Convolutional neural networks in image recognition

Freek Henstra
5725801

Universiteit Utrecht

Mathematics
Bachelor Thesis

Supervised by: Prof. dr. S. M. Verduyn Lunel

Abstract

In this thesis, we study the topic of deep learning with a focus on image recognition using convolutional neural networks. We cover the various components of deep learning, including the network structure, backpropagation and stochastic gradient descent. We explain the fundamentals of these components and compare theory to practice. We then examine convolutional neural networks and the various layers they consist of. Finally, we build and train a convolutional neural network to classify small images of coloured shapes. This network achieved an accuracy of around 85%.

Contents

1 Introduction **2**

2 Basic Theory of Deep Learning **4**
2.1 Function approximation 4
2.2 Learning 6
2.3 Overfitting 8
2.4 Layers of abstraction 8

3 Deep neural networks **12**
3.1 Feedforward neural networks 12
3.2 Forward propagation 13
3.3 Non-linearity 14
3.4 Expressivity 15
3.5 Recurrent neural networks 17
3.6 Residual neural networks 18
3.7 Restricted Boltzmann machines 19

4 Backpropagation **21**

5 Stochastic gradient descent **25**
5.1 Iteration 25
5.2 Vanishing and exploding gradients 26
5.3 Convergence 27

6 Convolutional neural networks in image recognition **30**
6.1 Experiment 30
6.2 Convolutional layers 33
6.3 Max pooling layers 36
6.4 Fully-connected layers 37
6.5 Results and analysis 38

Chapter 1

Introduction

In recent years, machine learning has become essential in numerous fields. Following various developments in recent years, several machine learning methods have proven to be more effective and efficient than traditional algorithms. Nowadays, it is practically impossible to use technology without running into some form of machine learning. These applications include predicting your next word in a text, choosing advertisements based on your interests or detecting faces when you take a picture.

Using various different machine learning approaches, algorithms can learn from large datasets by changing their parameters or structure to better fit the data. For example, given a set of images and sentences describing the images, an algorithm can learn to describe images with a sentence. If enough of these images contain cats, a successfully trained algorithm should be able to determine whether or not an image, even one it has never seen before, contains a cat.

Currently, one of the most popular machine learning methods is deep learning. This method uses an artificial neural network (ANN) as its architecture, a type of weighted directed acyclic graph. This network represents a function. Using a type of stochastic approximation, the weights of the network are altered to improve the function. In this sense, we can view deep learning as function approximation. While deep learning has been around for much longer, it was not very common until after several breakthroughs this decade.

One of these breakthroughs happened at the ImageNet [1] LSVRC-2012 contest. This contest asked contestants to submit algorithms to classify images in the ImageSet dataset, which at the time consisted of ”over 15 million labeled high-resolution images in over 22,000 categories.” [2] A team at the University of Toronto managed to win the contest with a deep learning algorithm using a convolutional neural network (CNN). [2] This was unprecedented and led to a strongly increased interest in deep learning. CNNs have since become the state of the art in image recognition.

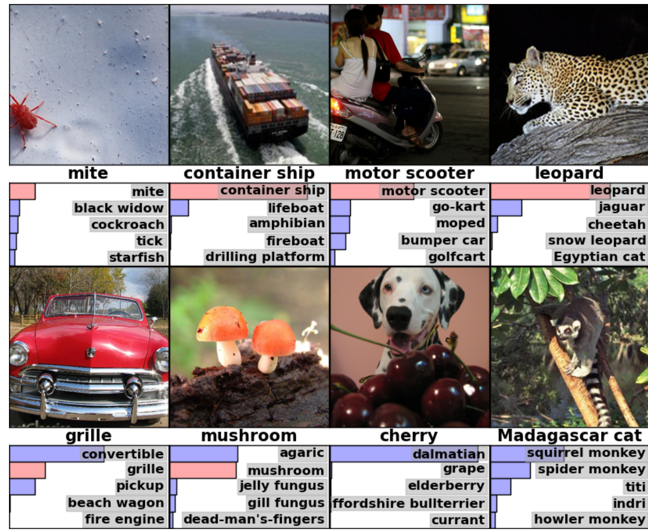

Figure 1.1: "Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5)." [2]

In the Chapter 2, we will start with the basic theory of deep learning. In the following chapters, we will explain in detail the components of deep learning: the deep neural networks (DNNs), backpropagation and stochastic gradient descent (SGD). Finally, in Chapter 6, we will focus specifically on CNNs. These networks consist of different types of layers with different functions. We will also examine a network we designed and implemented in Python using the library Keras. This network was trained to classify simple coloured shapes. Throughout the thesis, we will also take a closer look at how effective these methods are proven to be, comparing theory to practice.

Chapter 2

Basic Theory of Deep Learning

Before we dive into the details of deep learning, we will first explain the general idea of deep learning and how its various components fit together.

2.1 Function approximation

In deep learning, we build an ANN to be used as a function. More specifically, we build a DNN, but we will analyse the difference in Section 2.4. The function is not a typical mathematical function, due to the unusual input and output. In image recognition, a classification network would represent the function

$$F : \text{image} \longrightarrow \text{classification}. \tag{2.1}$$

The goal for the resulting function is to properly classify an image given as input. Alternatively, the output could simply be a category the image belongs to, or specific parts of the image, such as the locations of certain objects in the image. In machine learning, the goal is to create a function that gives the correct output given a certain input.

Suppose we have a function F_p such as above, that perfectly classifies images as intended. While it is effectively impossible to create this exact function in most cases, we are looking for a network representing an approximation F_n of F_p. After all, for any image x your network classifies correctly, we get $F_n(x) = F_p(x)$. Note that there are often many input values with no valid output. For example, a random assortment of pixels does not really have any interpretation and therefore no valid output. Similarly, a network trained for specific image classes, such as animals, has no correct output for other images. Therefore, this F_p is not well-defined. However, since we do not consider images that fall outside of the image class, we can still view deep learning as function

approximation.

An ANN is a weighted directed acyclic graph, consisting of nodes with directed connections between them, each with a certain weight. The nodes represent the neurons of biological neural networks, which send signals through the connections. The strength of the signal is dependent on the signals received by the node that sent it, as well as the weight of the connection. The network is divided into various layers, each with a set of nodes. The first layer is the input layer and the last is the output layer. The input layer contains the input nodes, which receive the input data and send it through the network. After signals have passed through the entire network, the values of the output nodes form the calculated output.

In an ANN, each input node j receives a single value $x_i \in \mathbb{R}$. Suppose there are N input nodes. The input contains one such value for each input node and is therefore a vector $x \in \mathbb{R}^N$. Suppose that there are M output nodes. Then the function of the network becomes

$$F : \mathbb{R}^N \longrightarrow \mathbb{R}^M. \tag{2.2}$$

If we compare this to our classification function (2.1), we can see that it is necessary to represent both the image and the classification as a vector. An image can be represented by a matrix of pixels. Each of these pixels has its r, g and b colour values, generally in the interval $[0, 1]$. A regular ANN would need separate input nodes for each value of each pixel. As we will see in Chapter 6, CNNs keep the matrix structure of the image intact. Regardless, the input is of the same dimensionality N. For an $m \times n$ image, we get $N = 3mn$. In image classification, there is often a finite number of possible classifications. Usually, every output node corresponds to one of these classifications.

Inputs of high dimensionality are very common in deep learning, even outside of image recognition. As a result, the functions can get very complex. Since we want to approximate such functions, it is vital that the architecture we use to build the functions can get just as complex. Fortunately, it can, because ANNs are universal approximators. [3,4] We will elaborate on this in Chapter 3.

Before we can begin learning, we need an ANN to start with. During the learning procedure, only the weights of the network will change. The structure of the network stays identical. This means we already have to build the structure of the network before we start learning. How this is done will be explained in Chapter 3. We also need to assign initial weights to the connections. Generating these weights randomly is an option, but this can lead to problems, which we will cover in Chapter 5. Another option is to do a form of pre-training to determine starting values. We will see an example of pre-training in Section 2.4.

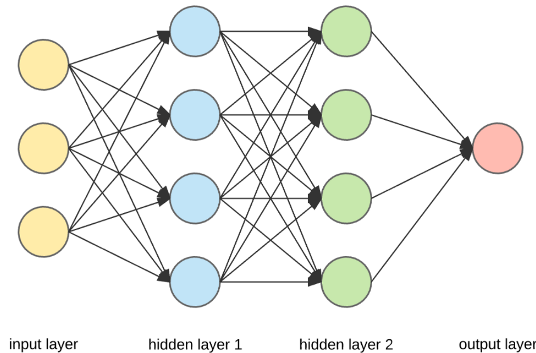

Figure 2.1: An example of an ANN with two hidden layers. The arrows represent the connections between the nodes.

2.2 Learning

In order to train a network, large datasets are used. These datasets consist of many different inputs and optionally their correct outputs. The learning algorithm is called supervised learning when the training dataset consists of input-output pairs (x, y). [5] Returning to the classification example, such a pair would consist of an image with its correct classification. This is the most common type of machine learning. When the dataset only contains inputs, the learning algorithm is instead called unsupervised learning. In this case, it might be unclear what the desired output is. If we are training an algorithm to solve puzzles or win games, then the desired output would be a solution or winning strategy, which can be verified without a given correct output. We will cover an example of unsupervised learning within image recognition in Section 2.4.

Once we have an ANN specified, we can start learning. Suppose $T = \{(x_i, y_i) \mid 1 \leq i \leq n\}$ is our training dataset containing n input-output pairs. After passing through the initial ANN using an input x, we can compare the calculated output $\hat{y} = F(x)$ to the desired output y using a loss function

$$C_0(x) = L(y, \hat{y}) \tag{2.3}$$

that, given the network and an input-output pair, outputs the error. There are several different loss functions that can be used here. For the loss function of the entire training dataset, we can simply calculate the sum of all the errors as follows:

$$C_0(T) = \sum_x C_0(x). \tag{2.4}$$

With this, the training can be seen as an optimisation problem

$$\min_{w_i \in w} (C_0(T)), \tag{2.5}$$

where w is a vector containing all weights in the network. We can solve this problem using SGD.

For SGD, the training dataset $T = \{(x_i, y_i) \mid i = 1, 2, \ldots\}$ is partitioned into smaller batches $B_i \subset T$, with $|B_i| = |B_j|$ for any indices i and j. These batches are determined at random, which is why it is stochastic. This is done by creating a random permutation of T and dividing it into partitions. Given one batch B_i, we then calculate the error of the batch as follows:

$$C_0(B_i) = \frac{1}{|B_i|} \sum_{x \in B_i} C_0(x). \tag{2.6}$$

where the inputs x are the inputs of the input-output pairs within the batch B_i. This is simply the average error of the inputs in the batch.

The training is an iterative process, with every step, one for each batch, attempting to get slightly closer to the optimum. Some steps going backwards is acceptable too, as long as the steps are generally moving in the right direction. Once we have used every batch in the dataset, and therefore the entire dataset, we have finished one epoch. We can repeat this process for many epochs. Usually, the batches are randomised again in between epochs, which is known as shuffling. Although our goal is to minimise $C_0(T)$, deep learning is non-deterministic. It can be left running for as long as needed, but actually reaching the exact optimum is in most cases effectively impossible.

Each step is a small adjustment of all weights in the network. To determine by how much we want to adjust each weight, we calculate the gradient vector $\nabla C_0(B_i)$ of the error of the batch with respect to the weights of the network. Each element of the gradient vector denotes by how much the error will increase or decrease when the associated weight increases infinitesimally. Note that this only applies to the current batch, so the gradient vector $\nabla C_0(B_i)$ is only an estimate of the gradient vector $\nabla C_0(T)$ of the entire training dataset. This, in turn, is an estimate of the "true" gradient vector, in which the error is a loss function comparing the function of the network to the theoretical function it aims to approximate.

Effectively, this gives the "direction" we need to move the weights in in order to increase the error of the batch. Since we want to decrease it, we take a step in the opposite direction. Note that a step could be too big and cause the error to increase. As long as most steps reduce the error, this is no problem. By repeating this and using different inputs from the dataset every step, we iteratively reduce the error. We will further elaborate on SGD in Chapter 5.

Because the function of the network is made by composition of smaller functions, as will be covered in Chapter 3, the gradient vector of the network can simply be calculated using the chain rule. This is known as backpropagation, because we move from the output to the input, so backwards through the net-

work. We will cover this in more detail in Chapter 4.

2.3 Overfitting

It is customary to partition the dataset into three different subsets: training, validation and testing data. When training a network, we do not want the network to learn only the provided data, even though this is the goal of the optimisation problem. Instead, we want it to learn general patterns in the data that will also apply outside of this dataset. We speak of overfitting when a network performs significantly better on the data it was trained on than on data it has not seen before.

To detect this overfitting, we make sure the network only trains using the training data. During training, after every epoch, we determine the error and other metrics (such as accuracy) for the training data alongside those of the validation data. The results from the validation data give a better indication of the quality of the network.

However, optimising a network consists of more than just the learning process. Many parameters remain the same during learning, such as which layers are used, how many nodes they have, what activation functions they use. These fixed parameters are known as hyperparameters and adjusting them can significantly improve the network. We can restart the learning process multiple times, each time with different values for hyperparameters. By using the results of the validation data, we can manually pick effective values for the hyperparameters. This can be seen as an optimisation problem of its own.

This can once again cause overfitting, however. Since we base the quality off of the validation data, we optimise the hyperparameters to specifically get high results on the validation data. This is why we use a third subset of testing data, as a final measure of quality after fully training a network. After all, this data is not used while optimising the network.

Note that partitioning the data does not solve overfitting, it simply helps detecting it. A simple solution to overfitting is to increase the size of the datasets. This forces the network to find more general patterns. Another way to combat overfitting is to reduce the complexity and depth of the network. Lowering the dimensionality, for instance, forces the network to find a representation of the data with a lower dimensionality.

2.4 Layers of abstraction

We will now get to what makes an ANN a DNN, or what makes deep learning "deep". As mentioned before, an ANN starts with the input layer and ends with the output layer. Between these layers are the hidden layers, containing

all other nodes. Generally, the layers can be ordered so that each layer receives signals only from the previous layer and sends signals only to the next layer, as we can see in figure 2.1. There are some exceptions to this rule, but we will leave those to Chapter 3. Note that this also means there are no connections between nodes within the same layer. Now, an ANN is considered a DNN if it has multiple hidden layers.

The idea behind these multiple hidden layers, rather than just having one layer with many nodes, is that they represent different layers of abstraction. Typically in image recognition, the first layer will simply represent anything that can directly be read from the input data. In image recognition, these are the pixels of the image, in which we can find edges by comparing neighbouring pixels. Later layers will start finding combinations of the previously detected edges and represent shapes, for example. Then the final layers will start piecing everything together, representing more complex objects in the image. Finally, we get the output. In the case of a classification, this will name all the detected objects in the image and how they relate to eachother.

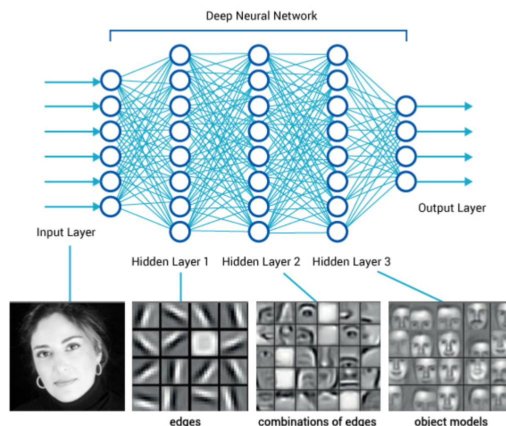

Figure 2.2: Layers of abstraction in a DNN.

These representations are all learned during the learning procedure. This is what sets deep learning apart from other machine learning methods, which would require manually engineering such representations into the architecture. Instead, deep learning simply finds this structure within the data.

In image recognition, it is possible for two completely different images to depict the same objects. In this case, you would want a classification algorithm to produce an identical output for both images. Consider two images of the same cat, but in different poses and lighting and from a different angle. The algorithm needs to be invariant to these factors. It is, however, also possible for

two very similar images to depict different objects. Consider an image of a dog with the same fur colour as the cat and the same pose, lighting and angle as one of the cat's images. Even if the images might be very similar when you only look at the pixel values, the algorithm needs to be selective of the important differences. This is known as the selectivity-invariance dilemma. By finding the structure within the data, deep learning can solve this problem.

The dimensionality of this structure can be much lower than that of the input. To further clarify this, we will take a look at an example of unsupervised learning known as an autoencoder. [6] This is an ANN with the goal of learning the identity function. That is, its desired output is the given input. This is simple as long as all hidden layers have at least as many nodes as the input and output layers. However, when at least one of the hidden layers has fewer nodes, the network will need to find a structure within the data of a lower dimensionality. Specifically, this dimensionality is the number of nodes of the smallest layer.

As an example, we will refer to [6], which features an autoencoder pre-trained using restricted Boltzmann machines (RBMs), which will be covered in Chapter 3. After this pre-training, regular deep learning was applied to the network. This autoencoder recreated 28x28 images fairly accurately, even though the smallest layer consisted of only 6 nodes. This means this autoencoder decreased the dimensionality of the data from 784 to 6 with only a small error.

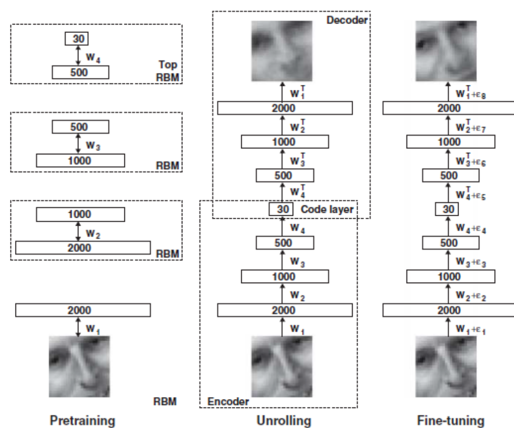

Figure 2.3: An example of a 30-dimensional autoencoder by G.E. Hinton and R.R. Salakhutdinov. [6]

This capability to find relatively simple structure in highly dimensional data is what makes deep learning so powerful. Now that we have a good idea of the concept of deep learning and its components, we will dive deeper into how exactly they work.

Chapter 3

Deep neural networks

In this chapter, we will examine the structure of the DNNs used in deep learning. We will start with the structure of a feedforward neural network and how the corresponding function is determined using forward propagation. Then, we will analyse the role of non-linearity in DNNs and the how effective DNNs are. Next, we will briefly explore two other types of DNNs: recurrent neural networks and residual neural networks. Finally, we will give a short elaboration on the previously mentioned RBMs that can be used for pre-training.

3.1 Feedforward neural networks

Recall that an ANN consists of nodes, divided into several layers. The nodes in a layer all have connections to nodes in the next layer. In a feedforward neural network, this is true for all connections. In Section 3.5 and 3.6, we will cover two types of ANNs that contain connections that break this rule. For now, we will consider only feedforward neural networks.

Recall that the network is essentially a function F. Similarly, we can view each layer l of the network as a function F_l. The function F of the entire network then simply becomes the composition $F = F_L \circ F_{L-1} \circ \cdots \circ F_2 \circ F_1$, where L is the number of layers. However, for the purpose of explaining the calculations in the network, we will not use this functional notation. Instead, we will introduce variables related to the nodes of the graph and explain their relations, which leads to a simpler notation overall.

Again, we define L as the number of layers in the network and we use $l = 1, 2, \ldots, L$ as the index for these layers. Typically, we only consider two layers at a time: layer $l-1$, which sends its output, and layer l, which receives the output of layer $l-1$ as input. We index the nodes within layer $l-1$ as $k = 1, 2, \ldots, N$. Similarly, the nodes within layer l are indexed as $j = 1, 2, \ldots, N$.

A node j in layer l receives an input $z_j^{(l)}$ and sends an activation output

$a_j^{(l)}$. The input-output pairs in the dataset contain input values x_k, each corresponding to one of the input nodes, and output values y_j, one for each of the output nodes. The output layer L outputs the values $\hat{y}_j := a_j^{(L)}$. These values are the output of the network. The weight of the connection from node k in layer $l-1$ to node j in layer l is denoted by $w_{jk}^{(l)}$. There are several other values and functions relevant to calculating the output. We will introduce those along with their notation in the next section.

3.2 Forward propagation

Since our calculations move forward through the network, from input to output, this is known as forward propagation. We will now take a look at how the output of a layer is determined.

Consider node j in layer l. For each node k in layer $l-1$ with a connection to j, it receives the activation output $a_k^{(l-1)}$. This output is multiplied by the weight $w_{jk}^{(l)}$ connecting the two nodes. After summing up all these values, we get what is known as the weighted sum. In some networks, nodes are given a bias that is added to the input. We will explain its purpose later on in this section. The bias of node j is denoted by $b_j^{(l)}$. The input $z_j^{(l)}$ of j is now computed as follows:

$$z_j^{(l)} = \sum_k a_k^{(l-1)} w_{jk}^{(l)} + b_j^{(l)}. \tag{3.1}$$

We can also view this as a matrix multiplication. Let m be the number of nodes in layer l and n the number of nodes in layer $l-1$. Let $W^{(l)} = \left(w_{jk}^{(l)}\right)$ be the weight matrix of all connections between layer $l-1$ and layer l. If two nodes j and k are not connected, simply assume $w_{jk}^{(l)} = 0$. Now $W^{(l)}$ is a $m \times n$ matrix. Let $a^{(l-1)}$ be the vector of the activation outputs $a_k^{(l-1)}$ of layer $l-1$. Let $b^{(l)}$ be the vector of the biases $b_j^{(l)}$ of the nodes in layer l. Now, the vector $z^{(l)}$ of the inputs $z_j^{(l)}$ of layer l can be written as

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}. \tag{3.2}$$

Finally, a non-linear activation function $f^{(l)}$ is applied to the input of each node j in layer l, which gives us the resulting activation output

$$a_j^{(l)} = f^{(l)}(z_j^{(l)}). \tag{3.3}$$

This is the output of layer l. If we view the layer as a function again, this function would satisfy

$$F_l\left(a^{(l-1)}\right) = a^{(l)}. \tag{3.4}$$

It appears that, in practice, the best activation function is often the rectified linear unit (ReLU), defined by

$$f(x) = max(0, x). \tag{3.5}$$

Before ReLU gained popularity, the most commonly used activation functions were sigmoids [5], such as the standard sigmoid function, defined by

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{3.6}$$

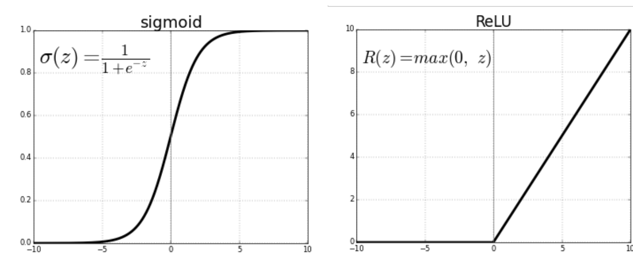

Figure 3.1: Two activation functions: the standard sigmoid and ReLU.

Now that we have seen the various components of ANNs, we will provide an interpretation for them. As mentioned in Chapter 2, we can compare the network to a biological neural network. Changing the weight of a connection increases or decreases the strength of the signals sent through it. In the receiving neuron, the strengths of these signals are added up. The ReLU function along with a negative bias represent the threshold that needs to be reached for the neuron to be activated, which causes it to send a signal. If the threshold is not reached, the ReLU function will output zero. A positive bias means that the output of the node will always at least be as strong as the value of the bias.

3.3 Non-linearity

The activation functions being non-linear is an extremely important aspect of deep learning. Linearity implies that similar inputs get similar outputs, while very different inputs get very different outputs. Recall that the selectivity-invariance dilemma, which was mentioned in Chapter 2, implies that we often want the opposite. As a result, a network built entirely from linear components is generally going to be less selective and invariant than a network with non-linear components. We call such an ANN, in which every activation function is linear, a linear ANN. As a matter of fact, we can even formulate the following theorem.

Theorem 1. *Any linear ANN can be written as an ANN with no hidden layers.*

Proof. Suppose we have an ANN in which each layer l has a linear activation function

$$f^{(l)}(x) = p^{(l)}x + q^{(l)}, \tag{3.7}$$

with $p^{(l)}, q^{(l)} \in \mathbb{R}$. Note that we apply this function to each element in the input vector $z^{(l)}$ of the layer. Define $e^{(l)} = (1, 1, \dots, 1)$, where the dimension of $e^{(l)}$ is the number of nodes in layer l. The output of layer l is now

$$\begin{aligned} a^{(l)} &= f^{(l)}\left(W^{(l)}a^{(l-1)} + b^{(l)}\right) \\ &= p^{(l)}\left(W^{(l)}a^{(l-1)} + b^{(l)}\right) + q^{(l)}e^{(l)} \\ &= p^{(l)}W^{(l)}a^{(l-1)} + p^{(l)}b^{(l)} + q^{(l)}e^{(l)} \\ &= \widetilde{W}^{(l)}a^{(l-1)} + \widetilde{b}^{(l)}. \end{aligned} \tag{3.8}$$

It follows that we can replicate the effect of the linear activation function by simply changing the weights and biases. We can now replace each weight matrix $W^{(l)}$ and bias vector $b^{(l)}$ in the network with the new $\widetilde{W}^{(l)}$ and $\widetilde{b}^{(l)}$, respectively. In this new network, setting all activation functions to the identity function $f(x) = x$ will result in a network functionally identical to the original. Then, the output of layer l becomes

$$\begin{aligned} a^{(l)} &= \widetilde{W}^{(l)}a^{(l-1)} + \widetilde{b}^{(l)} \\ &= \widetilde{W}^{(l)}\left(\widetilde{W}^{(l-1)}a^{(l-2)} + \widetilde{b}^{(l-1)}\right) + \widetilde{b}^{(l)} \\ &= \left(\widetilde{W}^{(l)}\widetilde{W}^{(l-1)}\right)a^{(l-2)} + \left(\widetilde{W}^{(l)}\widetilde{b}^{(l-1)} + \widetilde{b}^{(l)}\right) \\ &= \widetilde{W}^{(l,l-1)}a^{(l-2)} + \widetilde{b}^{(l,l-1)}. \end{aligned} \tag{3.9}$$

Using this, we can skip layer $l-1$, instead going straight from layer $l-2$ to l, using the new weight matrix $\widetilde{W}^{(l,l-1)}$ and replacing the bias vector $\widetilde{b}^{(l)}$ with the new $\widetilde{b}^{(l,l-1)}$. This once again results in a functionally identical network. Through induction, we can now continue skipping layers until there are no hidden layers left. It follows that we can write any ANN with only linear activation functions as an ANN with no hidden layers. □

This shows that in a linear ANN, new layers add nothing of value to the network, apart from an increase in number weights. In a linear DNN, the effective number of parameters is much smaller than the actual number of parameters in the network. Therefore, the optimisation becomes a degenerate problem. Despite this, there is some merit in studying linear DNNs, which we will clarify in Chapter 5.

3.4 Expressivity

Now, we will take a look at what types of functions ANNs are capable of creating. This is called the expressivity of a network. [7] It has been proven that ANNs are extremely expressive. Even with just a single hidden layer, an ANN can approximate any continuous function to any desired accuracy. [3, 4] This is known as the universal approximation theorem and ANNs are considered universal approximators. The theorem does have several conditions, such as the

activation functions being sigmoidal.

When we add a second hidden layer, networks using other activation functions have been proven to universally approximate large classes of functions as well. For instance, networks with two hidden layers using the ReLU activation function can universally approximate Lipschitz continuous functions, again with several conditions. [8]

While ANNs might not be proven to be universal approximators under all circumstances, the expressivity of ANNs is clearly very high, as we can see in both theorems and practice. There are still a couple of important differences between the two, however.

One of these differences is that the universal approximation theory applies to ANNs in which the width of any hidden layer is unbounded. Although the network only has one hidden layer, the number of nodes necessary to approximate a function with great accuracy could be far more than realistically possible. When we have a bounded network size, these theorems no longer apply fully.

To measure the expressivity of an ANN of bounded size, we can use the VC dimension defined by Vladimir Vapnik and Alexey Chervonenkis. [9] We can determine the VC dimension for any classification algorithm. The VC dimension of such an algorithm is the maximum number of inputs it can shatter. In this context, this means to perfectly classify all inputs, regardless of their correct outputs.

For a simple example, consider a 2D plane of points with one of two possible classifications. Now consider the classification algorithm of drawing a straight line in the plane to partition the points into two subsets, each corresponding to a different classification. As we can see in figure 3.2, most sets of three points can be shattered. As long as the three points are not in a straight line, you can draw a line to separate the two classifications, regardless of the correct classification of the points.

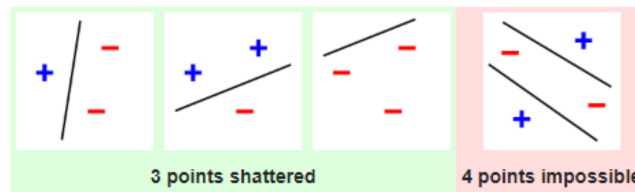

Figure 3.2: Shattering points in a 2D space using a straight line. This is only possible with up to three points.

When we add an extra point, this becomes impossible. For any set of four points, we can assign the correct classifications in a way that no straight line can

correctly separate the two classifications. Trivially, the same applies to larger sets of points. We can conclude that the VC dimension of the straight line algorithm is 3.

The VC dimension of an ANN is more difficult to determine. For an image classification network, the question becomes: what is the largest set of images that we could classify with perfect accuracy, for any assignment of correct classifications, by changing only the parameters of the network. Note that the VC dimension of an ANN is based purely on the architecture. The weights of the network and how it learns are irrelevant.

While there is no known way to determine the exact VC dimension, there are proven lower and upper bounds for the VC dimension of specific types of ANNs. [10] While the specific bounds are dependent on various factors such as the activation functions, they are generally of quadratic order with respect to the number of parameters and/or the number of nodes.

Another difference between theory and practice so far is the depth of networks. The universal approximation theorem implies one hidden layer is sufficient. Even when we take the VC dimension of a network into account, we can simply add more nodes in the one hidden layer to increase it, which should therefore improve the expressivity. However, practice shows that deeper networks, some obstacles aside, generally perform significantly better. It seems close to impossible to replicate the incredible results of DNNs with only one hidden layer, despite such a network being a universal approximator.

A recent article proves the existence of functions that can be accurately approximated using a DNN with 3 hidden layers, while a network with only 2 hidden layers needs a width exponential in the input dimension to achieve similar accuracy. [8] Although the theorem is very specific in the constraints for the function, it is not difficult to imagine that it would generalise to more functions and more hidden layers. This would provide an explanation for the enormous advantages of deeper networks in practice. Shallow networks might be universal approximators, but deeper networks seem to achieve the same with a much smaller network size.

3.5 Recurrent neural networks

When we apply deep learning in image recognition, we build a network that takes one image as input and gives one output, such as a classification. When we want to classify multiple images, we simply pass these through the network separately, one at a time. However, sometimes it might be desirable to take previous images into account. For example, when we want to classify what happens in a video, we can input each frame separately, but it would be far more effective to compare a new frame to the frames that came before it. This approach allows detection of motion, which is otherwise impossible.

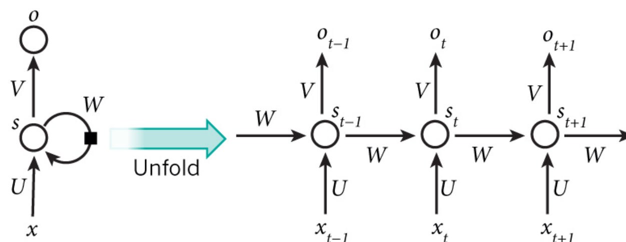

Figure 3.3: A recurrent neural network. The inputs x_i pass through the network to create an output o_i, while also updating the state for the next pass through the network.

Recurrent neural networks are built to input and/or output sequential data. These networks have been very effective in language-based applications like speech recognition and translation. By taking into account what words the network heard before, it can rule out words that sound similar to the audio, simply because they do not fit in the sentence. Similarly, in translation, keeping track of the context of the sentence makes for a better translation than one done word by word.

The way a recurrent neural network works, is that it keeps track of a hidden state vector. To explain this concept, we will assume both the input and output are sequential. In our first pass through the network, we do not yet have a hidden state, so we set it to zero. After reaching the end, we will have our first output. Instead of ending here, we use the output to update our hidden state. This hidden state then becomes part of the input, along with the next part of the input sequence, of a new copy of the network. Using this hidden state, the network becomes aware of the context of previous parts of the sequence. It then uses this to create a more reliable output for this part of the input sequence. This process continues until the entire input sequence has gone through the network and the entire output sequence is determined.

If we have an input or output that is not sequential, this method still works. In the case that only the input is sequential, we simply do not output anything other than the hidden state until the input sequence is exhausted. If only the output is sequential, the hidden state completely replaces the input of the network.

3.6 Residual neural networks

Three years after the first victory of CNNs at the ImageNet LSVRC-2012, a new team from Microsoft Research emerged victorious at the ImageNet LSVRC-2015 by making one key change to the structure of their neural network. [11] Using a residual neural network, they achieved an error of only 3.57%.

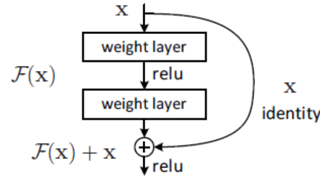

Figure 3.4: The skip in a residual neural network.

Where recurrent neural networks take a step back to the start of the network, residual neural networks instead skip ahead every other layer. Suppose layer l is receiving an input x. This input x is then also sent forward through a shortcut connection. Layer l then passes its output on to layer $l+1$. Suppose the output of layer $l+1$ is $F(x)$. The x we previously sent forward is added to the output here. Now, the input for layer $l+2$ is $F(x)+x$.

This building block can be repeated many times throughout the network. Note that this extra connection never changes during learning. It is simply the identity function.

The reasoning behind this choice, is that the researchers noticed that, while increasing the depth generally allowed the network to find more complex structures and reduce the error, adding too many layers actually caused the error to increase. By adding a shortcut connection with the identity function, the network can effectively decrease its depth by setting all weights to zero in the layers skipped by a shortcut connection.

Because of this, it should never be disadvantageous to increase the depth of the network. The network the researchers entered into the competition had an unprecedented depth of 152 layers.

3.7 Restricted Boltzmann machines

In the previous chapter, we covered an autoencoder that used RBMs for pre-training. As we can see in the top left of figure 2.3, the RBMs are small two-layer networks with undirected connections between the layers. These RBMs correspond to parts of the autoencoder.

Effectively, these RBMs are small autoencoders with one visible layer and one hidden layer, with the exception that each connection is undirected. They encode the input data in a lower dimensionality and then convert it back to the input shape. These RBMs can be trained separately through unsupervised learning and then stacked together to create a larger autoencoder including all the weights set during the training of the RBMs. Afterwards, the autoencoder can be trained as a whole to further improve the network.

RBMs can also be used to build deep belief networks. In these networks, "the top two hidden layers form an *undirected* associative memory [...] and the remaining hidden layers form a directed acyclic graph that converts the representation in the associative memory into observable variables such as the pixels of an image." [12] Similarly to autoencoders, these networks can be trained using unsupervised learning, one layer at a time.

Chapter 4

Backpropagation

Now we have moved through the network forwards, we can move backwards through the network to compute the gradient vector we need for SGD. This gradient contains the partial derivatives of the error with respect to each of the weights in the network. In a network with biases, the gradient also includes the partial derivatives with respect to the biases. As mentioned before, backpropagation is an application of the chain rule.

For this explanation, we will use the least squares error as our loss function, since it is simple and commonly used. [13] Given an input vector x, we define $y(x)$ as its desired output vector, and $\hat{y}(x)$ as the output of the network. For each input x, we determine the loss by square the Euclidean distance between the vectors $y(x)$ and $\hat{y}(x)$. The loss function $C_0(x)$ of this input x is defined by

$$C_0(x) = \frac{1}{2} \left\| y(x) - \hat{y}(x) \right\|^2 . \tag{4.1}$$

We multiply it by $\frac{1}{2}$ to cancel out the factor 2 that appears after differentiation, which we will do for every layer during backpropagation.

For the error of the batch, we now average the losses of all inputs in the batch. Let n be the number of inputs. This gives us the loss function

$$C_0(B_i) = \frac{1}{2n} \sum_{x \in B_i} \left\| y(x) - \hat{y}(x) \right\|^2 . \tag{4.2}$$

Now we can start computing the gradient vector. We want to compute

$$\frac{\partial C_0}{\partial w_{jk}^{(l)}}$$

for each weight $w_{jk}^{(l)}$ and

$$\frac{\partial C_0}{\partial b_j^{(l)}}$$

for each bias $b_j^{(l)}$. To simplify the formulas, we will assume we have only one input-output pair from the dataset, rather than a batch. Since we take the average of all the errors, we can do the same for all the gradients we calculate. This follows directly from the fact that differentiation is a linear operator.

We start with the weights of the connections between layer $L-1$ and the output layer, which is layer L. Note that during forward propagation, we determine the input $z_j^{(L)}$ of a node j in the output layer using the weights $w_{jk}^{(L)}$ of all connections to node j. We then use $z_j^{(L)}$ to determine the activation output $a_j^{(L)}$. Finally, we use $\hat{y}_j = a_j^{(L)}$ for all nodes j in the output layer to compute the loss function C_0. Clearly, we are looking at the composition of three functions, assuming we start with the output of layer $L-1$. We can now apply the chain rule as follows:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}. \tag{4.3}$$

We can now easily determine each of these partial derivatives. Looking back at equation (3.1), we get

$$\frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} = a_k^{(L-1)}. \tag{4.4}$$

The next partial derivative is simply the derivative of the activation function $f^{(L)}$. In the case of ReLU, this derivative is not well defined in $x=0$. In practice, it suffices to define the derivative as either 0 or 1 in this point. This gives

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = f^{(L)'}\left(z_j^{(L)}\right). \tag{4.5}$$

The third factor can be determined using equation (4.1), since we assume we have only one test input. Using the definition of the Euclidean distance, we obtain

$$\frac{\partial C_0}{\partial a_j^{(L)}} = \hat{y}_j - y_j. \tag{4.6}$$

We can now give the complete expression of our first elements of the gradient, which is

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = a_k^{(L-1)} f^{(L)'}\left(z_j^{(L)}\right)(\hat{y}_j - y_j). \tag{4.7}$$

For the biases, we only need to make a slight adjustment to equation (4.3). Since the bias $b_j^{(L)}$ is also used to determine the input $z_j^{(L)}$, we can simply replace the first factor with the new partial derivative

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}.$$

Note that from equation (3.1) we find that this equals 1. We get

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}} \tag{4.8}$$

$$= f^{(L)\prime}\left(z_j^{(L)}\right)\left(\hat{y}_j - y_j\right). \tag{4.9}$$

Now we are done with the output layer, we can start moving backwards to the layers before it. Assume we are on layer l. For both the weights and biases, we can respectively use equation (4.3) and equation (4.8), with L substituted by l. While the first two factors in each of the equations are determined identically to before, again substituting L by l, the third factor

$$\frac{\partial C_0}{\partial a_j^{(l)}}$$

changes. Before we find an expression for this, we can give the generalised expressions for both types of gradient elements, which are

$$\frac{\partial C_0}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}}, \tag{4.10}$$

$$\frac{\partial C_0}{\partial b_j^{(l)}} = f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}}. \tag{4.11}$$

Note that layer 1 does not have biases and $a_k^{(1)} = x_k$ for each node k.

When we are in a layer $l-1$, we cannot use equation (4.6) to determine the partial derivative

$$\frac{\partial C_0}{\partial a_k^{(l-1)}}.$$

Similar to the bias, we can determine this by slightly changing equation (4.3). We once again substitute L by l. Since we go backwards through the network, we will already have determined

$$\frac{\partial C_0}{\partial a_j^{(l)}}.$$

Now, since our activation output $a_k^{(l-1)}$ is usually sent to multiple, if not all, nodes in layer l, we have to calculate the contribution of $a_k^{(l-1)}$ to C_0 via each of the inputs $z_j^{(l)}$ in layer l. We do this by calculating the sum

$$\frac{\partial C_0}{\partial a_k^{(l-1)}} = \sum_j \frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C_0}{\partial a_j^{(l)}}. \tag{4.12}$$

We can now return to equation (3.1) to easily determine the first factor in this equation, which gives

$$\frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} = w_{jk}^{(l)}. \tag{4.13}$$

We can now complete equation (4.12), which is now

$$\frac{\partial C_0}{\partial a_k^{(l-1)}} = \sum_j w_{jk}^{(l)} f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}}. \tag{4.14}$$

Note that we can recursively unpack this equation until we reach the output layer, which is determined using equation (4.6).

To summarise, we propagate backwards through the network, using the equations (4.10) and (4.11) to calculate the elements of our gradient vector. The final factor in each of these equations is calculated using equation (4.6) in the output layer and then using the recursive equation (4.14) in all preceding layers. This gives us the gradient vector of one input-output pair from the dataset.

For convenience, the final algorithm is as follows: calculate the gradient elements using

$$\begin{aligned}\frac{\partial C_0}{\partial w_{jk}^{(l)}} &= a_k^{(l-1)} f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}},\\ \frac{\partial C_0}{\partial b_j^{(l)}} &= f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}}\end{aligned}$$

where the final factors are determined recursively using

$$\begin{aligned}\frac{\partial C_0}{\partial a_j^{(L)}} &= \hat{y}_j - y_j,\\ \frac{\partial C_0}{\partial a_k^{(l-1)}} &= \sum_j w_{jk}^{(l)} f^{(l)\prime}\left(z_j^{(l)}\right) \frac{\partial C_0}{\partial a_j^{(l)}}.\end{aligned}$$

We can repeat this for many other input-output pairs and simply take the average of all the gradient vectors. This resulting gradient vector is then used for SGD.

Chapter 5

Stochastic gradient descent

In this chapter, we will examine how SGD works. We also explain the vanishing and exploding gradient problems and how to solve them. Finally, we will look at the effectiveness of this type of learning.

5.1 Iteration

The algorithm of SGD is actually very simple. It is an iteration where the weights and biases of the network are changed every step. The step size, which is how much the weights and biases are changed every step, is affected by the learning rate η. Each step, we randomly pick a batch from the dataset and feed it to the network. Then, we use backpropagation to calculate the average gradient vector. Then, we update the weights of our network as follows:

$$w := w - \eta \nabla C_0, \tag{5.1}$$

where w is a vector containing all weights and biases and ∇C_0 is the gradient vector we receive from backpropagation. The order of the elements within these vectors is not important, as long as they are consistent with each other.

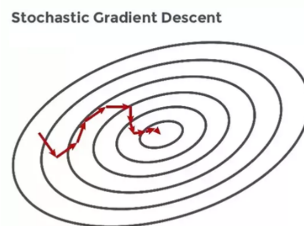

Figure 5.1: Steps towards a local minimum with SGD.

Because the gradient tells us how much the error would change if we were to increase or decrease the weights, we know that, for each of the weights, taking a small enough step in the opposite direction of its derivative will decrease the error. Since we do this for all of the weights, it is very likely that the average error of the tested batch dataset will decrease as well. By repeatedly iterating over new random batches and repeating this over multiple epochs, the network should converge to a local minimum.

It is worth noting that the idea behind SGD is to only pick one random input-output pair from the dataset every step. This is still likely to converge to a minimum, despite the higher chance of taking steps in the wrong direction. However, in practice, picking a larger batch each step is far more efficient. The main reason for this is that the GPUs used to perform the calculations, can do numerous identical calculations in parallel. This allows efficient simultaneous computations for forward and backpropagation.

There are a couple of optional hyperparameters besides the learning rate η: the momentum β and the weight decay λ. The momentum causes each step to retain some of the direction of the previous step, in order to create a smoother path to the minimum. The weight decay causes the weights to slightly decrease every step. With these additions, it becomes necessary to index our weight vectors w^i and introduce a new step vector v^i. Note that these indices do not denote the elements of the vectors, but rather the index of the step in the SGD optimisation. We now update the weights as follows:

$$v^{i+1} := \beta v^i - \lambda \eta w^i - \eta \nabla C_0^i, \tag{5.2}$$

$$w^{i+1} := w^i + v^{i+1}. \tag{5.3}$$

Finally, the learning rate can be changed during learning. This can be done manually, where it is usually lowered whenever the learning stops improving. This allows for smaller steps that might make it deeper into the local minimum. Alternatively, it be lowered automatically using a learning rate decay. This, however, runs the risk of ending too soon. To solve this problem, some adaptive learning methods exist that alter the learning rate based on the learning results.

5.2 Vanishing and exploding gradients

Two issues that SGD faces are vanishing and exploding gradients. Recall the expressions for the elements of the gradient in equations (4.10) and (4.11). Both of these contain a partial derivative that is recursive through equation (4.14).

Because of this recursion, all of the gradients in layers to the right of a certain layer, factor into the gradient of that layer. If the initial weights are not chosen carefully, it is fairly likely for gradients to exponentially increase or decrease as we move from output to input. As a result, the gradients close to the input might be very small, which we call a vanishing gradient. Otherwise,

it might be very big, which we call an exploding gradient.

The issue with a vanishing gradient, is that during the iteration, the changes made to the weights closer to the input of the network, are minimal. If the network is not already close to a local minimum, it is now unlikely to reach one, because the weights will barely change.

With an exploding gradient, we get the opposite problem. Every step in the iteration, the weights closer to the input will change drastically, while the weights closer to the output stay relatively similar. This makes the network unlikely to converge to a local minimum as well.

For networks such as autoencoders and deep belief networks, a solution to these problems is to pre-train the network using RBMs, as we mentioned in Chapter 3. By pre-training the network, its weights should be closer to the end result. Vanishing or exploding gradients are less likely to occur when the weights are already sensible. Even if they do still occur, the impact is not nearly as high.

For other types of networks, using RBMs to pre-train is not an option. Fortunately, it turns out that simply using the ReLU activation function helps against the vanishing and exploding gradient problems. Combined with general improvements to computation speed and several simple methods to initialise reasonable weights, these problems are no longer that problematic.

5.3 Convergence

While it might be evident that SGD works in practice, an insight into the reason it works is not only interesting, but could be very helpful in improving the method. Since SGD relies on randomised batches, it can only give a rough estimate of the true gradient. In practice, this does not appear to be much of an issue. Although it runs the risk of taking steps in the wrong direction, the loss function does appear to converge to the minimum over time. This convergence is not merely a coincidence, as it has been proven using statistical learning theory. [14]

Similar to ANNs, statistical learning theory studies learning machines that can implement functions that can minimise a loss function using independently drawn random inputs with their corresponding outputs. In practice, these inputs are often not truly random and there is no infinite supply of inputs, which means the theory does not entirely apply. By building sufficiently large datasets that are representative of the types of inputs we want to train the network to understand, we can increase the similarity to the theory and therefore increase the likelihood of convergence. When the dataset is sufficiently shuffled, SGD has been proven to converge. [15]

A common problem in optimisation is getting stuck in a local minimum with

a significantly higher error than the global minimum. This would logically be especially worrying for a method such as SGD, that uses its gradient to determine the direction. In such a local minimum, the calculated gradients would likely be close to zero, making the network unable to leave the local minimum. In practice, however, this does not seem to be an issue.

It is common to study linear DNNs instead of non-linear DNNs. Linear DNNs are not interesting in practice, since they are effectively shallow, as we saw in Theorem 1. However, they can be analysed quite effectively using linear algebra. In comparison, the theoretical understanding of non-linear DNNs is much worse. Despite the large differences in expressivity, there are several similarities between training non-linear and linear DNNs in practice. [16]

The optimisation of a linear ANN is much easier than that of a non-linear ANN. By using Theorem 1, we can represent the ANN with the function

$$F(x) = Wx + b. \tag{5.4}$$

When we use the standard least squares error as the loss function, the optimisation problem is to minimise

$$C_0(T) = \sum_{x \in T} ||y(x) - (Wx + b)||^2, \tag{5.5}$$

where the parameters are all elements of W and b. For each input x, we can define the matrix equation

$$Wx + b = y(x). \tag{5.6}$$

We can view this as a system of linear equations for each input x. We can add all these linear equations together into one system of linear equations of the entire dataset. We can now place all parameters, the weights and biases, into a vector w as we have done before. This allows us to write the system of linear equations of the entire dataset as a matrix equation with the elements of w as the variables. We call the matrix M and the result γ, which is a vector containing all elements $y(x)_i$ of every desired output $y(x)$. Now, the matrix equation is

$$Mw = \gamma. \tag{5.7}$$

A solution of this equation would give us the weight vector w for which the total error $C_0(T)$ is 0. Since the expressivity of a linear DNN is low, a sufficiently large training dataset could mean there is no such solution. Solving such a matrix equation using singular value decomposition solves this problem. [17] This method finds a solution if there is one. If no such solution exists, it instead finds a vector w for which the least squares error is minimal. Since this is our loss function $C_0(T)$, this method simply gives us an optimal solution. In the case of multiple solutions, it returns the vector w with the shortest length. We can even obtain the whole solution space, since it gives the orthogonal vectors that

span this space.

Even if one would opt to optimise a linear DNN using SGD instead, this should go fairly well because of the following attribute.

Theorem 2. *In a linear ANN, every local minimum is a global minimum. [18]*

This can be proven through linear algebra. Although this has not been proven for non-linear DNNs, it does appear to be mostly true in practice. Local minima with higher values than the global minimum seem to be quite rare in the high dimensional spaces often found in deep learning. Instead, most critical points are in fact saddle points. [19]

The overabundance of these saddle points in the function space pose a bigger threat for SGD. Although the saddle points can be escaped by moving in the direction of decreasing error, this can prove difficult. This is due to the saddle points often being part of larger plateaus, where the downward slope we are looking for is very subtle. This would not be an issue if the true gradient could be used, but since we are approximating it through random batches of a finite dataset, the slope might become unnoticeable. Because of the high dimensionality of the function space, randomly finding the correct dimension is unlikely.

Given a sufficiently large and representative dataset, SGD should still find a way out of these saddle points and converge to the local and therefore global minimum. However, the saddle points could still significantly slow down the learning. The saddle-free Newton method is a promising alternative to SGD designed specifically to counter this problem. [19] However, SGD-based methods are still the most commonly used in deep learning.

Chapter 6

Convolutional neural networks in image recognition

In this chapter, we will examine the type of DNN used in image recognition: the CNN. We will showcase our image recognition experiment, in which we built a CNN in the Python library Keras. We will then cover the various types of layers that make up such a network and how we implemented them in our experiment. Finally, we will analyse the results of our implementation.

6.1 Experiment

For our experiment, we built our own dataset in C#. While there are many large datasets available we could have used, such as ImageNet, classifying these images requires a well-constructed and well-trained network. We could not build such a network due to time and hardware constraints, so we decided to create a simple dataset for which a relatively shallow network would suffice.

Our dataset consists of 100,000 32×32 images of coloured shapes against a black background. The dataset is split into a training dataset of 60,000 images and validation and testing datasets of 20,000 images each. In one image, there are 1 to 4 shapes, each of which has the same colour and shape. The possible colours are red, blue, green, yellow and white. The possible colours are circles, triangles, squares, pentagons and hexagons. Note that all of these polygons are regular. The shapes have randomised positions and sizes. By enforcing a minimal distance between the centers of shapes, we ensured that only minor overlap was possible between shapes. The individual pixel values in each image were given a small deviation as well, to add a grainy filter to the images.

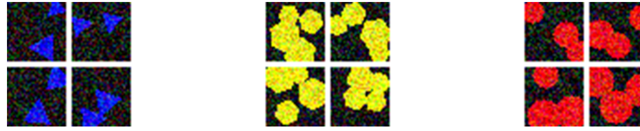

Figure 6.1: Images belonging to three of the classifications in the dataset: "2 blue triangles", "4 yellow hexagons" and "3 red circles".

The goal of the experiment is to train a network to properly classify the number, colour and shape. Since there are 4 possible numbers, 5 colours and 5 shapes, there is a total of $4 \cdot 5 \cdot 5 = 100$ classifications the network should be able to distinguish accurately. We could also train separate networks for number, colour and shape to achieve the same result. However, since one of the biggest strengths of deep learning is finding such structure in the data, this would be less efficient.

The input of our network is a 32×32 image, with each pixel containing three values: r, g and b. In a regular ANN, this would make for a total of $32 \cdot 32 \cdot 3 = 3,072$ homogeneous nodes. In a CNN, however, the image structure is kept intact. We view each of the three colour channels as one node in the form of an image.

The output of the network is a 100-dimensional vector, with each element of the vector corresponding to the predicted probability of a specific classification being correct.

The structure of our network is shown in figure 6.2. We start off with two convolutional layers, followed by a max pooling layer. We repeat this set of layers twice, with the last set of layers containing three convolutional layers instead. Finally, we have a flattening layer and two fully-connected (dense) layers. The network has a total of 677,612 parameters, which consist of the kernel elements, weights and biases.

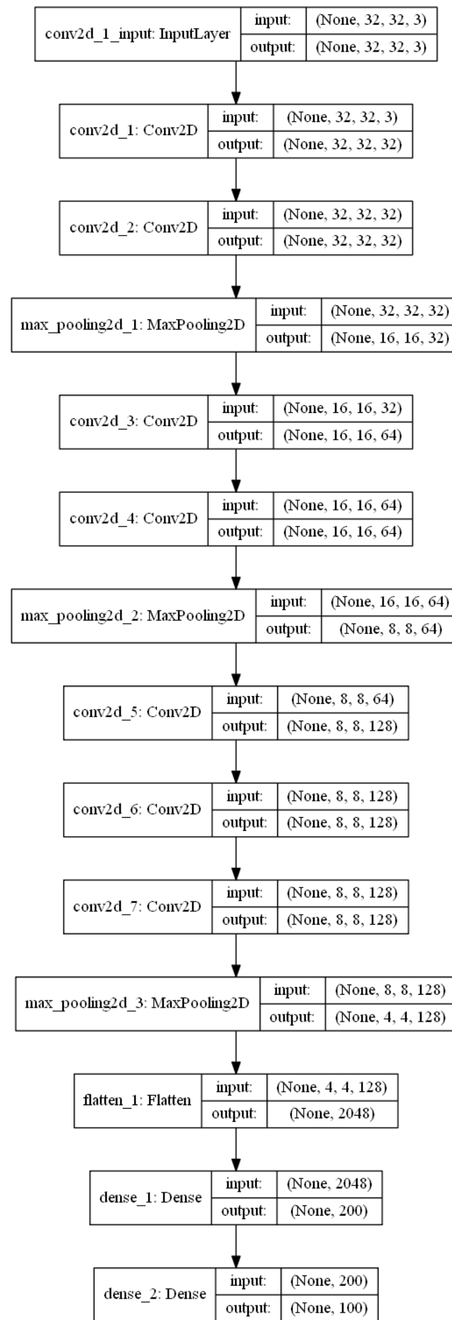

Figure 6.2: The CNN at the end of our experiment. Each layer shows the name, the type of layer and the input and output shape.

The design of the network was loosely based on CNNs used in the ImageNet LSVRC, primarily the winning network of the 2012 contest mentioned in Chapter 1. We followed several conventions surrounding the output sizes of layers, which we will elaborate on in the following sections. Regardless, the fine-tuning of the network consisted of much trial and error. As previously mentioned, finding the optimal hyperparameters is an optimisation problem of its own. The mathematical size of this problem is difficult to define, because there are theoretically infinitely many hyperparameters to optimise when the network structure is not fixed. However, due to following the conventions and due to the various aforementioned constraints, the optimisation problem is not as big. Note that it is not necessary to not find the optimal solution, as a fairly efficient solution is sufficient.

The hyperparameters to optimise include the structure of the network: which layers it consists of, the connections between the layers, the activation functions and the biases. Other hyperparameters are those related to the optimisation, such as the learning rate, the batch size and many other available options.

The implementation of a CNN in the Keras library is fairly straightforward. All required layers and their hyperparameters are built-in. We simply had to specify our network design in Python code, connecting it to our image dataset and export the results.

6.2 Convolutional layers

CNNs consist primarily of convolutional layers, as the name suggests. In particular, image recognitions uses 2D convolutional layers. A convolutional layer looks at small regions of the input image at a time. This groups together nearby pixels to more easily detect patterns and greatly reduces the number of connections and parameters.

Instead of weights, a convolutional layer uses kernels as its parameters. Kernels are small matrices used to convolve images. This convolution is a 2-dimensional discrete convolution. Suppose we are convolving a matrix $A = (a_{ij})$ with a 3×3 kernel $B = (b_{ij})$. The convolution of A and B in $(2, 2)$ becomes

$$A * B = \sum_{j=1}^{3} \sum_{i=1}^{3} a_{ij} b_{(4-i)(4-j)}. \tag{6.1}$$

This is really nothing more than an element-wise multiplication of a 3×3 region of A with B, where B is flipped horizontally and vertically. Alternatively, it can be seen as the dot product of the vectorised matrices after flipping. Since this calculation is easier to follow and to implement, the kernel is usually already flipped. We will also follow this convention.

Note that this is a linear function, similar to the weighted sum we used in ANNs. Therefore, we again need to use a non-linear activation function. When

it comes to classification, ReLU is usually still the best choice here.

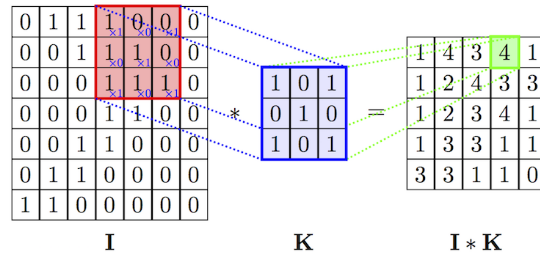

Figure 6.3: An example of a convolution. The region in red convolved with the kernel (blue) outputs a pixel (green) with value 4.

The above calculation is repeated for every 3×3 region of our matrix A to output a new matrix. We are effectively sliding the kernel over our image, row by row and column by column, convolving it with different regions of the image by multiplying the overlapping values and outputting the sum. The stride of the convolution is the number of steps taken each time we move the kernel. With a stride of 1, we get every region of the image. With a stride of 2, we skip every other region, both vertically and horizontally. As a result, each dimension of the resulting image will be less than half of the original.

Even if we have a stride of 1, the output image will still be smaller than the input, because of the size of the kernel. To keep the input and output size identical, we can use zero-padding. This adds zeros around the input matrix so that we can pick one region for each of the pixels of the original image.

Kernels are often used in image processing to blur or sharpen images. This works because we are essentially changing the value of pixels based on their surrounding pixels. By making them more similar, we blur the image. By making them less similar, we instead sharpen it.

Another application of kernels is edge detection. A kernel such as the one in figure 6.4 . Regions of the image where the colour of each pixel is nearly the same get a convolution output close to zero. However, if there is an edge in the middle with different colours on each side, the output will have a higher absolute value. This type of kernel is how CNNs detect edges and other shapes. Kernels are made to detect specific patterns and output high values when they match.

A kernel can take one image and output another. As mentioned before, however, we start out with three channels, r, g and b. This number of channels generally increases as we get deeper into the network. The channels do not necessarily represent colours anymore, but rather different patterns such as the edges mentioned before. In a convolutional layer, all channels of the previous

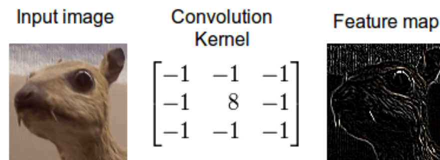

Figure 6.4: An example of a kernel that detects edges.

layer have a different kernel connecting to each of the channels in the current layer. The elements of each of these kernels are the parameters that will change during learning.

Suppose layer i is convolutional with c_i channels and kernels of size $n_i \times m_i$. Suppose layer $i-1$, convolutional or not, has c_{i-1} channels. Then the number of parameters in layer i is $c_{i-1}c_i n_i m_i + c_i$, where the last term represents the bias of each of the channels, which is added to each pixel in the output of the convolutions. This is a relatively low number of parameters, especially considering how high the dimensionality of each layer usually is. This is possible, because the image resolution, which is the main cause of the high dimensionality, does not factor into the number of parameters.

We can visualise a channel in any layer of a convolutional network by creating activation maps. When we input an image, we can see the activation in each of the channels. Alternatively, we can look directly at the filters, although there are often far too many of these to use them as a good overview of the network. For either of these visualisations, they become more abstract and harder to interpret as we go deeper into the network.

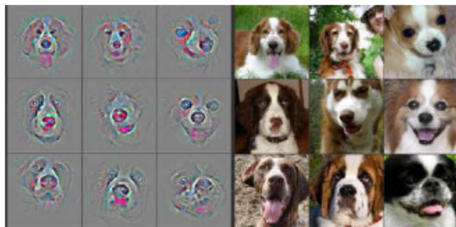

Figure 6.5: Visualising a channel of a network using deconvolution based on specific input images. [20]

More advanced methods of visualisation also exist. One such method [20] uses deconvolutional networks. Given a trained CNN, a deconvolutional network is built that is similar to the original network, but reversed. This allows the visualisation of a specific channel, by taking its output based on a specific input image and sending it to that same channel in the deconvolutional network. There, it makes its way back to the input space, which will result in an image

somewhat similar to the original input. The reason for the difference is that much of the information of the image is lost as it goes through the network. The idea is that what remains of the image, are the parts of the image that are important to the channel. Therefore, it serves as a good visualisation of the channel, given a specific input.

It is also possible to visualise channels by generating its input using SGD. Instead of minimising a loss function, we try to maximise the activation of a specific channel. Instead of changing the weights every step, we change the input image. This creates an input image that maximises the activation of a certain channel.

We can even take this one step further and create saliency maps. [21] Similarly to the SGD used during learning, we try to minimalise the loss for a certain classification. Except now, we once again change the input image instead of the weights. This creates an image that effectively shows how the network sees a specific classification. Since the images of a classification can vary immensely most of the time, this will not create a convincing image of said classification. Rather, the saliency maps are strange patterns, although often with recognisable traits of the classification.

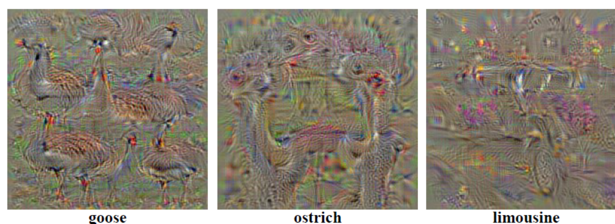

Figure 6.6: Saliency of several classifications. [21]

6.3 Max pooling layers

The second most common type of layer in CNNs is the max pooling layer. This is a fairly straightforward layer that has no parameters. For each channel, this divides it into disjoint regions of a given size. This size is usually 2×2, but larger regions are sometimes used for bigger images. For each of these regions, it takes the maximum value and outputs it as a pixel in a new smaller image. With a region size of 2×2, this creates a 4 times smaller image.

This layer aims to reduce the dimensionality of the network, which would otherwise increase due to the increasing number of channels. Max pooling layers are often inserted in between every two or three convolutional layers, usually right before the channel count increases. Since the channel count is often doubled, this causes the dimensionality to halve after every max pooling layer. We

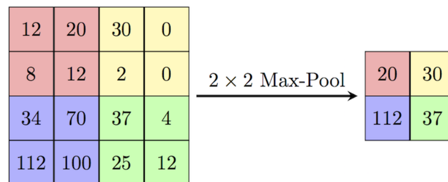

Figure 6.7: An example a 2×2 max pooling.

followed this convention for our network as well. This is evident from the input and output shapes we see in figure 6.2.

6.4 Fully-connected layers

The final set of layers in a CNN are fully-connected layers. In these layers, all nodes have connection from all nodes in the previous layer. Before we can make this transition, we need to flatten the output of the previous layer. This means taking all channels and turning each pixel into a node, as we know them from ANNs. The goal now is to work towards the output vector, which is generally a huge drop in dimensionality. Despite this, this part of the network usually has the majority of the parameters.

In a network built to classify images, we want every element of the output to be the probability of the corresponding classification being correct. At this point, the values will not necessarily even be between 0 and 1. To fix this, we use the softmax activation function in the final layer. This function makes the elements of the vector add up to 1. On top of that, it makes the highest values relatively stand out more. Since the desired output is a 1 for the desired classification and 0 everywhere else, this helps train the network in that direction.

Given an input vector $z^{(L)}$ of the final layer, we determine the activation output vector $a^{(L)}$ using the softmax function as follows:

$$a_j^{(L)} = \frac{\exp\left(z_j^{(L)}\right)}{\sum_{j'} \exp\left(z_{j'}^{(L)}\right)}. \tag{6.2}$$

This is often paired with the cross-entropy loss function, which is also commonly used for classification networks. [22] Given an input x, the output vector $\hat{y}(x)$ and the desired output $y(x)$, the cross-entropy loss function is given by

$$C_0(x) = -\sum_j y(x)_j \log \hat{y}(x)_j. \tag{6.3}$$

Note that the usual desired classification output is zero in all elements except one, which leaves only one non-zero term in the sum. The pairing of the softmax

and cross-entropy functions leads to high classification accuracy, which can be justified by a probabilistic interpretation. [22]

6.5 Results and analysis

On the test set, our final version of the network correctly classified 16,903 of 20,000 images, which is an accuracy of around 84,5%. It achieved this after around 5 hours of training. Since the accuracy on the training set was already around 96%, it was unlikely to improve much more. Considering a random guess would only have a 1% accuracy, this indicates a fairly effective network.

Before attempting to create a network for a full classification, we first tried training a network on determining only number, colour or shape. We started with a simpler version of the network, which did not include the final set of convolutional layers and max pooling layer. The dataset initially consisted of only 10,000 images, compared to the 100,000 of the final dataset. This change was made because the improvements from changing hyperparameters stagnated. This significantly improved performance, primarily reducing overfitting.

All changes made to the network since are a result of necessary fine-tuning. Changing one of the many hyperparameters can significantly change the effectiveness of training. While we could look at existing CNNs to help choose better hyperparameters, this will not necessarily make for a good network. What worked for their dataset, did not always work for ours.

For instance, the hyperparameters for SGD include momentum and decay. The winners of ImageNet LSVRC-2012 had a momentum of 0.9 and a weight decay of 0.0005, noting that these were important for the model to learn. [2] In our experience, however, setting these hyperparameters to values other than zero, seemed to negatively affect learning efficiency, sometimes to the point of coming to a complete stop early on.

Since it was often hard to tell what was causing inefficiencies in our training, the optimisation took some time to get started properly. At the start, many hyperparameters were at inefficient values, so it was often unclear whether certain changes really made a difference. It often took several changes to multiple hyperparameters before we saw noticeable improvement.

Initially starting with a simpler classification helped follow the progress of our fine-tuning before it was potent enough to perform adequately at the full classification. The easiest type of classification was clearly the colour. In fact, it was noticeably better at learning to classify colour in a smaller network. This is not surprising, because this classification can be read directly from the input data. Given the ability to do so, a network will often overcomplicate things during learning. The residual neural networks we covered in Chapter 3 had great results due to being able to prevent this overcomplication in much deeper

networks.

The confusion matrix in figure 6.8 shows the results of our final network at colour classification. Out of all 20,000 images, only one was classified incorrectly. The network mistook a blue triangle for a white triangle. It seems that even on the easiest part of the classification, our network is not perfect.

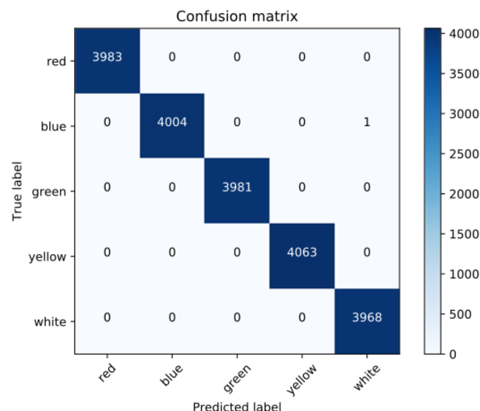

Figure 6.8: The confusion matrix for colour classification.

The second easiest classification was determining the number of shapes. While simply counting the number of black pixels could provide a decent estimate, the varying sizes of the shapes do make this more difficult. More than likely, some form of edge detection is necessary. The potential overlap between shapes makes counting them more difficult, but this is overall a fairly easy task.

The results of our final network at classifying the number of shapes can be seen in figure 6.9. The number of wrong classifications is fairly low and the number the network predicted was only ever off by one at most.

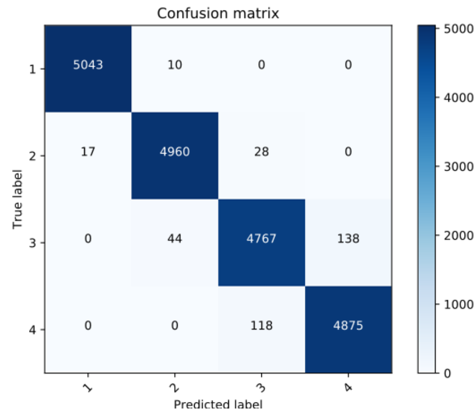

Figure 6.9: The confusion matrix for number classification.

Finally, classifying the shapes was the most difficult of the three. Shapes like pentagons, hexagons and circles are not very far apart, especially not when their size is small. The network is forced to be very selective to the angles between edges, while being completely invariant to position and size. Due to this complexity, this classification task was the best choice for fine-tuning before moving on to the complete classification.

Again, we plotted the results of shape classification in a confusion matrix, figure 6.10. It is clear that this was the toughest for the network to predict. Most mistakes appear to be a result of the close similarities between the pentagon, hexagon and circle. Other types of wrong classifications occur as well, but less frequently. The easiest shape to recognise seems to be the triangle, although it has been mistaken for both a pentagon and a hexagon, one time each.

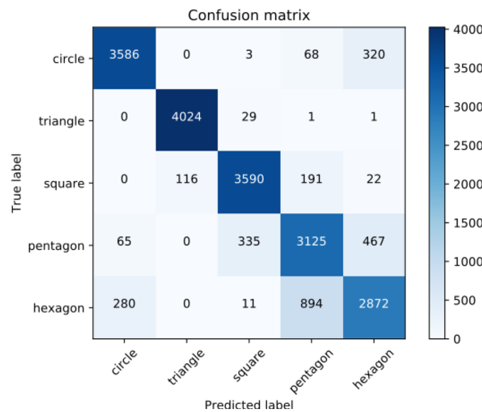

Figure 6.10: The confusion matrix for shape classification.

For the complete classification, we spend the most time fine-tuning the network architecture. This consisted of adding and removing layers and changing the output sizes. Some of these changes can greatly affect the number of parameters, which sometimes got into the millions. Increasing the number of parameters often improved the effectiveness of learning, at the cost of performance, taking significantly longer to finish an epoch. We fine-tuned the architecture until we had a good balance between the two.

At this point, we still had a dataset of 10,000 images. This proved to be the main limiting factor, as our network had major overfitting that the fine-tuning did not help much against. While the accuracy on the training dataset started reaching as far as 90%, the accuracy on the validation dataset stayed behind, never exceeding 25%. Increasing the dataset with a factor of 10 caused the accuracy on both sets to rise almost equally fast up to around 80%. From there, the training dataset exceeded 95%, while the validation set got to around 85%.

The results are not optimal, but still quite impressive. It might not be much compared to the winners of the ImageNet LSVRC contests, which achieve better results on far more complicated datasets. However, it did achieve its goal of classifying images in the simple dataset, despite limited time and hardware and being new to the field.

Bibliography

[1] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, June 2009.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[3] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[4] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989.

[5] G. H. Yann LeCun, Yoshua Bengio, “Deep learning,” *Nature*, vol. 521, pp. 436–444, 2015.

[6] R. R. S. G. E. Hinton, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, pp. 504–507, 2006.

[7] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, “On the expressive power of deep neural networks,” *arXiv preprint arXiv:1606.05336*, 2016.

[8] R. Eldan and O. Shamir, “The power of depth for feedforward neural networks,” *CoRR*, vol. abs/1512.03965, 2015.

[9] V. N. Vapnik and A. Y. Chervonenkis, “On the uniform convergence of relative frequencies of events to their probabilities,” in *Measures of complexity*, pp. 11–30, Springer, 2015.

[10] M. Karpinski and A. Macintyre, “Polynomial bounds for vc dimension of sigmoidal and general pfaffian neural networks,” *Journal of Computer and System Sciences*, vol. 54, no. 1, pp. 169–176, 1997.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[12] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[14] V. N. Vapnik, "An overview of statistical learning theory," *IEEE transactions on neural networks*, vol. 10, no. 5, pp. 988–999, 1999.

[15] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *arXiv preprint arXiv:1709.10432*, 2017.

[16] K. Kawaguchi, "Deep learning without poor local minima," in *Advances in Neural Information Processing Systems 29* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 586–594, Curran Associates, Inc., 2016.

[17] S. A. Teukolsky, B. P. Flannery, W. Press, and W. Vetterling, "Numerical recipes in c," *SMR*, vol. 693, no. 1, pp. 59–70, 1992.

[18] P. Baldi and K. Hornik, "Neural networks and principal component analysis: Learning from examples without local minima," *Neural networks*, vol. 2, no. 1, pp. 53–58, 1989.

[19] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 2933–2941, Curran Associates, Inc., 2014.

[20] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 818–833, Springer International Publishing, 2014.

[21] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[22] R. A. Dunne and N. A. Campbell, "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function," in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne*, vol. 181, p. 185, 1997.