UNSW
AUSTRALIA

COMP4121

Major Project

Convolutional Neural Networks

Course Convenor: Aleks Ignjatovic

Table of Contents

1 What is an Artificial Neural Network? 1

1.1 Perceptrons 2

1.2 Multi-Layer Perceptrons 4

1.3 Cost Function 7

1.4 Gradient Descent 8

1.5 Stochastic Gradient Descent 12

1.6 The Idea Behind Back Propagation 12

1.7 The Math Behind Back Propagation 17

2 What is a Convolutional Neural Network? 24

2.1 Convolutions 24

2.2 Convolutional Layer 28

2.3 Pooling Layer 31

2.4 Fully Connected Layer 33

In this paper, we will look at how convolutional neural networks work focussing on the example of classifying handwritten digits. Understanding convolutional neural networks requires foundational knowledge of standard artificial neural networks, so we will first explore basic neural networks and the algorithms behind them, again following the example of handwritten digit recognition.

1 What is an Artificial Neural Network?

An artificial neural network is a type of model based on the neural structure of the brain.

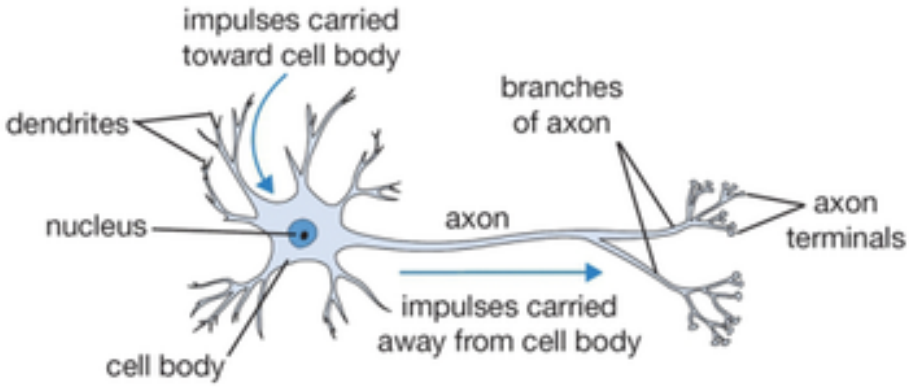

Figure 1: A diagram of a neuron in the brain (Karlijin Willems, 2019)

The biological neuron takes in electrical inputs from its multiple dendrites, sums these all up and sends the output through its single axon (Roell, 2017).

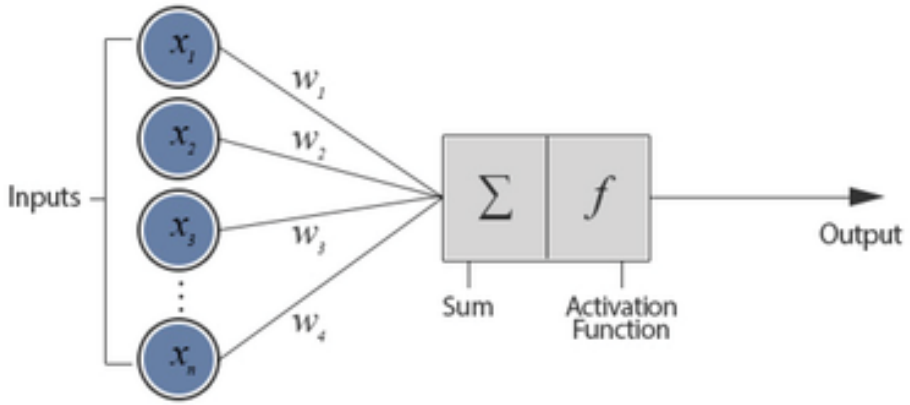

Figure 2: An artificial neuron (Karlijin Willems, 2019)

The artificial neuron uses this same basic structure, it takes in multiple inputs, does some computation, and sends a single output. (Roell, 2017)
An artificial neuron makes up a single component of our neural network, we can combine these artificial neurons into a layer structure to build our more complex neural network.

1.1 Perceptrons

A perceptron consists of a single artificial neuron, and in itself already forms a very simple neural network. Let us follow Figure 2 from left to right to illustrate the components of the perceptron.

Starting on the left we have our input nodes $x_1.\cdots, x_n$, which are real numbers. Then for each of these inputs we have the corresponding weights $w_1, \cdots, w_n$ which are also real numbers, note here, although this is not illustrated in the Figure 2, we often also add an extra weight w_0 and a corresponding input node $x_0 = 1$, this is called the bias and allows for shifting of the output of our activation function.

Then we calculate the weighted sum $w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$, and this is what we feed into the activation function f. The value of the activation function at a particular neuron is called the activation of the neuron. A common activation function for the perceptron is the step function, it is somewhat based on the biological neuron

which only fires an output when it is above a certain threshold, mathematically this function is defined as,

$$f(x) = \begin{cases} 0 & \text{, if } x \leq 0 \\ & \\ 1 & \text{, if } x > 0 \end{cases}.$$

On it's own, this perceptron gives us a linear decision boundary which is not so useful for more complicated data. We can increase the complexity of our model by using non-linear activation functions.

A common non-linear activation function is the sigmoid function or logistic curve (Roell, 2017),

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

This is a continuous function valued between 0 and 1, this property is useful for predicting probabilities in a model (Sagar Sharma, 2017). When the sigmoid function is used as an activation function, this type of neuron is called a sigmoid neuron, or sigmoid unit. We will be using the sigmoid function as our activation function of choice when we look at the example of handwritten digit recognition.

Another common non-linear activation function worth mentioning is the ReLU (Rectified Linear Unit) activation function, defined as

$$R(x) = \max(0, x).$$

This activation function essentially removes all negative values by setting them to zero. A neuron that uses the ReLU function as its activation function is called a ReLU unit.

Now if we look back, we can see that different weights $w_1, \cdots, w_n$ will give different outputs from the activation function. If we increase the weight w_i this would give more importance to the input x_i, and conversely if we decreased the weight w_i we would be giving less importance to the input x_i (Roell, 2017). This leads us to ask how do we choose these weights? These weights must be 'learned', and we will discuss this more in the topic of gradient descent.

Now that we have looked at the most basic component of a neural network, the single perceptron, let us build the 'classic' neural network, the multi-layer perceptron.

1.2 Multi-Layer Perceptrons

In multi-layer perceptrons, perceptrons are organised in a layered structure, with at least three layers - an input layer, and output layer and one or more hidden layers, this is illustrated in Figure 3 below with a single hidden layer.

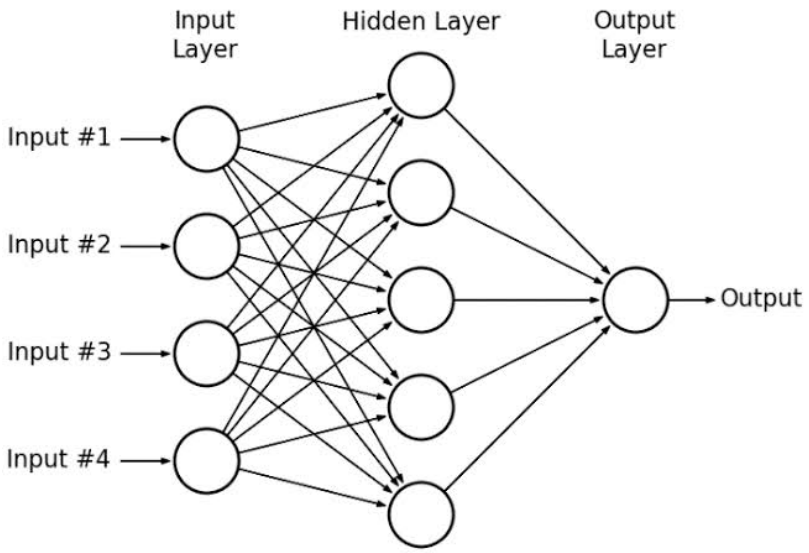

Figure 3: A multi-layer perceptron (Mohamed Zahran, 2015)

By having only a single perceptron we can only make very simple decisions based on one pass through the activation function, but by adding multiple layers of perceptrons we can make more complex and subtle decisions, and as we move through each layer of the network, we begin to make decisions in an increasingly abstract way (Michael Nielsen, 2019b).

In this model, each perceptron gives its output as inputs to all the perceptrons in the next layer and so on. Note that although in Figure 3 it appears as though each perceptron has multiple outputs, this is not the case, each perceptron only has a single output and the multiple arrows leaving each node represents sending the single output to each node in the next layer (Michael Nielsen, 2019b). Each connection in

the network has a different weight associated to it, to highlight this further, let us consider the mathematical expression for the activation of the m-th neuron in the l-th layer of the network, calling this value $a_m^{(l)}$. Call the weight connecting the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer, $w_{jk}^{(l)}$, and bias corresponding to the m-th neuron in the l-th layer of the network $b_m^{(l)}$. Then the expression for the activation in this neuron using the sigmoid function is

$$a_m^{(l)} = \sigma\Big(\sum_{k=1}^{n_{l-1}} a_k^{(l-1)} w_{jk}^{(l)}\Big),$$

where $n_{(l-1)}$ is the number of neurons in the $(l-1)$-th layer. The figure below illustrates two layers of a network with activations and weights.

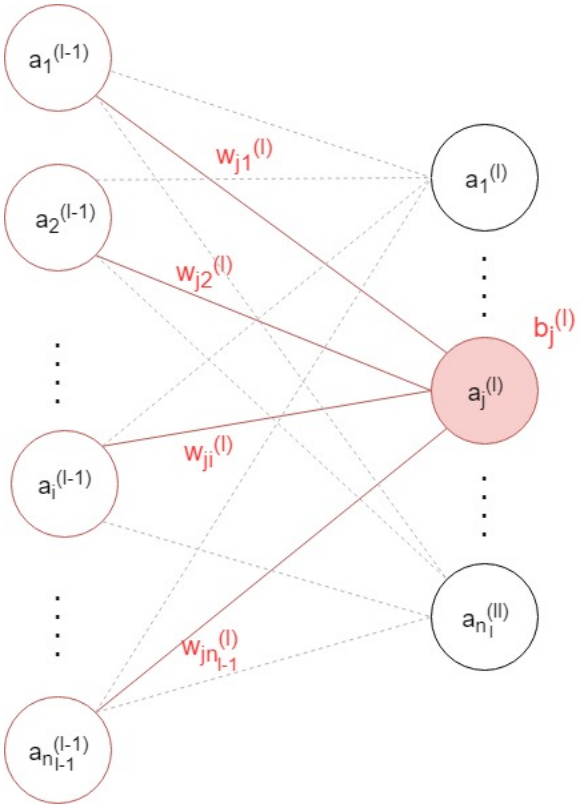

Figure 4: Activations and weights in a multi-layer perceptron

Let us now see how we could apply this structure to the example of handwritten digit recognition. We want our neural network to be able to read images of handwritten digits (see Figure 4) and then output the number it believes it to be.

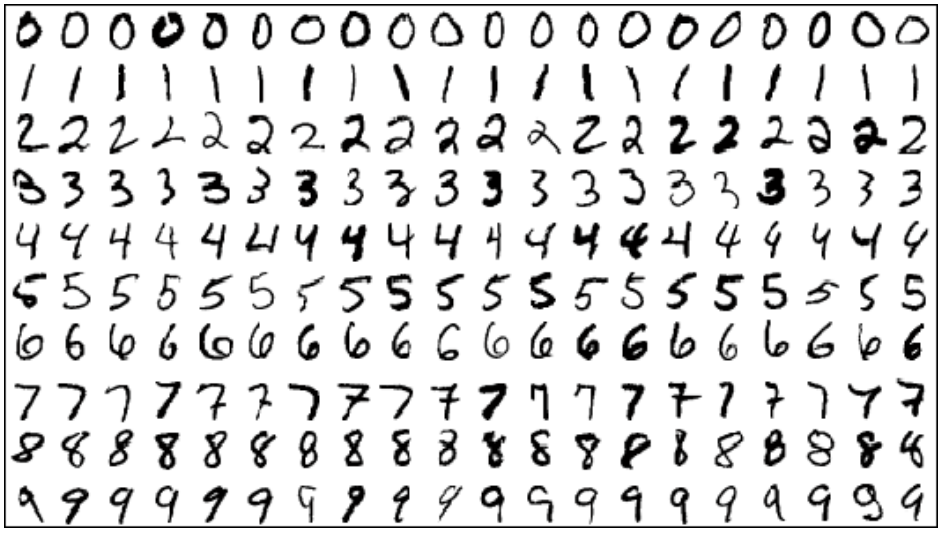

Figure 5: Handwritten digit dataset from MNIST database (Synced, 2019)

Suppose we are given a 28×28 pixel, greyscale image, where the colour of each pixel can be represented by a number between 0 and 1, 0 being white and 1 being black (and numbers inbetween represent grey). Then we can have $28 \times 28 = 784$ input neurons where the neuron takes the colour value of the corresponding pixel as the input value, and the output layer will contain 10 neurons where each neuron represents the digits 0-9.

Then if the first output neuron (corresponding to the digit zero) outputs a high value close to 1, then this would mean the network believes the image is nearly certainly a zero, however if it outputs a number close to 0 then this means the network believes the image is nearly certainly not a zero, and similarly with the output neurons corresponding to all the other digits. So each output node will produce a probability value representing how much the network believes it is that corresponding number (Michael Nielsen, 2019b).

When we begin to explore the algorithms involved with neural networks, we will denote the input values as a vector $\boldsymbol{x}$, and with our digit recognition example, $\boldsymbol{x}$ is a vector with 784 entries, and we will denote the desired output value for input $\boldsymbol{x}$ as a vector $\boldsymbol{y} = y(\boldsymbol{x})$ where in our example $\boldsymbol{y}$ is a vector with 10 entries.

1.3 Cost Function

Now, let us explore how we learn the weights in the network as mentioned in Section 1.1. Our aim is to find the set of all weights and biases in our network so that the network outputs the most accurate results.

With the example of digit recognition, if we feed in a handwritten image of the number 5, we want the network to accurately output 5, but if it doesn't, we want to adjust the weights and biases so that the network produces a more accurate output. However we don't want to do this for just one input, we want to optimise the accuracy across our entire dataset, so this means in order to decide how to change the weights and biases in the network we want to take into consideration all the training examples and how the network performs on all these examples.

We want to be able to quantify how well our network is performing, so to do this we will define a cost function as follows, where a high output value from the cost function means the network is performing poorly, and a low value means the network is performing well. Let $\boldsymbol{w}$ be the collection of all weights and biases in our network, let n be the number of training inputs, and let $a(\boldsymbol{x})$ be the output vector of the network when the input vector is $\boldsymbol{x}$, note that $a(\boldsymbol{x})$ is dependent not only on $\boldsymbol{x}$, but also the weights and biases $\boldsymbol{w}$ of the network. The cost function C is defined as

$$C(\boldsymbol{w}) = \frac{1}{n} \sum_{\boldsymbol{x}} \|y(\boldsymbol{x}) - a(\boldsymbol{x})\|^2 \tag{1}$$

where $\|\boldsymbol{v}\|$ denotes the standard Euclidean norm of the vector $\boldsymbol{v}$ (Michael Nielsen, 2019b).

Let's look at what is actually going on in this cost function. Consider one training input $\boldsymbol{x}$, we compare the output from the network when given the input $\boldsymbol{x}$ which we denoted by $a(\boldsymbol{x})$ with the desired output $y(\boldsymbol{x})$. We do this by taking the difference between the two vectors, we can interpret this by looking at the resulting difference vector $y(\boldsymbol{x}) - a(\boldsymbol{x})$. If the entries of the vector are large, then this means the output differs a lot from what was desired, and if the entries are small values, then the output

can be considered close to the desired output. A way to quantify this is by taking the squared norm of the difference vector, $\|y(\boldsymbol{x}) - a(\boldsymbol{x})\|^2$.

Then since we need to take into account all the training data, we want to calculate the squared norm of the difference vector for all $\boldsymbol{x}$ in the training inputs and sum these all up which gives, $\sum_{\boldsymbol{x}} \|y(\boldsymbol{x}) - a(\boldsymbol{x})\|^2$. Finally we take the average of these values by dividing the sum by the number of training inputs n, and this gives us the cost function $C(\boldsymbol{w}) = \frac{1}{n} \sum_{\boldsymbol{x}} \|y(\boldsymbol{x}) - a(\boldsymbol{x})\|^2$, this function is called the mean squared error.

Notice that $C(\boldsymbol{w})$ is a non-negative function, and that when $C(\boldsymbol{w})$ is small ie. close to 0, this indicates that our outputs, $a(\boldsymbol{x})$, are close to the desired values, $y(\boldsymbol{x})$, across all the training input, which tells us that the network is performing well with $\boldsymbol{w}$ as our set of weights and biases. Conversely if the value of $C(\boldsymbol{w})$ is large then this indicates that our network is performing poorly and that we need to adjust the weights and biases (Michael Nielsen, 2019b).

So as we can see, $C(\boldsymbol{w})$ gives a quantitave value to the performance of our network with $\boldsymbol{w}$ as the set of weights and biases, and when $C(\boldsymbol{w})$ is small our network is performing well. This brings us to our big goal for building our neural network, we aim to find the set of weights and biases $\boldsymbol{w}$ that minimises $C(\boldsymbol{w})$. Which brings us to exploring gradient descent.

1.4 Gradient Descent

Gradient descent is an algorithm used to minimise any function, and not just in the context of neural networks (Michael Nielsen, 2019b), note that gradient descent only helps us to find a local minimum, but does not necessarily find a global minimum, which is a much harder task to achieve. However, we might ask ourselves, why do we need a fancy algorithm to find the minimum of a function, can't we just use calculus? As we might already know from doing really hard calculus practice questions, sometimes finding an explicit expression for a minimum point of a function is very difficult, so we need a method that ensures we are able to find a minimum without explicit

computation.

Let us first visualise the problem on the two-dimensional plane, see Figure 5 below.

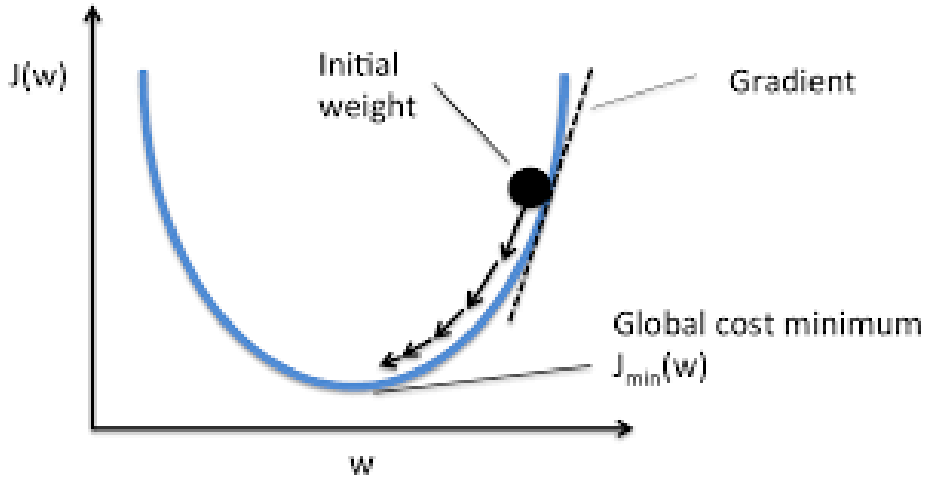

Figure 6: Visualisation of the gradient descent algorithm (Suryansh S, 2018)

The idea behind gradient descent is that if we have a point close to the minimum of the curve then the slope at that point is close to zero, opposingly if we have a point a bit further away from the minimum then the slope gets bigger in magnitude, so the slope at a point can tell us how far we are from the minimum. Also, the direction of the slope tells us in which direction we must move to get closer to the minimum, a postive slope tells us the minimum is to the left, while a negative slope tells us the minimum is to the right. Keeping these ideas in mind, let us introduce the algorithm for gradient descent.

Let us first analyse the univariate case, where our function to be minimised is a function of one variable. Suppose we want to minimise the single variable function $C(w)$, that is, we want to find the value of w that makes $C(w)$ the smallest. We will do this in iterations, where at each iteration we update our value of w so that we get closer and closer to the minimum point. We begin by choosing some initial value for w, then the update rule for w is

$$w := w - \alpha \frac{d}{dw} C(w),$$

where α is a fixed, predetermined value called the learning rate (Misa Ogura, 2017).

Let's break apart this algorithm to see what is going on underneath. As we mentioned earlier, the slope at a point gives us information about how close we are to the minimum and in what direction the minimum is, the slope, or also called the gradient is represented by the derivative of the function, in this case $\frac{d}{dw}C(w)$. So we can see that each update we move w by α multipled by the value of the derivative at w.

Thus, if the value of the derivative is large then we move w by a large amount which agrees with what was mentioned earlier that large slope means we are further away from the minimum and hence we can taking a larger step in the direction of the minimum. Conversely, if the derivative is small, then we only move w by a small value, since small derivative means we are close to the minimum and should only take small steps as to not overshoot the minimum point.

We should also notice that when the derivative is negative, then according to the algorithm, we add a positive value to w which means we move w to the right, and when the derivative is positive, we subtract a positive value, moving w to the left, this is depicted in Figure 5.

Finally let us address the learning rate α, this is a fixed value that does not change during the process of gradient descent. As you may be able to tell from the name the learning rate, α determines how fast or slow the value of w moves (Misa Ogura, 2017), if α is small, this allows w to take smaller steps, but also means it will take longer to find the minimum, and conversely if α is big then w can take larger steps, and potentially find the minimum faster. However, we want to pick a good inbetween value for α that is not too small and not too big, since if we pick an α that is too small, the algorithm will not converge in a feasible amount of time, whereas if we pick an α too large, then we might overshoot the minimum and not converge to any value (Misa Ogura, 2017).

Using this as our foundation for gradient descent, let us now look at gradient descent in three dimensions that is, when our function takes two variables instead of one, and from there it should be easy to generalise to n dimensional gradient descent.

Suppose we have a multivariable function $C(\boldsymbol{w})$, where $\boldsymbol{w} = (w_1, w_2)^T$ is a two dimensional vector, that we wish to minimise. In the three dimensional problem, the slope is represented by the two partial derivatives of the function, $\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}$. So with the univariate case, we only had two directions we could move w, but in the multivariate case, we have many more directions, and we need to find the direction of steepest descent, that is, the direction that gives the most decrease in $C(\boldsymbol{w})$ (Grant Sanderson, 2017b). We can represent this mathematically using the gradient function of C, ∇C which is a vector defined as

$$\nabla C = \Big(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\Big)^T$$

(Michael Nielsen, 2019b). In fact, this gradient vector gives us the vector of steepest increase, so the vector of steepest decrease is $-\nabla C$, and further, the norm of this vector quantitatively tells us exactly how steep the descent is (Grant Sanderson, 2017b). So now, our update rule for $\boldsymbol{w}$ is

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \nabla C,$$

where again α is the learning rate and behaves just as mentioned previously.

Now, let us briefly address the n dimensional case, which follows almost directly from the three dimensional case. So, suppose we want to minimise the function $C(\boldsymbol{w})$ where $\boldsymbol{w} = (w_1, \cdots, w_n)$ is an n dimensional vector, and let us define the gradient vector of C much like in the three dimensional case,

$$\nabla C = \Big(\frac{\partial C}{\partial w_1}, \cdots, \frac{\partial C}{\partial w_n}\Big)^T.$$

Again, $-\nabla C$ gives us the vector of steepest descent, and so we define the update rule as,

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \nabla C,$$

where the intuition behind this rule is exactly the same as the previous two cases.

1.5 Stochastic Gradient Descent

Let's now think about how we compute gradient descent on our training data. Looking at the cost function in equation (1), to compute it we need to compute the expression $\|y(\boldsymbol{x}) - a(\boldsymbol{x})\|$ for every single $\boldsymbol{x}$ in our training input. So when we have a very large number of training samples, then this becomes computationally infeasible (Michael Nielsen, 2019b).

Instead, we can apply a method called stochastic gradient descent which takes small batches of the training data and computes the gradient ∇C for each input in the mini-batch and takes the average, and this turns out to be a good approximation for the gradient of C. More explicitly, we randomly divide the full training dataset into mini-batches of size m then considering only the first mini-batch, let's call each input in this batch $X_1, \cdots, X_m$. Then given that m is large enough so that the average across ∇C_{X_j} is a good approximation for the average of ∇C_x across the entire training set, then

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_{\boldsymbol{x}} \nabla C_{\boldsymbol{x}}}{n} = \nabla C \tag{2}$$

(Michael Nielsen, 2019b). Then at each update for the weights $\boldsymbol{w}$, we compute ∇C with a new mini-batch, and when we use up all the mini-batches we just start again from the first, and this goes on until the gradient descent algorithm converges.

1.6 The Idea Behind Back Propagation

The following two sections on back propagation uses ideas and examples from Grant Sanderson's videos on back propagation, 'What is Back Propagation Really Doing?' and 'Back Propagation Calculus'.

Now let us explore how gradient descent is implemented in the context of neural networks, back propagation is a method used to find the negative gradient of the cost function, $-\nabla C$, that we need for our gradient descent alogrithm.

First let us interpret the negative gradient of our cost function C. Since interpreting large dimensional vectors is very difficult, and it is impossible to visualise, let's try to understand the components in the vector with the following example.

Suppose we calculate the negative gradient of C for some set of weights and biases, and suppose the component corresponding to the first weight is 3.2, while the component corresponding to the second weight is 0.1, that is, $-\nabla C(\boldsymbol{w}) = (3.2, 0.1, \cdots)^T$, this would mean that adjusting the first weight by a certain amount would have 32 times more of an impact on the cost function than if we were to adjust the second weight by the same amount (Grant Sanderson, 2017c).

Now, let us look at a single training input and see how this one input changes all the weights and biases in the network. Suppose we input an image of a digit two, on an untrained network with random weights and biases, so the output is nowhere near the output that we would hope for, our aim is to change the weights and biases so that we get our desired output of 2, see figure 6 below.

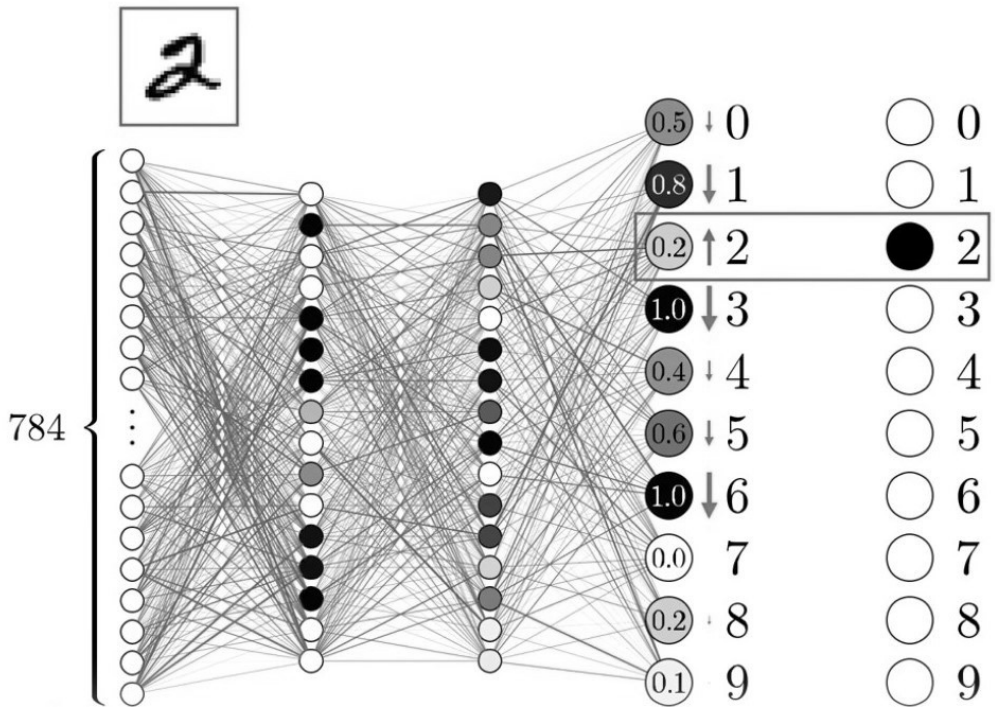

Figure 7: Example output from input of digit two with random weights and biases (Grant Sanderson, 2017c)

From the figure the output vector given this particular input is

$$(0.5, 0.8, 0.2, 1.0, 0.4, 0.6, 1.0, 0.0, 0.2, 0.1)^T,$$

, but since the image is of a two, we want the network to output

$$(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)^T.$$

Although we cannot directly change the activations of any neuron, we need to think about what the desired changes to the output are in order to know what weights and biases to change.

Since the input is of a two, we want our network to output the vector corresponding to the digit two. So, we want the value corresponding to the digit two to be closer to 1, therefore we want to increase the third component of the output vector from 0.2 to 1, and we want all other components to decrease to 0.

Additionally, we should also consider how much we want to increase and decrease the components by, intuitively we should increase or decrease a component in a way that is proportional to the difference between the current value and the desired output value (Grant Sanderson, 2017c).

In this example, the value 0.2 in the third component of the output vector is far from it's target value of 1, so we would like to increase it by a significant amount. In contrast, the last component 0.1 which has target value 0, does not need to be changed by much, since 0.1 is close to 0.

Now let's focus more closely on the neuron whose activation we want to increase, see the figure below.

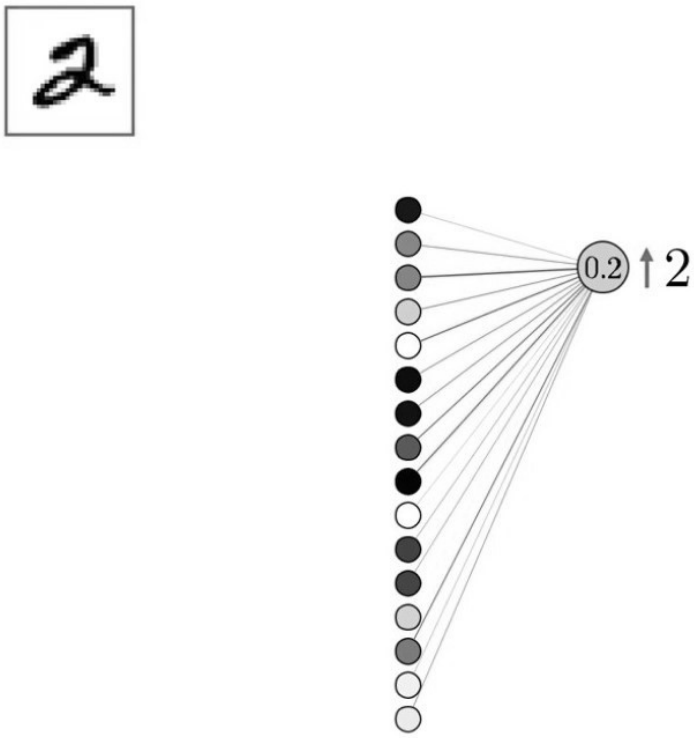

Figure 8: Focusing on the single neuron (Grant Sanderson, 2017c)

Recall how that activation of 0.2 in the figure above would have been computed, it is the output of whatever activation function we choose, in this example we will use the sigmoid function σ,

$$0.2 = \sigma(w_0 + w_1a_1 + w_2a_2 + \cdots + w_na_n),$$

where w_0 is the bias, the w_i for $1 \leq i \leq n$ are the weights of the previous layer and the a_i are the activations of the previous layer.

So from observing this, we can see that there are three ways of increasing the activation of 0.2, "we can increase the bias, increase the weights, and change the activations from the previous layer" (Grant Sanderson, 2017c).

Also, notice that increasing some weights will have greater effect than increasing other weights, if we increase the weights multiplied by higher activations (the darker circles in the figure) this will have a greater impact than increasing a weight that is multiplied by a smaller activation, thus we should increase weights proportionally to

their corresponding activation (Grant Sanderson, 2017c).

Additionally, we can also choose to adjust the activations of the previous layer proportional to their corresponding weights, this is a good thing to keep track of even though we cannot directly change these activations because we will need this information for the weights of the next layer back. So, for activations in the previous layer whose corresponding weight is positive, we should increase the activation, and for activations whose corresponding weight is negative, we should decrease the activations, and this would overall increase the activation of the digit two neuron. Similarly to the weights, we want to adjust activations proportional to the weights as this would give us a greater increase in the activation of the digit two neuron (Grant Sanderson, 2017c).

This only tells us how the digit 2 neuron would change the activations of the second last layer, but we need to also consider how each of the other neurons in the last layer would change these same activations, and the overall idea is the same. So, to take into account how all the neurons in the last layer would change the activations of the second last layer, for each neuron in the second last layer, we sum up the value that each last layer neuron chooses to change activation by proportional to its corresponding weight and proportional to the activation of that neuron from the last layer, and this gives us the overall desired changes that we would like to happen to the second last layer (Grant Sanderson, 2017c).

Finally, we will now see the 'back propagating' coming into play, now that we know what changes we want to happen to the second last layer, we can recursively repeat this idea of changing weights, biases to then determine the desired activation of the previous layer and so on.

Now, this is just how one input sample wishes to change the weights and biases of the network, but we still need to take into account all the other training samples. If we remember back to the section about stochastic gradient descent, we performed gradient descent on mini-batches, so here we look at each input from the mini-batch and see how each of these inputs wishes to changes all the weights and biases in the

network. Then for each weight and bias we take the average change across all the samples in the mini-batch, and these average values are roughly proportional to $-\nabla C$. So this gives us just a intuitive understanding of how back propagating works, now let us dig a little deeper into the maths behind all this.

1.7 The Math Behind Back Propagation

Let us consider a very basic example where we have a network with only one neuron in each layer and focus on the first training example, and to start off we will just focus on the connection between the last two neurons. We will call the activation of the last neuron $a^{(L)}$, the activation of the second last neuron $a^{(L-1)}$, and the desired output y, see the figure below.

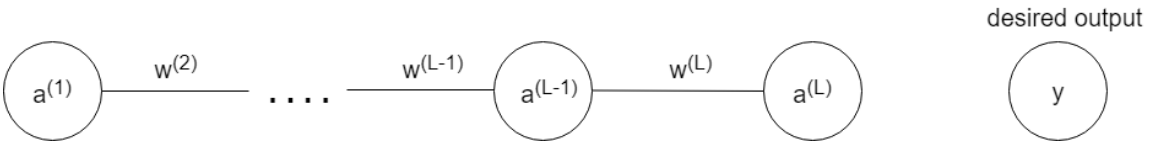

Figure 9: Network with one neuron in each layer

Denote the cost of this first training input to be

$$C_0(\boldsymbol{w}) = (y - a^{(L)})^2,$$

where as usual, $\boldsymbol{w}$ is the vector of all the weights in the network.

Note here that the cost of the $(k+1)$-th training example is C_k and is defined in the same way, additionally note that this is not to be confused with the cost function of the full network C which describes how the network behaves over all the training input, while these costs just described only tell us the behaviour of a single input.

Recall that $a^{(L)}$ is the output from the sigmoid function with the weighted sum as the input, that is,

$$a^{(L)} = \sigma(z^{(L)}),$$

where,

$$z^{(L)} = w^L a^{(L-1)} + b^L,$$

and where $b^{(L)}$ is the bias of the neuron in the last layer (Grant Sanderson, 2017a).

The aim is to quantify how much changes in different weights and biases change the cost function, and this is so that we can make the changes that cause the most decrease to the value of the cost function. So let us first look at how much the cost function changes when we change the weight $w^{(L)}$, this is mathematically represented by the partial derivative $\frac{\partial C_0}{\partial w^{(L)}}$, and since C_0 is a function of $a^{(L)}$ which in turn is a function of $z^{(L)}$ which is then a function of $w^{(L)}$ computing this partial derivative requires multiple applications of the chain rule, as follows,

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial C_0}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}$$

(Grant Sanderson, 2017a).

More explicitly, we can compute the relevent partial derivatives,

$$\frac{\partial C_0}{\partial a^{(L)}} = -2(y - a^{(L)}),$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}),$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)},$$

and hence,

$$\frac{\partial C_0}{\partial w^{(L)}} = -2(y - a^{(L)})\sigma'(z^{(L)})a^{(L-1)}.$$

From this expression we can see that a change to the weight $w^{(L)}$ changes the cost function C_0 in a way that is proportional to the activation of the previous layer $a^{(L-1)}$ which is exactly what we mentioned in the previous section describing the intuition.

Similarly, we can also see how much the cost function changes when we change the bias of the last layer by computing the partial derivative of the cost function with

respect to $b^{(L)}$, again this is done by multiple applications of the chain rule,

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial C_0}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial b^{(L)}},$$

where the first two partial derivatives are the same as the previous calculation and,

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1,$$

so,

$$\frac{\partial C_0}{\partial b^{(L)}} = -2(y - a^{(L)})\sigma'(z^{(L)}).$$

Now, we can see how changes to the activation of the neuron in the second last layer affects the cost function by once again using the chain rule to compute the partial derivative,

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial C_0}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}},$$

where the first two partial derivatives are the same as the previous calculation and,

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)},$$

so,

$$\frac{\partial C_0}{\partial a^{(L-1)}} = -2(y - a^{(L)})\sigma'(z^{(L)})w^{(L)}.$$

Here, we can see that the influence of the activation from the preceding layer on the cost function is proportional to the corresponding weight and this reflects what was mentioned in the previous section. We should note that although we cannot directly change this activation, this will help us to compute the partial derivatives of the weights and biases in preceding layers.

To see how weights and biases of previous layers affect the cost function we keep iterating through the layers applying the chain rule at each layer, for example let's quickly look at the partial derivative of the cost function with respect to $w^{(L-1)}$, the

weight of the second last layer,

$$\begin{aligned}\frac{\partial C_0}{\partial w^{(L-1)}} &= \frac{\partial C_0}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial a^{(L-1)}}\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \\ &= \frac{\partial C_0}{\partial a^{(L-1)}}\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}.\end{aligned}$$

So, we can see the idea of back propagating coming through, we start at the last layer then propagate backwards to calculate the partial derivatives for preceding layers. See the figure below for a depiction of the 'chain' for the chain rule, based on Grant Sanderson's diagram in his video 'Backpropagation Calculus'.

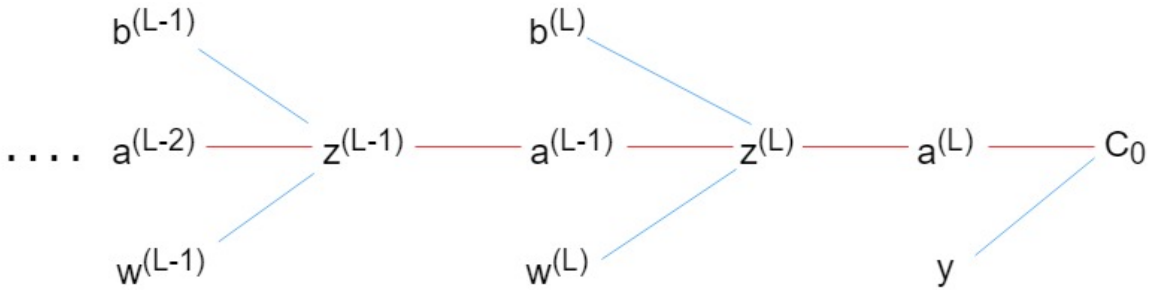

Figure 10: The 'chain' structure in the chain rule

Now, that we have the partial derivatives with respect to all weights for the first training example, we must also calculate all these partial derivatives for all the other training samples in the batch. Then recall equation (2) from the section Stochastic Gradient Descent, we can approximate the partial derivatives of the cost function of the network with the average of our costs for each training example,

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C}{\partial w^{(L)}}$$

(Grant Sanderson, 2017a).

All these partial derivatives of the cost function with respect to the weights and biases give us the gradient vector ∇C which was essential to the gradient descent algorithm.

Now, this was just the case for a network with only a single neuron in each layer.

For network with more than one neuron in each layer, the idea is very similar.

Let us consider just a single training example first. Since now we have multiple activations in one layer of the network, we need to fix our labelling scheme, now $a_i^{(M)}$ is the i-th activation from the M-th layer of the network. Suppose we have n_L elements in the last (L-th) layer of the network, then the cost function of this training example is

$$C_0(\boldsymbol{w}) = \sum_{j=1}^{n_L}(y_j - a_j^{(L)})^2,$$

where y_i is the i-th component of the expected output vector. To denote the weights, call the weight connecting the k-th neuron in layer $L-1$ to the j-th neuron in layer L, $w_{jk}^{(L)}$. Then as before

$$a_j^{(L)} = \sigma(z_j^{(L)}),$$

where

$$z_j^{(L)} = b_j^{(L)} + \sum_{i=1}^{n_{L-1}} w_{ji}^{(L)} a_i^{(L-1)}.$$

The figure below illustrates the weights and activations of layers $L-1$ and L in the network.

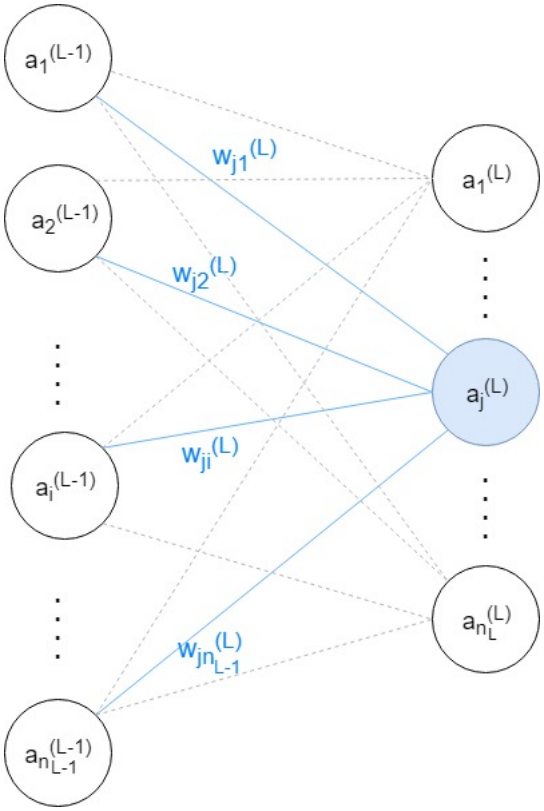

Figure 11: The labelled weights and activations of the network

Now, if we compute the partial derivative of this cost function with respect to one of the weights in the last layer $w_{jk}^{(L)}$, then this looks very similar to the previous simplified case,

$$\begin{aligned}\frac{\partial C_0}{\partial w_{jk}^{(L)}} &= \frac{\partial C_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \\ &= -2(y_j - a_j^{(L)})\sigma'(z_j^{(L)})a_k^{(L-1)}.\end{aligned}$$

Similarly, for the bias $b_j^{(L)}$,

$$\begin{aligned}\frac{\partial C_0}{\partial b_j^{(L)}} &= \frac{\partial C_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \\ &= -2(y_j - a_j^{(L)})\sigma'(z_j^{(L)}).\end{aligned}$$

However for the activations in the previous layer, the partial derivatives become slightly more complicated. If we consider the k-th activation in the $(L-1)$-th layer,

$a_k^{(L-1)}$, this activation influences all the activations in the L-th layer, so to compute the partial derivative we must sum over each activation in the L-th layer. So this partial derivative is

$$\begin{aligned}\frac{\partial C_0}{\partial a_k^{(L-1)}} &= \sum_{j=1}^{n_L} \frac{\partial C_0}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \\ &= \sum_{j=1}^{n_L} -2(y_j - a_j^{(L)})\sigma'(z_j^{(L)})w_{jk}^{(L)}\end{aligned}$$

(Grant Sanderson, 2017a).

Once we get the partial derivatives of the all activations in the $(L-1)$-th layer, we can then propagate backwards and find the partial derivatives of all the weights and biases in the preceding layers. Then general partial derivative for a weight in the l-th layer of the network is

$$\begin{aligned}\frac{\partial C_0}{\partial w_{jk}^{(l)}} &= \frac{\partial C_0}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} \\ &= \frac{\partial C_0}{\partial a_j^{(l)}} \sigma'(z_j^{(l)}) a_k^{(l-1)}\end{aligned}$$

where,

$$\frac{\partial C_0}{\partial a_j^{(l)}} = \sum_{j=1}^{n_{l+1}} w_{jk}^{(l+1)} \sigma'(z_j^{(l+1)}) \frac{\partial C_0}{\partial a_j^{(l+1)}}$$

(Grant Sanderson, 2017a)

This concludes this sub-section on back propagation and the section on basic neural networks. Now that we have a basic understanding of neural networks, let us now delve into the world of convolutional neural networks.

2 What is a Convolutional Neural Network?

Recalling back to the example of digit recognition, in the previous section we explained how one would predict the number in an image using a fully connected neural network, while this is not too bad of a strategy, we could better use the spatial features of the images. Firstly, this method would not scale well, if we have larger input images it is infeasible to maintain a fully connected network. Additionally, our previous network considered each pixel as an independent number and made no connections with surrounding pixels, it makes more sense to consider pixels together so we can see particular patterns forming like edges. Convolutional neural networks are designed for image classification and in this section we will explore the structure of these networks, however many of the fundamental ideas of neural networks like activation functions, gradient descent and back propagation remain the same (Michael Nielsen, 2019a).

2.1 Convolutions

Convolutions are used to obtain particular features from images (Prakhar Ganesh, 2019). Here, we will discuss 2-dimensional convolutions, suppose we have an input image, for our example we will use a 5 by 5 image. For the convolution operation, we need a kernel, which essentially a matrix of real values, generally much smaller than the size of the image. Different choices of kernels will have different effects on our image, such as blurring or sharpening the image, the resulting output of a convolution is called a feature.

To compute the convolution, the heuristic is to slide the kernel matrix over the input image, the amount we slide by is known as the stride, and at each step we compute something like a 'matrix dot product'. This is better illustrated in the figure

below, in this example, the stride is 1, and the kernel is the 3×3 matrix

$$K = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}.$$

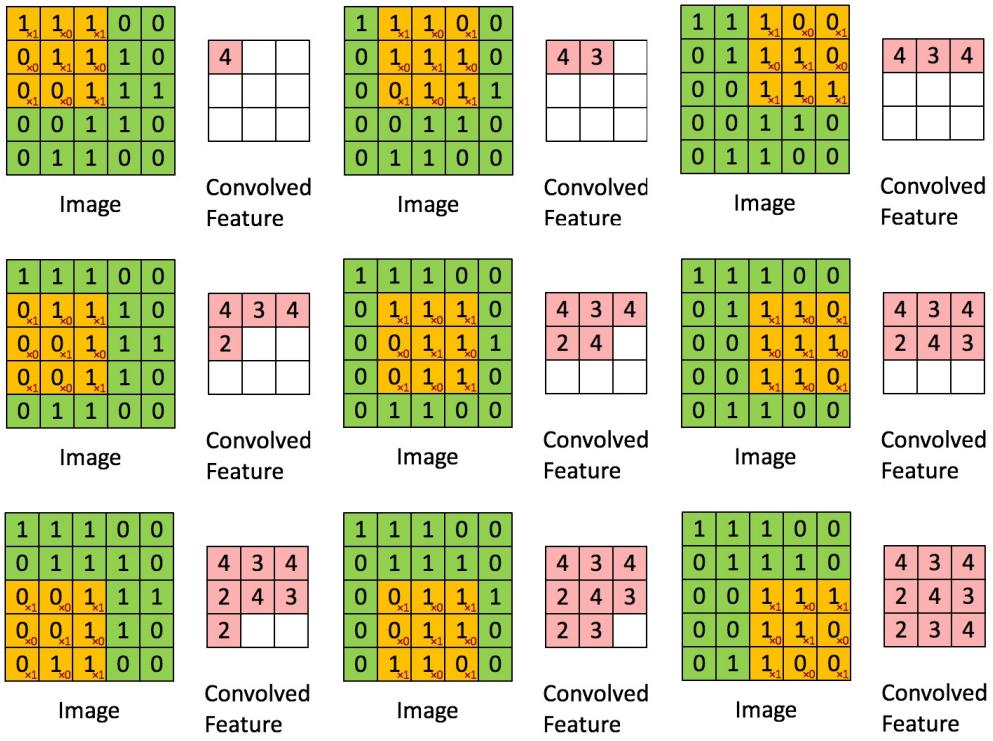

Figure 12: Example of Convolution (Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, and Sameep Tandon, n.d.)

Let us clarify the computation in each step, we can think of this operation as a 'matrix dot product', let's denote this operation as $\otimes$. Let us consider the first step of the convolution (the top right of the figure), we perform this 'matrix dot product' of the top 3×3 square of the image with the kernel matrix, and this becomes the first

entry of the resulting matrix,

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 1$$

$$= 4.$$

More generally put, we can describe this operation as,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \otimes \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} b_{ij}.$$

Let's look at an example to see how convolutional kernels are able to identify features of an image. The example we will use is from Adit Deshpande's "A Beginner's Guide to Understanding Convolutional Neural Networks".

Suppose we want to look for curves in a simple image, we would need a kernel that would be able to identify curves, this would essentially look like the type of curve you are trying to find, as in the figure below.

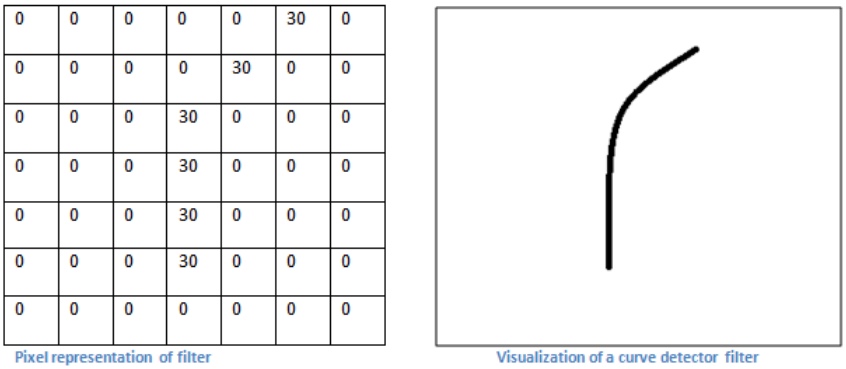

Figure 13: Kernel for Curve Detection (Adit Deshpande, n.d.)

Now let's see what happens when we apply this 7×7 kernel over different 7×7 regions of an image, here we will use a simple drawing of a mouse and we wish to first see what happens when we apply this kernel to a curved portion on the back of the

mouse.

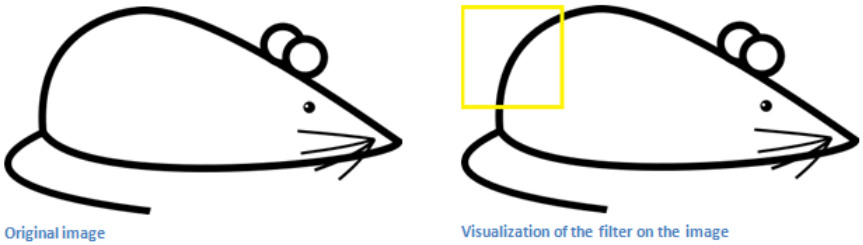

Figure 14: Image of a mouse and first portion to analyse in yellow box (Adit Deshpande, n.d.)

Now, applying the kernel over this portion of the image and calculating the result using the 'matrix dot product',

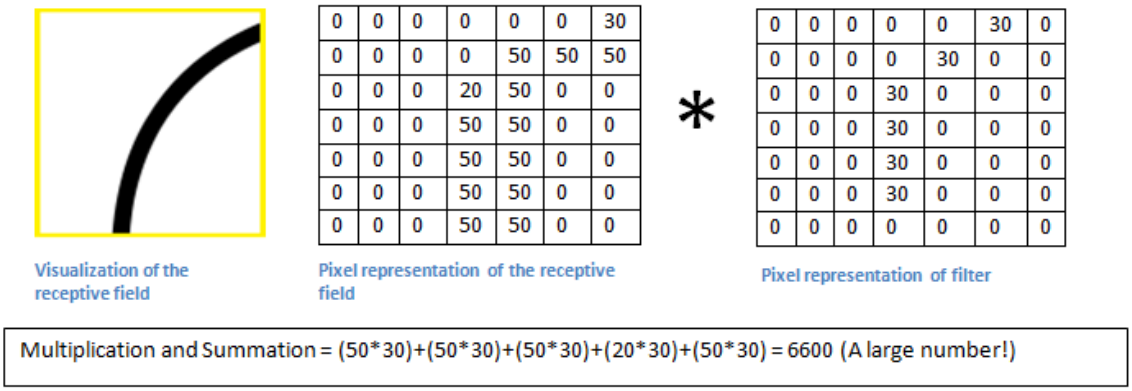

Figure 15: Applying kernel to curved image (Adit Deshpande, n.d.)

The result is 6600, which was obtained since many of the non-zero entries of the input image match up with the non-zero entries of the kernel, resulting in a large sum. Now, let us try again over a portion of the image that is not curved in this way, the ear.

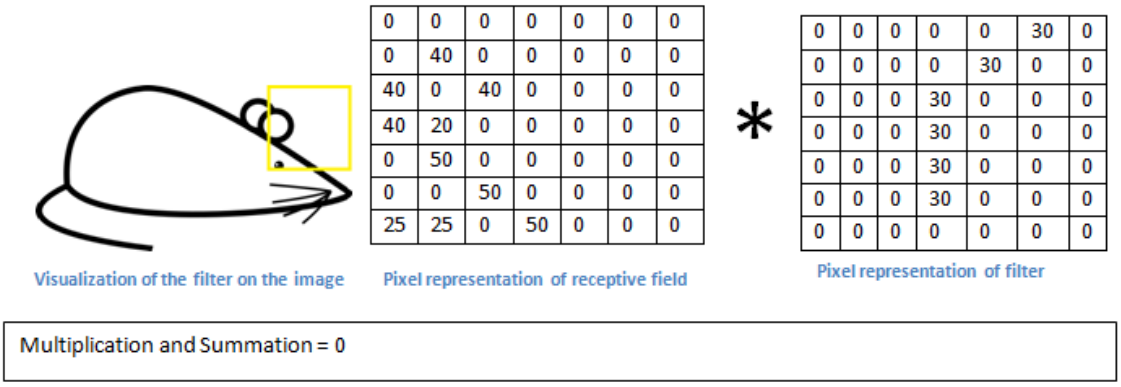

Figure 16: Applying kernel to non curved image (Adit Deshpande, n.d.)

The result is 0, and this is because none of the non-zero entries of the input image align with the non-zero entries of the kernel matrix. So if we apply the convolution over the whole image, we'll see that entries in the resulting feature matrix with large values indicate that the corresponding region in the input image displays the feature in question, and those with small values don't.

2.2 Convolutional Layer

Now, that we have a basic understanding of the convolution operation, we can apply this in the context of neural networks. To begin, let's introduce a new way of visualising layers in this new network, since the layers identify spacial aspects of an image we will view each layer as a grid of pixels, and in our example we will use a 28×28 pixel grid.

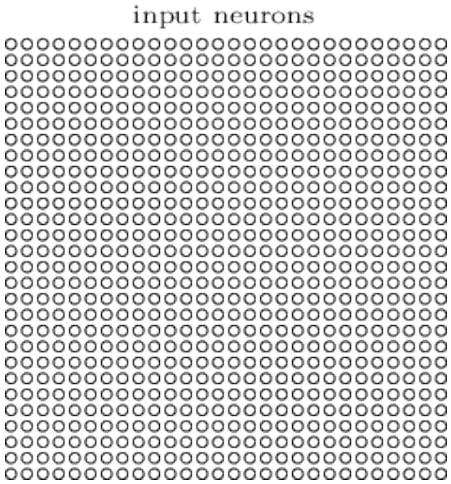

Figure 17: Input neurons in a convolutional neural network (Michael Nielsen, 2019a)

Unlike the multi-layer perceptron, we won't connect every input neuron to every neuron in the first hidden layer, instead we will connect a small, localised area of the input neurons to one neuron in the hidden layer. We can visualise this like the figure below.

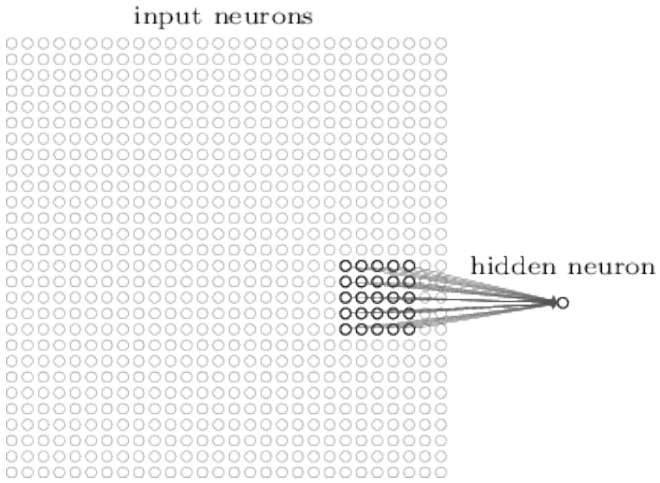

Figure 18: A 5 by 5 region of input neurons connect to the hidden neuron (Michael Nielsen, 2019a)

This region that gets connected to that hidden neuron is called the local receptive field for that hidden neuron. The way we connect the local area of input neurons to the hidden layer neuron is by using that 'matrix dot product' operation and overall this is the same as taking the convolution of the input image and connecting the section of the input image to the corresponding component of the resulting feature matrix. This type of layer is called a convolutional layer.

We can visualise this in the figures below, in these examples we have a 5×5 kernel matrix.

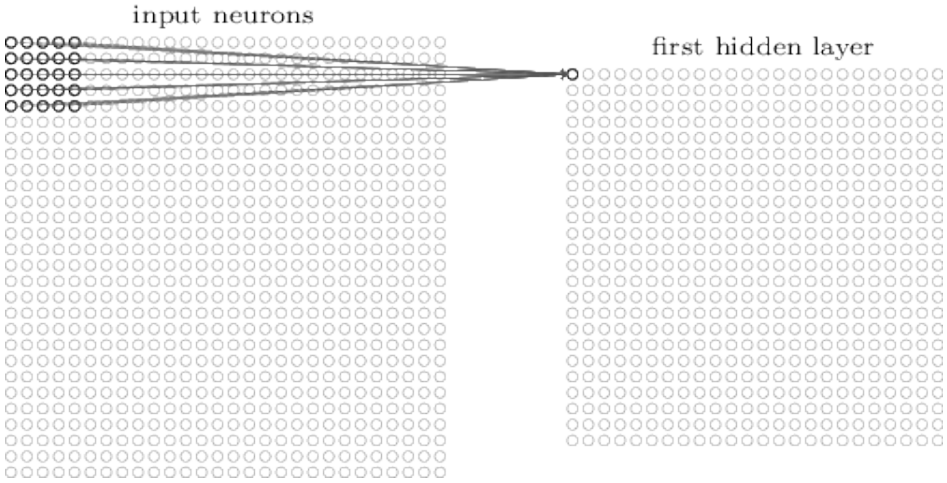

Figure 19: Sliding the 5 by 5 receptive field over input neurons (1) (Michael Nielsen, 2019a)

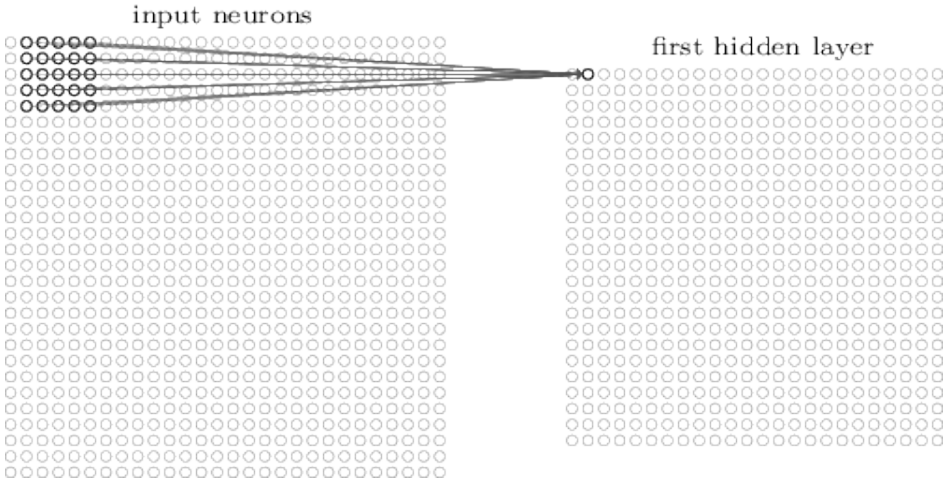

Figure 20: Sliding the 5 by 5 receptive field over input neurons (2) (Michael Nielsen, 2019a)

From the perspective of neural networks, the values in the kernel matrix are the weights of this layer, so this means that all the neurons in this layer share the same weights (or kernel) and additionally the is one bias for this layer, so we also have a shared bias. In this way the network learns significantly less weights and biases than a fully connected network and as such, this scales much better.

Like in the standard neural network, we can put the weighted sum through an activation function like the sigmoid function, however a more commonly used activation function for convolutional neural networks is the ReLU function, since this function

enables the network to train much faster. Put mathematically the activation of the hidden layer neuron in the j-th row and k-th column is

$$R\Big(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m}a_{j+l,k+m}\Big),$$

where b is the shared bias, $w_{l,m}$ is the element in the l-th row and m-th colum of the kernel matrix, and $a_{j,k}$ is the element in the j-th row and k-th column of the input matrix (Michael Nielsen, 2019a). Applying the ReLU function is often considered a layer of it's own and is simple called a ReLU Layer.

In our example, we have described a convolutional layer with only one kernel producing only a single feature, however a more complete convolutional uses multiple different kernel maps producing multiple features (Michael Nielsen, 2019a). The figure below shows a convolutional layer with three features.

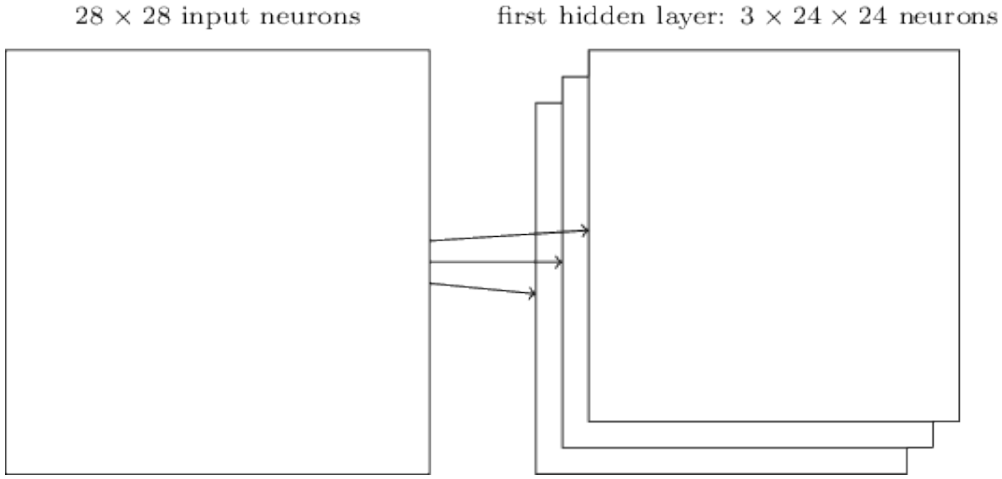

Figure 21: Convolutional layer with three kernels (Michael Nielsen, 2019a)

2.3 Pooling Layer

A pooling layer often follows the convolutional layer, it's job is to reduce the size of the layer, this is especially useful for decreasing the necessary computational power to process the data (Sumit Saha, 2018).

Simlarly to convolutions, we think of pooling as sliding a fixed size grid usually of size 2×2 over the input, and usually with a stride of 2 (Adit Deshpande, n.d.).

However instead of computing the 'matrix dot product', in the case of max pooling we take the maximum value over the portion of the input, see the figure below for a visualisation of this.

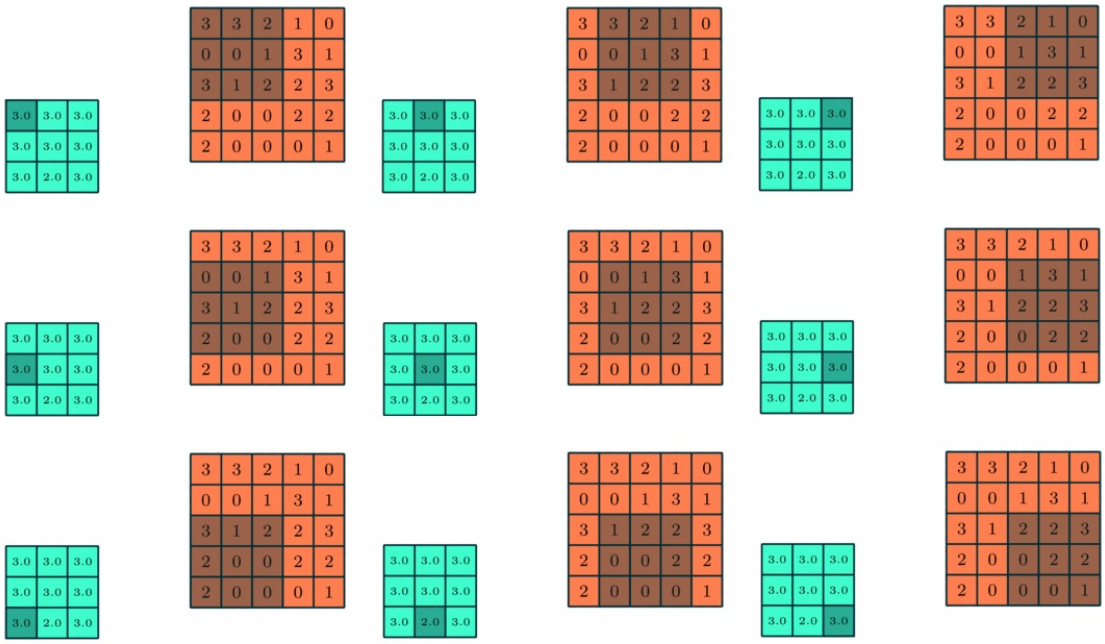

Figure 22: Example of Max-Pooling (Sumit Saha, 2018)

There are also other types of pooling such as average pooling, where the average is taken of the portion of the input data.

In our example, we had three features in the convolutional layer, so for each feature we perform max pooling, this structure is depicted in the figure below.

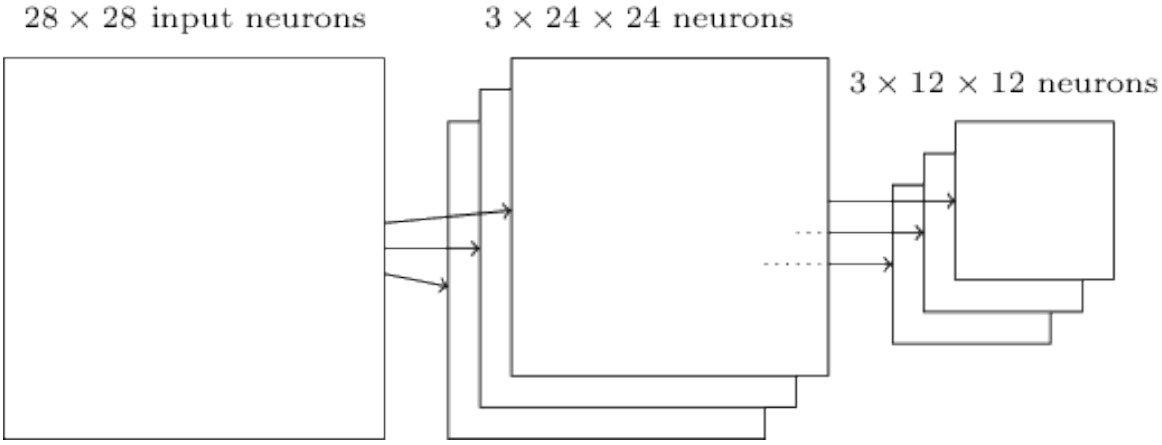

Figure 23: Example of Max-Pooling (Michael Nielsen, 2019a)

2.4 Fully Connected Layer

The fully connected layer, as the name suggest connects every neuron from one layer to every neuron in the next, this is the same as in standard neural networks. In the example of digit recognition, we will fully connect the max-pooled layer to the output layer containing 10 neurons, one for each digit. The full architecture of the network is depicted in the figure below.

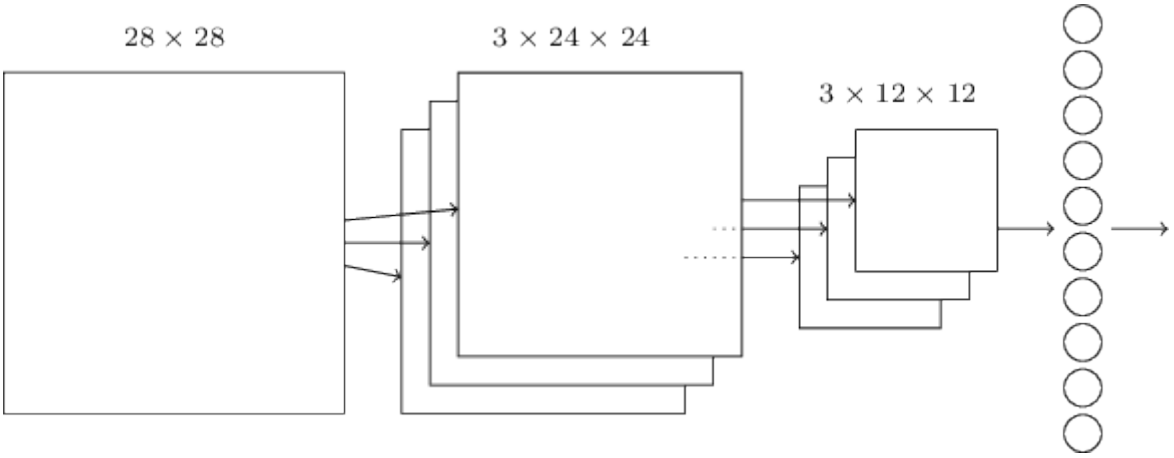

Figure 24: Full Architecture of the network (Michael Nielsen, 2019a)

Of course, this is just one particular example of a convolutional neural network, in practice we can add more convolutional and pooling layers to build a deeper network.

In conclusion, throughout this paper we looked at standard neural networks and the big algorithms involved with them including gradient descent and back propagation. We then looked at how we could extract particular features from images using convolutions, and then how we could incorporate this idea into a neural network.

References

Adit Deshpande. (n.d.). *A Beginner's Guide To Understanding Convolutional Neural Networks*. Retrieved December 8, 2019, from https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, Adam Coates, Andrew Maas, Awni Hannun, Brody Huval, Tao Wang, and Sameep Tandon. (n.d.). *Feature Extraction Using Convolution*. Retrieved December 8, 2019, from http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

Grant Sanderson. (2017a, November 3). *Backpropagation Calculus*. Retrieved December 9, 2019, from https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4

Grant Sanderson. (2017b, October 16). *Gradient descent, how neural networks learn | Deep learning, chapter 2*. Retrieved December 5, 2019, from https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2

Grant Sanderson. (2017c, November 3). *What is backpropagation really doing? | Deep learning, chapter 3*. Retrieved December 6, 2019, from https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3

Karlijin Willems. (2019, February 5). *Deep Learning in Python*. Retrieved November 26, 2019, from https://www.datacamp.com/community/tutorials/deep-learning-python/

Michael Nielsen. (2019a, June). *Deep Learning*. Retrieved December 8, 2019, from http://neuralnetworksanddeeplearning.com/chap6.html

Michael Nielsen. (2019b, June). *Using neural nets to recognize handwritten digits*. Retrieved December 3, 2019, from http://neuralnetworksanddeeplearning.com/chap1.html

Misa Ogura. (2017, December 9). *Intuition (and maths!) behind univariate gradient descent*. Retrieved December 5, 2019, from https://towardsdatascience.com/machine-learning-bit-by-bit-univariate-gradient-descent-9155731a9e30

Mohamed Zahran. (2015). *A hypothetical example of Multilayer Perceptron Network*. Retrieved November 26, 2019, from https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065

Prakhar Ganesh. (2019, October). *Types of Convolution Kernels : Simplified*. Retrieved December 8, 2019, from https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37

Roell, J. (2017, June 13). *From Fiction to Reality: A Beginner's Guide to Artificial Neural Networks*. Retrieved November 26, 2019, from https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b

Sagar Sharma. (2017, September 7). *Activation Functions in Neural Networks*. Retrieved December 8, 2019, from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Sumit Saha. (2018, November 16). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Retrieved December 8, 2019, from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Suryansh S. (2018, March 10). *Gradient Descent: All You Need to Know*. Retrieved December 4, 2019, from https://hackernoon.com/gradient-descent-aynk-7cbe95a778da

Synced. (2019, June 20). *MNIST Reborn, Restored and Expanded: Additional 50K Training Samples*. Retrieved December 3, 2019, from https://medium.com/syncedreview/mnist-reborn-restored-and-expanded-additional-50k-training-samples-70c6f8a9e9a9