, s):
    '''
    Backpropagation through a maxpooling layer. The gradients are passed
through the indices of greatest value in the original maxpooling during the
forward step.
    '''
    (n_c, orig_dim, _) = orig.shape

    dout = np.zeros(orig.shape)

    for curr_c in range(n_c):
        curr_y = out_y = 0
        while curr_y + f <= orig_dim:
            curr_x = out_x = 0
            while curr_x + f <= orig_dim:
                # obtain index of largest value in input for current window
                (a, b) = nanargmax(orig[curr_c, curr_y:curr_y+f,
curr_x:curr_x+f])
                dout[curr_c, curr_y+a, curr_x+b] = dpool[curr_c, out_y, out_x]

                curr_x += s
                out_x += 1
```

```
                curr_y += s
                out_y += 1

        return dout
```

3.3 Various functions

The network uses ReLU for its activation function and softmax paired with cross entropy loss function.

```
    def softmax(X):
        out = np.exp(X)
        return out/np.sum(out)

    def categoricalCrossEntropy(probs, label):
        return -np.sum(label * np.log(probs))
```

3.4 Forward and backward operation of network

We define the backwards operation in a convolutional layer in the function "convolutionalBackward" as such:

```
    def convolutionBackward(dconv_prev, conv_in, filt, s):
        '''
        Backpropagation through a convolutional layer.
        '''
        (n_f, n_c, f, _) = filt.shape
        (_, orig_dim, _) = conv_in.shape
        ## initialize derivatives
        dout = np.zeros(conv_in.shape)
        dfilt = np.zeros(filt.shape)
        dbias = np.zeros((n_f,1))
        for curr_f in range(n_f):
            # loop through all filters
            curr_y = out_y = 0
            while curr_y + f <= orig_dim:
                curr_x = out_x = 0
                while curr_x + f <= orig_dim:
                    # loss gradient of filter (used to update the filter)
                    dfilt[curr_f] += dconv_prev[curr_f, out_y, out_x] * conv_in[:,
    curr_y:curr_y+f, curr_x:curr_x+f]
                    # loss gradient of the input to the convolution operation
    (conv1 in the case of this network)
                    dout[:, curr_y:curr_y+f, curr_x:curr_x+f] +=
    dconv_prev[curr_f, out_y, out_x] * filt[curr_f]
                    curr_x += s
                    out_x += 1
                curr_y += s
                out_y += 1
            # loss gradient of the bias
```

```
            dbias[curr_f] = np.sum(dconv_prev[curr_f])

        return dout, dfilt, dbias
```

We can now define a single forward and backward operation for the network in the function "conv":

```
    def conv(image, label, params, conv_s, pool_f, pool_s):

        [f1, f2, w3, w4, b1, b2, b3, b4] = params

        ################################################
        ############## Forward Operation ###############
        ################################################
        conv1 = convolution(image, f1, b1, conv_s) # convolution operation
        conv1[conv1<=0] = 0 # pass through ReLU non-linearity

        conv2 = convolution(conv1, f2, b2, conv_s) # second convolution operation
        conv2[conv2<=0] = 0 # pass through ReLU non-linearity

        pooled = maxpool(conv2, pool_f, pool_s) # maxpooling operation

        (nf2, dim2, _) = pooled.shape
        fc = pooled.reshape((nf2 * dim2 * dim2, 1)) # flatten pooled layer

        z = w3.dot(fc) + b3 # first dense layer
        z[z<=0] = 0 # pass through ReLU non-linearity

        out = w4.dot(z) + b4 # second dense layer

        probs = softmax(out) # predict class probabilities with the softmax
    activation function

        ################################################
        ##################### Loss #####################
        ################################################

        loss = categoricalCrossEntropy(probs, label) # categorical cross-entropy
    loss

        ################################################
        ############# Backward Operation ###############
        ################################################
        dout = probs - label # derivative of loss w.r.t. final dense layer output
        dw4 = dout.dot(z.T) # loss gradient of final dense layer weights
        db4 = np.sum(dout, axis = 1).reshape(b4.shape) # loss gradient of final
    dense layer biases

        dz = w4.T.dot(dout) # loss gradient of first dense layer outputs
        dz[z<=0] = 0 # backpropagate through ReLU
        dw3 = dz.dot(fc.T)
```

```
33          db3 = np.sum(dz, axis = 1).reshape(b3.shape)

34          dfc = w3.T.dot(dz) # loss gradients of fully-connected layer (pooling
35      layer)
36          dpool = dfc.reshape(pooled.shape) # reshape fully connected into
37      dimensions of pooling layer

38          dconv2 = maxpoolBackward(dpool, conv2, pool_f, pool_s) # backprop through
39      the max-pooling layer(only neurons with highest activation in window get
40      updated)
41          dconv2[conv2<=0] = 0 # backpropagate through ReLU

42          dconv1, df2, db2 = convolutionBackward(dconv2, conv1, f2, conv_s) #
43      backpropagate previous gradient through second convolutional layer.
44          dconv1[conv1<=0] = 0 # backpropagate through ReLU

45          dimage, df1, db1 = convolutionBackward(dconv1, image, f1, conv_s) #
46      backpropagate previous gradient through first convolutional layer.

47          grads = [df1, df2, dw3, dw4, db1, db2, db3, db4]

48          return grads, loss
```

3.5 Batch Optimization

When you have a large training set, it can be time consuming to evaluate a cost function on the data set in its entirety. A common way to avoid this is by instead evaluating the cost function in batches, or subsets of the training data. In other words, for each batch, we evaluate the cost function and adjust the paramaters accordingly. After using all the training data to tweak the paramaters, we have finished an epoch. This process can then be repeated over several epochs, but in our case, we have found that two epochs works sufficiently. The fact that we adjust the paramaters after each batch, means that we have to choose a suitable optimization algorithm such as Stochastic Gradient Descent. In this implementation, however, the author has used an algorithm that is shown to be very efficent in terms of batch optimization, namely the Adam Gradient Descent algorithm. We will not spend time explaining how it works, but its main principles are very similar to that of the classic gradient descent. For each training data in the batch, we evaluate the output of the loss by the forward opartion and retrieve the gradients using the backward operation described earlier. Afterwards, we evaluate the cost function of the batch and adjust the parameters. The implementation of this concept in python, is summarized in figure ()

```
1   def adamGD(batch, num_classes, lr, dim, n_c, beta1, beta2, params, cost):
2       '''
3       update the parameters through Adam gradient descnet.
4       '''
5       [f1, f2, w3, w4, b1, b2, b3, b4] = params

6       X = batch[:,0:-1] # get batch inputs
7       X = X.reshape(len(batch), n_c, dim, dim)
8       Y = batch[:,-1] # get batch labels
```

```
        cost_ = 0
        batch_size = len(batch)

        # initialize gradients and momentum,RMS params

        for i in range(batch_size):

            x = X[i]
            y = np.eye(num_classes)[int(Y[i])].reshape(num_classes, 1) # convert
    label to one-hot

            # Collect Gradients for training example
            grads, loss = conv(x, y, params, 1, 2, 2)
            [df1_, df2_, dw3_, dw4_, db1_, db2_, db3_, db4_] = grads

            df1+=df1_
            db1+=db1_
            df2+=df2_
            db2+=db2_
            dw3+=dw3_
            db3+=db3_
            dw4+=dw4_
            db4+=db4_

            cost_+= loss

        # Parameter Update



        cost_ = cost_/batch_size
        cost.append(cost_)

        params = [f1, f2, w3, w4, b1, b2, b3, b4]

        return params, cost
    def train(num_classes = 10, lr = 0.01, beta1 = 0.95, beta2 = 0.99, img_dim =
    28, img_depth = 1, f = 5, num_filt1 = 8, num_filt2 = 8, batch_size = 32,
    num_epochs = 2, save_path = 'params.pkl'):

        # training data
        m =50000
        X = extract_data('train-images-idx3-ubyte.gz', m, img_dim)
        y_dash = extract_labels('train-labels-idx1-ubyte.gz', m).reshape(m,1)
        X-= int(np.mean(X))
        X/= int(np.std(X))
        train_data = np.hstack((X,y_dash))
```

```
12          np.random.shuffle(train_data)
13
14          ## Initializing all the parameters
15          f1, f2, w3, w4 = (num_filt1 ,img_depth,f,f), (num_filt2 ,num_filt1,f,f),
16      (128,800), (10, 128)
17          f1 = initializeFilter(f1)
18          f2 = initializeFilter(f2)
19          w3 = initializeWeight(w3)
20          w4 = initializeWeight(w4)
21
22          b1 = np.zeros((f1.shape[0],1))
23          b2 = np.zeros((f2.shape[0],1))
24          b3 = np.zeros((w3.shape[0],1))
25          b4 = np.zeros((w4.shape[0],1))
26
27          params = [f1, f2, w3, w4, b1, b2, b3, b4]
28
29          cost = []
30
31          print("LR:"+str(lr)+", Batch Size:"+str(batch_size))
32
33          for epoch in range(num_epochs):
34              np.random.shuffle(train_data)
35              batches = [train_data[k:k + batch_size] for k in range(0,
36      int(train_data.shape[0]/16), batch_size)]
37
38              t = tqdm(batches)
39              for x,batch in enumerate(t):
40                  params, cost = adamGD(batch, num_classes, lr, img_dim, img_depth,
41      beta1, beta2, params, cost)
42                  t.set_description("Cost: %.2f" % (cost[-1]))

43          to_save = [params, cost]

44          with open(save_path, 'wb') as file:
45              pickle.dump(to_save, file)

46          return cost
```

3.6 Results

Using our own laptops, we initially trained the network on the entire MNIST data set, which is in fact 10.000 labeled images of handwritten digits. The results are are represented on figure (). Though the network's predictions were accurate, we found that the eight-hour total run-time of the training phase was too long. To remedy this, we simply reduced the training set by a factor of 2 then by a factor of 16, respectively achieving a run-time of 4 hours and then 30 minutes.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016).

[2] A. Escontrela, *Convolutional Neural Networks from the ground up* (Towards Data Science, 2018).

[3] P. Solai, *Convolutions and Backpropagations* (Medium, 2018).

[4] N. Sean, *Image of neural network structure* (Knowing Neurons, 2018).

[5] Y. S, *Gradient descent illustration* (2019).

[6] M. Agarwal, *Illustration of forward and backward propagation* (2017).