



UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

Predicting Race Results using Artificial Neural Networks

Eloy Stoppels
M.Sc. Thesis
December 2017

Supervisors:

prof. dr. M.J. Uetz (UT)

Discrete Mathematics and Mathematical Programming
Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

In this thesis Artificial Neural Networks are used to predict Formula One finish results. The last four races of the season 16/17 are predicted based on the first seventeen races.

The first part of this thesis is dedicated to the theory behind Artificial Neural Networks. The aim is to give readers insights in the world and terminology of artificial neural networks. All the important terminology is discussed and explained, using some simple examples. Furthermore some key aspect are discussed in more depth. We look at what the influences of multiple layers and multiple neurons are. We make clear why a deep neural network is used. The next key aspect we discuss is; 'training an artificial neural network'. During training the free parameters are optimized. This optimization is done by solving a minimization problem. Therefore, training an artificial neural network can be done by using state-of-the-art optimization methods. Other key aspect are the different activation functions and cost functions which are used in artificial neural networks. Lastly, we explain how possible errors during training can be avoided by using so called regularization methods.

The second part is dedicated to the experimental research. First we give and explain public available data which we use as prediction features. Afterwards we show how to initialize an artificial neural network, where we show how multiple layers, activation functions, etc. influence the predicted results. We finish the experimental research by using the initialized network to predict the outcome of the last four Formula One races of the season 16/17. We compare three different data-sets, the first is the actual data of the season 16/17, the other two are data-sets where we added data in a certain way. Next to this, we compare the predicted results of the artificial neural network with two simple prediction methods and with multiclass logistic regression. The conclusion of this comparison is that using artificial neural networks has benefits as it predicts better outcome results.

Keywords: Artificial Neural Networks, multi-layer network, activation function, cost function, regularization, minimization, optimization

Preface

Dear reader, at this moment you are about to read my thesis. This thesis is my final step in graduating for my master's degree Applied Mathematics.

This report is the result of hard work, dedication and passion. It was a pleasure to combine my theoretical knowledge of mathematics to a more practical problem at confidential company.

Through this way I want to thank several people in particular for their help and support during this final project.

First, I would like to thank Prof. Dr. Marc Uetz. He has been my supervisor for both my internship and my thesis. Therefore, we have had a lot of contact the past year and a half. During this period, I learned a lot from him, not only theoretical knowledge but also (mathematical) writing skills. Furthermore, he pushed me to discover my limits and explore a whole new mathematical area. Namely, the area of machine learning and in particular Artificial Neural Networks. I found it pleasant to discuss all the papers and key features. Thanks to these discussions I can say that this report is both deep in mathematical and intuitive-explanations.

The rest of this part is only available with agreement of the author and the confidential company.

I hope you enjoy reading my thesis.

Eloy Stoppels,
December 2017, Amsterdam

Contents

1	Introduction	6
1.1	Company Background	6
1.2	Problem description	6
1.3	Motivation	6
1.4	Outline of this report	6
I	Theoretical Research	7
2	Literature review	8
3	Artificial Neural Networks	11
3.1	History	11
3.2	An introduction to Artificial Neural Networks	12
3.2.1	Definitions of Artificial Neural Network	12
3.2.2	Overview of Artificial Neural Networks	15
4	The influence of multiple layers and multiple neurons	23
5	Training an artificial neural network	27
6	Activation functions	32
7	Cost functions	36
8	Regularization	39
II	Experimental Research	44
9	Data preparation	45
10	Different data-sets	49
10.1	Enlarging the data-set	50
11	Results structure	52
12	Explanation of how to initialize an Artificial Neural Network	54
13	Results Formula One	60

13.1	Results for data-set consisting of 42 races	60
13.2	Results for data-set consisting of 42 races with Hamilton as a bad rain driver . . .	64
13.3	Results for the actual data-set, only consisting of 21 races	67
14	Conclusion	71
15	Discussion and further research	72
	Appendices	73
A	The XOR-problem	73
B	Derivative of the sigmoid activation function	74
C	Derivative of the hyperbolic tangens activation function	74
D	Derivative of the Softmax activation function	74
E	Explanation of the probabilities	76
F	Explanation of the cross-entropy function	76
G	Example of calculating the predicted race results	78

1 Introduction

1.1 Company Background

Only available with agreement of the author and the confidential company.

1.2 Problem description

This thesis has two main purposes. First of all the goal is to get insights in the world of machine learning. The main focus will be on Artificial Neural Networks. This implies that one part of this thesis is literature research. This research will be concluded with a written part which can be seen as an introduction to Artificial Neural Networks. In the end readers with little to no knowledge about artificial neural networks, should be able to understand the idea of artificial neural networks and get a feeling of all the different aspects involved.

The second part of this thesis is dedicated to the confidential company. Therefore, most of this part is only available with agreement of the author and the confidential company.

Note: rest of this section is not available without agreement.

1.3 Motivation

Not available without agreement.

Writing an introduction to artificial neural networks has the following reasons. First of all, there has been a lot of research in the area of artificial neural networks, and particularly in recent years tremendous successes have been reported, e.g. in the area of pattern and face recognition ([70], [71]). Therefore, the interest in artificial neural networks has renewed.

The focus of this work lies in the data analytics of timed (motorized) races. The power of artificial neural networks in (motorized) race prediction is only little investigated. This thesis can be used as a guidance in the world of artificial neural networks, specifically for race prediction. It can provide students and professors with the basic and necessary knowledge regarding artificial neural networks. Next to this a demonstration on Formula One race prediction shows how an artificial neural network could work in this area.

1.4 Outline of this report

This report is divided into two parts. In the first part the theoretical research is explained. In the second part the experimental research is discussed.

The theoretical research will start with a short review of previous literature research in the area of race prediction. The section afterwards will be about Artificial Neural Networks, where we will tell something about the history of artificial neural networks and we discuss in detail all aspects of this machine learning approach. Then we continue with sections that discuss the important parts of artificial neural networks in more detail.

In the second part we first start with data explanation and afterwards the predicting of Formula One races will be discussed.

We will finish this report with a conclusion and a discussion.

Part I. Theoretical Research

2 Literature review

This chapter will give an overview of the relevant performed research for the problem we are dealing with.

The aim is to find literature in the area of race prediction with the focus on motorized sports. Next to this we are also interested in machine learning approaches in sports in general.

In general, predicting the outcome of races, or other sports events, can be divided in three different areas: mathematical^{[1],[2],[3]}, psychological^{[4],[5],[6],[7]} and data mining or machine learning techniques. We will discuss some of those machine learning articles in more depth, because our focus is on machine learning approaches.

First let us look at research to make predictions in motorized race sports.

In the area of motorized sports events, such as motor-racing and car-racing, little research has been done. However, there are some interesting articles worth mentioning. In the paper of Graves, Reese and Fitzgerald ([8]) a Bayesian hierarchical framework is used to identify potential good individual racers, which could be perform well in a higher race class. For this they use three different classes of NASCAR races, namely WC, BGN and CT. Their data consists of the seasons 1996 till 2000.

In [9], Depken and Mackey discuss the driver success when he is part of a multi-car team instead of a single-car team. For this they explored the NASCAR Sprint Cup Series from 2005 till 2008. To predict the driver's success, they use a multiple regression method.

Allender [10] tries to predict the outcome of NASCAR races and in particular the importance of driver experience for this prediction. Allender uses a regression model with interaction terms. This model is tested on data of the season 2002.

Pfitzner and Rishel, [11], did research to find factors/features which could help to predict the outcome of a race. They only looked at features which are known before a NASCAR race starts. For this research they used correlation analysis on the finishing order with quantitative and categorical information collected. As research data they used the NASCAR season of 2003.

The last paper worth mentioning is that of Tulabandhula and Rudin, [12]. To goal of this article is to make real-time decisions such as when to change tires. This paper differs from other research, as it is the only one which is in the area of real-time decisions. They are searching for analytical methods for race prediction within the race and decision making strategies. In this paper the authors also describe some experimental shortcomings that restrict the predictions and some key differences between predictions in different sports areas. Furthermore they describe some of the complexities in race data. For modeling, they use all kinds of state-of-the art machine learning techniques, such as: support vector regression^[13] and LASSO (least absolute shrinkage and selection operator)^[14].

Interested readers in the area of predicting car racing are encouraged to read the article of Tulabandhula and Rudin.

The next step is to look at machine learning research in different sport areas. The main focus of these papers should be on outcome prediction.

In [15] Haghigat, Rastegari and Nourafza give a review of data mining techniques for results prediction in sports. This article is written in 2013 and studies the advantages and disadvantages of different data mining systems to predict sports results. They discuss the following classification techniques: Artificial Neural Networks, Support Vector Machines, Bayesian Method, Decision Trees, Fuzzy System and Logistic Regression. All methods have their own advantages and disadvantages. Furthermore, the methods can not be evaluated in the sense that one works better compared to another, because the methods are tested on different sports.

Leung and Joseph discuss in their paper, [16], a sports data mining approach to predict the win-

ners of American college football games. Instead of using the traditional approach of comparing the statistics of the two competing teams and projecting the outcome, their approach predicts the outcomes based on the historical results of American college football games.

In 'Predictive modeling in 400-metres hurdles races'^[17], Przednowek, Iskra and Przednowek try to predict the results of a 400-meters hurdles races. For this, they use the data of 21 athletes from the Polish National Team, with a high level of performance (score for 400 meter hurdles: $51.26 \pm 1.24s$). They use linear and nonlinear multi-variable models. The linear methods include; ordinary least squares regression and LASSO regression. For the nonlinear methods they look at; artificial neural networks and radial basis function network. The ANN predicts the result with an error of 0.72s.

Schumaker and Johnson, [18], investigated the use of Support Vector Machines to predict longshot greyhound races. They tested a Support Vector Regression algorithm on 1953 races across 31 dog tracks. For different betting systems they showed that their model performs better than random chance.

Purucker [19] and extended work of Kahn [20] uses Artificial Neural Networks to predicted outcomes in the National Football League (NFL). The model of Purucker almost predicted as good as experts in NFL prediction and the model of Kahn predicted the outcomes better than the experts. Tax and Joustra, [21], did research to a public based match prediction system for the Dutch Eredivisie. They tested different combinations of dimensionality reduction techniques and classification algorithms.

At last we will look at some papers in which the use of Artificial Neural Networks in race prediction is investigated¹. Due to these articles the idea to use Artificial Neural Networks for motorized (Formula One) race prediction is born.

The first article which we discuss can be seen as the start of this research area. It is an article by Chen et al. called 'Expert Prediction, Symbolic Learning and Neural Networks: An Experiment on Greyhound Racing' and is written in 1994 [23]. Their goal is to use machine learning techniques on greyhound racing. In this experiment they compare the predicted performances of two machine learning approaches with three human track experts. The machine learning approaches chosen are: decision-tree and artificial neural network. This research is one of the first in the area of *game playing*. This area is unstructured, complex and seldom-studied as stated by Chen. The following two questions are important in this paper: Can machine-learning techniques reduce the uncertainty in a complex game-playing scenario? Can these methods outperform human experts in prediction?

The data they use is of the Tucson Greyhound Park in Tucson, Arizona. This park holds about 112 races in an average week. For each race the park gives information about the dogs which are racing in that specific race and about track conditions etc. In total there are around 50 variables. The first phase consists of reducing the complexity, i.e. choosing the right variables. In the article, they use human experts and heuristic methods for this. This has to do with the fact that machine learning is not yet suited for this task. In the end, they have 10 performance variables, including; fastest time, win percentage, finish average, time 3 average. The following step is then to normalize/pre-process the data.

For the neural network they choose only one hidden layer with 25 neurons. The conclusion of this research is that both the decision tree method and the ANN outperform the human experts in terms of expected profit.

An article following the idea of Chen et al. is written by Johansson and Sonstrod [24]. They did a case study where artificial neural networks where used for the gambling domain in the area of greyhound racing. Instead of using 10 performance variables they used 18.

The following paper, [25], investigates the performance of four different neural network algorithms in horse racing. The horse racing data is from the Caymans Race Track in Jamaica. The data

¹ An recent article focusing on the application of Artificial Neural Networks to sports prediction in general is written by Bunker and Thabtah [22]

consists of 143 races, collected from 1 January to 16 June 2007. The four learning algorithms² are; back-propagation, quasi-newton, Levenberg-Marquardt and conjugate gradient descent. The experimental results show an accuracy of 74% when using back-propagation. To predict the results, Williams and Li use the following features; racing distance, type of race, past position, weight of horse and its jockey, horse's finish time, equipment used, age of horse and number of horses in the race.

In [26], an article by Davoodi and Khanteymoor, Artificial Neural Networks are applied to horse race prediction. Five different learning approaches are used, namely; normal back-propagation, back-propagation with momentum, quasi newton, Levenberg-Marquardt and conjugate gradient descent. The performances are tested on data collected from AQUEDUCT Race Track in New York, which include 100 actual races from 1 January to 29 January 2010. The results of this paper show that ANNs are appropriate methods in horse racing prediction context. Furthermore the results show that BP algorithm performs slightly better than other algorithms, but it needs a longer training time and more parameter selection.

² Later we will show that an artificial neural network can be seen as a minimization problem. To solve a minimization problem different approaches can be used, these approaches are called learning algorithms.

3 Artificial Neural Networks

Artificial Neural Networks have been used to solve a variety of problems in different scientific disciplines. For example, problems in pattern recognition, making predictions, optimization problems etc. Already existing approaches solve these kind of problems. Nevertheless, typically these do not perform well outside their domain. Artificial Neural Networks (we use the notation: ANNs) on the other hand, seem to be suitable to tackle all kinds of problems and many applications could benefit from using ANNs.

Indeed, modern computers outperform the human brain in multiple areas, such as numeric computation and related symbol manipulation. Yet the brain (humans) can solve complex perceptual problems without effort. Which again is a challenge for computers. An example of this is face recognition. We can recognize a person in a huge crowd by just a glimpse of his/her face. Even the world's fastest computer cannot do this at such a high speed. Therefore, the architecture of the biological neural system is a highly interesting area for researchers and this research has led to the introduction of Artificial Neural Networks. ANNs are inspired by these biological neural networks. For the interested reader we refer to page 33 of [27], where a section is dedicated to *Biological neural networks*.

Nevertheless, here we will focus on the mathematical idea behind ANNs. The goal of this section is to give the reader an insight in the world of ANNs. They will become acquainted with the kind of problems that ANNs can be useful for, the structure of an ANN and all the elements corresponding to this. Furthermore we will be looking at the difficulties when working with ANNs.

First we give a history overview with the most important contributions to ANNs. Then we start with an overview of ANNs where we give the definitions, the structure of the network and the mathematical idea behind ANNs. In the sections afterwards we explain the parts of ANNs which need further elaboration.

Until now a lot of research has been done with regards to artificial neural networks. A lot is discovered and explained, however, despite all this research there are still some unanswered questions in the world of 'deep' artificial neural networks. These questions are:

- How are 'deep' artificial neural networks able to perform in the way they do?
- What exactly do 'deep' artificial neural networks learn?
- Which network initialization is optimal for which task?

The goal of this research is not to answer these questions, as this is not currently doable. In the remainder of this report we give an overview of the key aspects to understanding the content of the questions better.

3.1 History

If we look throughout the history of ANNs we can see multiple periods of extensive activity. The first peak in the research of ANNs can be found in the 1940s, with the pioneering work of McCulloch and Pitts [28]. The second peak is the work of Rosenblatt. In 1957 he introduced the perceptron algorithm [29] and in the 1960s he came up with the *perceptron convergence theorem*. The perceptron is the first known artificial neural network and is able to divide input data in two classes. The algorithm calculates the weighted sum of the inputs and then uses a threshold function. If the sum of the input times their weights is higher than a certain value then the input belongs to class a and if it is lower it belongs to class b . Later, Minsky and Papert's work showed

the limitations of a simple perceptron[30]. They showed that a simple (single-layer) perceptron cannot perform simple logical XOR-operations. Due to this publication the interest in ANNs seems to have dropped. This drop in interest lasted for almost twenty years.

Around 1980 the interest in ANNs renewed. One of the reasons was the *back-propagation learning algorithm for multi-layer perceptrons*, proposed by Werbos [31],[32] and popularized by Rumelhart et al. in 1986[33]. Besides the fact that multi-layer perceptrons (*MLPs*) can solve the XOR problem, the back-propagation algorithm was a huge progress. It namely allows for quick training of MLPs. However, before this can actually happen, another step is required: instead of using the step function as activation function, the use of differentiable activation functions turned out to allow faster training. The back-propagation algorithm uses gradient descent to learn fast.

Nowadays most of the research is in the area of training an artificial neural network. New methods are researched to improve the learning abilities of an ANN. This new peak is due to; a) the availability of data, b) the availability of enormous computer capabilities in terms of storage and processing power and and c) the necessity to make sense of data for companies like Yahoo or Google. Another fact which lead to a lot of attention to ANN is the big success in face/pattern recognition lately, specifically using deep learning.

For the interested reader we refer to the article of Schmidhuber, where he gives an excellent overview of deep learning in neural networks [34].

We will shortly explain some of the basic terminology in ANNs, these terms will be explained in more detail in the next sub-section. An ANN can be seen as a function which maps its input to an output, where the inputs are multiple variables which are real valued numbers and the output is in most cases a probability. For example, we look at 'hypothetical' pre-screening for cancer detection, where the input is age, gender, genetic predisposition, etc. . The ANN maps this input to an output which says whether the person has the change of having cancer or not. The mapping consists of multiple steps, first all inputs are multiplied by a weight parameter. This weight parameter says how much influence the corresponding input variable has on the output. The next step is to sum all these new values, i.e. the new value is weight times input variable. The new value is the input for a function, where the answer of this function corresponds to the output.

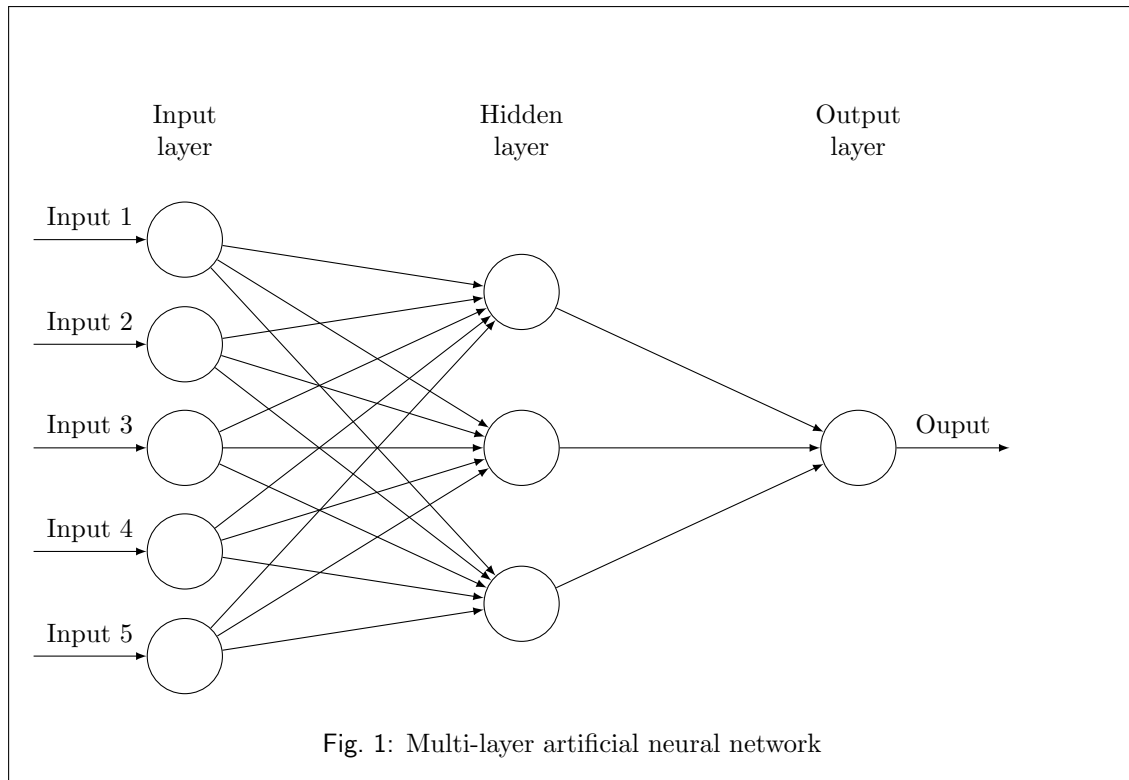
The functioning of an ANN is based on the idea that it "learns" by means of a set of input data that is used in a learning or training phase. In this phase, the "right" set of weights is determined iteratively, and based on the given training data. This is possible if the training data comes with actual output data, which is assumed to be the case. After finding the "right" weights the ANN can be used for unseen input data to predict their corresponding output.

3.2 An introduction to Artificial Neural Networks

3.2.1 Definitions of Artificial Neural Network

First of all we will give a short introduction to some of the important terms. These terms will be used multiple times throughout this report, so it is necessary that we understand these terms well.

To make our understanding better we start with a visualization in Figure 1.



As can be seen, an ANN is a collection of *layers*. Layers are used to bring more complexity power to the ANN, if we have more layers we can learn more levels of abstraction, as argued in [39]. We will explain this complexity later. A layer consists of *neurons*. All the neurons from layer i are connected to all the neurons from layer $i + 1$. These connections will be called *weights*. A neuron with its incoming weights is called a *perceptron*. A single perceptron is one of the first artificial neural networks, as proposed by Rosenblatt [29]. A perceptron is also called an *artificial neuron*. A perceptron, given in Figure 2, maps its input values to an output value. A formal definition will be given in Section 3.2.2.

If we look at Figure 1 we can see that there are three different kinds of layers. These are the *input layer*, the *output layer* and the *hidden layer(s)*. The *input layer* is the layer which stores the input data of the network. The *output layer* is the layer where the prediction output of the network is done. The *hidden layers* are added to be able to find complex structures in the data. In Figure 1 there is only one hidden layer, but in general there can be multiple hidden layers.

The network can have two types of basic structures, namely the *feed-forward* structure, as shown in Figure 1. In a feed-forward network there are no loops or backward connections. The network in which a loop or backwards connection exists is called a *recurrent* network.

The following representation is used for the ANN in Figure 1: 5-3-1. Here the first number means the number of neurons in the input layer, the second number is the number of neurons in the hidden layer and the last number is the number of neurons in the output layer.

In general an ANN has L -layers. The output layer is layer L and we have $L - 1$ hidden layers. The input layer is not an actual layer, so it is mostly called layer 0. To represent a network we thus have: $l_0 - l_1 - \dots - l_{L-1} - l_L$. Where l_i represents the number of neurons in layer i . This representation is also called the *network architect*.

In each layer there are neurons, a visualization of a neuron is given in Figure 2. In this figure we have three input variables, x_1 , x_2 , x_3 , the input variables are connected to the neuron by weights. These weights are w_{11} , w_{21} , w_{31} . The weights are free parameters and therefore can

be trained, this will be discussed in detail in Section 5. Furthermore we have a bias value b and $\Sigma = b + w_{11} \cdot x_1 + w_{21} \cdot x_2 + w_{31} \cdot x_3$. This Σ is the input for a so called *activation function*, denoted by $\phi(\Sigma) = a$. The value a tells if the neuron is *activated* or not. If it is activated the neuron adds value to the prediction. How this works will be discussed using an example.

If we look at Figure 1 we see three neurons in the hidden layer. In multi-layer networks the weights are denoted by $w_{i,j}^{l+1}$, this is the weight connecting neuron i in layer l with neuron j in layer $l + 1$. The trained weights can be such that the first neuron in the hidden layer has as input a combination of Input 1, Input 2 and Input 3, e.g. $w_{1,1}^1 = 1$, $w_{2,1}^1 = 1$, $w_{3,1}^1 = 1$. The next neuron is a combination of Input 3 and Input 4, e.g. $w_{3,2}^1 = 1$, $w_{4,2}^1 = 1$. The last neuron only represents Input 5, e.g. $w_{5,3}^1 = 1$. For all other $w_{i,j}^1$, $w_{i,j}^1 = 0$. If Input 5 is a randomly added input variable, then it will probably not influence the outcome of the network. In this case, the neuron in the hidden layer which represents Input 5, will not be activated. The activation function can be a binary function, where the output is 1 if the neuron is activated and zero for non-activated. Thus for this specific example, $\phi(w_{5,3}^1 \cdot \text{Input } 5) = 0$ and $\phi(w_{3,2}^1 \cdot \text{Input } 3 + w_{4,2}^1 \cdot \text{Input } 4) = 1$.

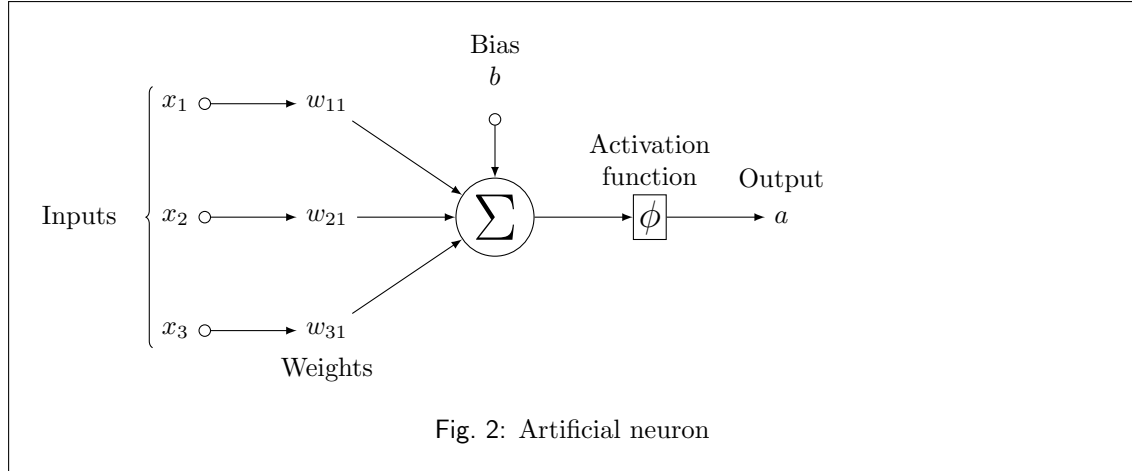
The goal of an ANN is to learn underlying structures of the data-set. If we look at the example of cancer detection, this could imply that the we want to know if there is a connection between age and gender with a higher probability of having cancer. In order to accomplish this, multiple learning methods are available.

In the example of cancer detection, we could have a data-base where age, gender, etc. is given for the people that participated in this hypothetical survey and if they have cancer or not. In this case, we know the corresponding output to each input, this is called *supervised*. In *supervised learning* the network learns from available input-output pairs. If we do not know the corresponding output, e.g. we only know age, gender, etc., but we are uninformed about their condition in terms of cancer, learning will be called *unsupervised learning*. A combination of both leads to *hybrid learning*.

When training the network, we want to know how good the predictions made by the trained ANN are. For the example of cancer detection, this amounts to knowing how many cancer patients were detected by the network, but also how many people are falsely classified. For this we use a *cost function*. The cost function will be discussed in Section 7. Training a network is finding the optimal weights, which capture the underlying structures of the data, this will be discussed in Section 5.

During training, the available data is divided into two sets; the *training set* and *testing set*. The training set is used to train the ANN in such a way that it learns the underlying structures and the testing set is used as validation. The testing set functionates as a check.

This check is namely needed, as there could occur possible errors. These errors are *overfitting* and *overtraining*. We will discuss these errors in depth in Section 8. To be short; overfitting has to do with the structure of the network and overtraining has to do with the number of training sample. The methods which are used to prevent the network to run into these potential pitfalls are called *regularization methods*.



3.2.2 Overview of Artificial Neural Networks

Problems

There is a wide class of problems for which ANNs can be used. An example, already given, is the problem of cancer detection. Here the ANN can be used to predict if a certain person has cancer or not. In this subsection we will discuss some problem types which can be solved using ANNs. Most attention will be given to classification problems. The end goal of this research is to predict race results. This can be seen as a classification problem, because we classify the racers for a certain end position.

A formal definition of the (supervised) classification problem follows; given a set of N different instances $x^{(i)} \in \Gamma \subset \mathbb{R}^n$, e.g. $x^{(i)}$ is a n -dimensional vector³, with corresponding labels $f(x^{(i)}) = y^{(i)} \in \mathbb{R}^k$, where k is the number of possible labels and f an unknown function. Next to this there exists a *training set*, called S^{train} with instances and corresponding labels:

$$S^{train} := \{(x^{(i)}, y^{(i)}) | i = 1 \dots m \text{ where } m \leq N\} \quad (1)$$

The classification problem is nothing more than approximating the unknown function $f : \Gamma \rightarrow \mathbb{R}$ such that $f(x^{(i)}) = y^{(i)}$ for $(x^{(i)}, y^{(i)}) \in S^{train}$. The goal is to find the "perfect" classifier, i.e. the function which classifies all $x^{(i)}$ to their the correct labels $y^{(i)}$. The classifier is trained on the set S^{train} . To measure the accuracy of the classifier f there is a different set, called *test set*, S^{test} which satisfies $S^{train} \cap S^{test} = \emptyset$. The meaning of accuracy in this problem is how much inputs of the test set are correctly classified to their corresponding labels.

The following holds for S^{test} ; $|S^{test}| = N - m$, furthermore for each $x^{(j)} \in S^{test}$ the corresponding label $y^{(j)}$ is known. S^{train} and S^{test} are disjoint sets because it is desired to train and test on different instances. The reason why this is desired has to do with the bias and variance of the approximate function, we discuss this in detail in Section 8, here a short explanations follows. Suppose we approximate the function f with \hat{f} , where for each $x^{(i)} \in S^{train}$ the $\hat{f}(x^{(i)}) = y^{(i)}$. In other words we have found the 'perfect' classifier for the training set. This means that the bias of the approximate function is low. However most of the times, in real life, the input consists of noise. Therefore, a low bias corresponds to a high variance, which is also known as overfitting. This implies that for 'untrained' inputs, e.g. the instances in S^{test} , the classifier has a lower accuracy than for the training inputs. To make sure there is a fair trade-off between bias and variance the accuracy is tested on inputs which are not the same as the training inputs. Based on the accuracy of both the training as the validation set we either continue with approximate the unknown function or we stop, e.g. terminate. In general we have N known instances, in S^{train}

³ In this case there are n -variables as input.

we have m instances and in S^{test} we have the other $N - m$ instances. A division of 75% training instances and 25% validation instances is a common choice.

For the cancer detection example an instance $x^{(i)}$ is the information known about a person i , e.g. $x^{(i)} = (\text{age}_i, \text{gender}_i, \text{genetic predisposition}_i, \text{etc.})$. In this case $n \geq 3$. The corresponding labels are 'cancer' and 'no cancer', where 'cancer' corresponds to $y = 1$ and 'no cancer' corresponds to $y = 0$.

For race prediction $x^{(i)}$ could be for example: $x^{(i)} = (\text{start position}_i, \text{car}_i, \text{best lap}_i, \text{experience}_i, \text{etc.})$. The corresponding labels are $y^{(i)} = 1, y^{(i)} = 2, y^{(i)} = 3, \text{etc.}$, where each $y^{(i)}$ corresponds to the finish position of racer i .

We will give some "simple" examples of classification problems to demonstrate how we could approximate the function f . In the first example we have three classes; circles, crosses and triangles. The goal is to separate these objects from each other. In Figure 3 a visualization is given. The three classes correspond to the three labels, thus for example $y^{(i)} = \text{circle}$. We have in total 30 different training instances x , instances are points in the Euclidean space, $x = (x_1, x_2) \in \mathbb{R}^2$. We denote with $X^{circles}$ all input instances which have label $y = \text{circle}$, e.g. $X^{circles} = \{x^{(j)}; f(x^{(j)}) = \text{circle}\}$. If $x^{(j)} \in X^{circles}$ then $x^{(j)} \notin X^{triangles}; X^{crosses}$.

One way of solving this problem is by first separating the circles from the crosses and triangles, e.g. finding a function f which divides all input instances in either cross/triangle or circle. The data is linearly separable, hence a classifier is given by some line that separates the circles from the crosses and triangles. This line is given by a hyperplane: $w_1^T x = b_1$. All instances x for which $w_1^T x < b_1$ holds are classified as circles. The same can be done to divide the crosses from the triangles, this hyperplane is given by $w_2^T x = b_2$. In the end this means the following for unseen instances x , if $w_1^T x > b_1$ and $w_2^T x > b_2$ the instance x belongs to the class of crosses. If however $w_1^T x < b_1$ the instance belongs to the class circles. A similar decision region can be obtained for the triangles.

For this example, note that the function f that will correctly classify the instances is an inner product of the input, x , times the weight vector w_i followed by a threshold b_i .

For this example constructing the function f is relatively simple, the data is linearly separable, which enable us to use hyperplanes. In general, the data which we want to separate is not linearly separable and we need to find another method.

Figure 4 depicts data that can be separated by means of a circle, which can be seen from visual inspection of the figure. Here the goal is to separate the inner circle with crosses from everything else, e.g. the circles. In this case it is not possible to separate the data using only hyperplanes $w_i^T x = b_i$. Using four hyperplanes gives the opportunity to make a square around the crosses. However, some instances x are then falsely classified. With 8 hyperplanes, the opportunity exists to make an octagon around the crosses, but still some instances will be classified wrong. Therefore, another method should be used to separate this inner circle. In this case a transformation can be used to make the data more representative. This method is a so called mapping from Cartesian coordinates to Polar coordinates⁴. In the new Polar coordinates, it is easier to linear divide, e.g. using hyperplanes, the data.

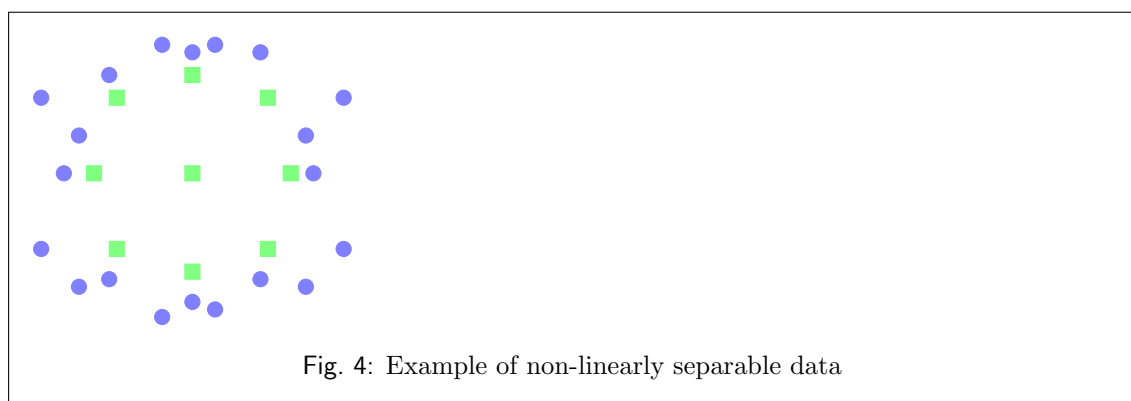
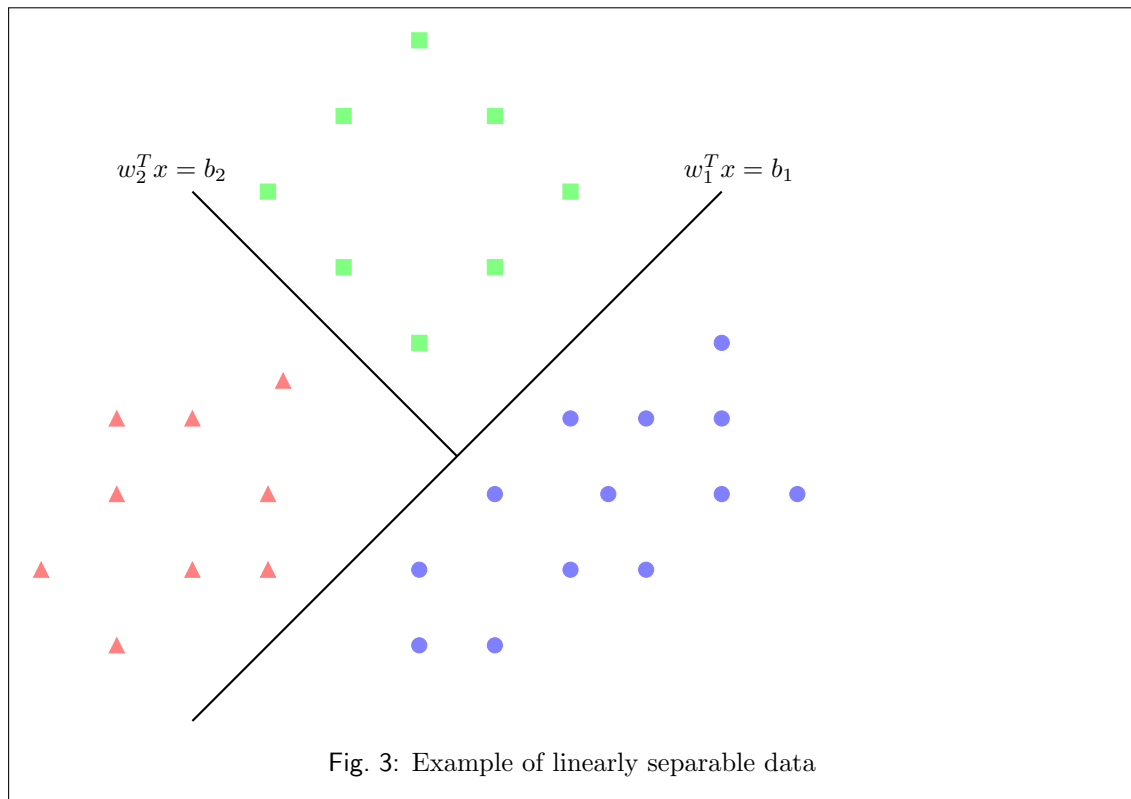
For this specific problem first a transformation, e.g. from Cartesian coordinates to Polar coordinates, is performed. After this transformation, the data is more representative and classification is easier. The transformation of a circle in Cartesian coordinates gives a rectangle in the Polar coordinates. A circle in Cartesian coordinates around the origin with radius R gives after the transformation a rectangle in Polar coordinates of size $[0, R] \times [0, 2\pi]$.

For the example we have a circle of crosses around the origin with radius R_{cross} , therefore after the transformation we get the rectangle $[0, R_{cross}] \times [0, 2\pi]$. If we perform the same transformation on the circles we get the rectangle $[R_{cross} + \epsilon, \infty] \times [0, 2\pi]$, where $\epsilon > 0$. These rectangles can be divided by the hyperplane $w_1^T x = R_{cross} + \frac{\epsilon}{2}$.

⁴ $(x_1, x_2) \rightarrow (r, \theta)$ where $r = \sqrt{(x_1^2 + x_2^2)}$, $\theta = \arctan \frac{x_2}{x_1}$

The approach discussed above seems like a good method to tackle general non-linear classification problems: first map the data to a more informative feature space and separate the data in this space instead of in the input space. This is exactly what an (multi-layer) ANN does, it maps its input data to a more informative space and performs separating in that space⁵.

An ANN thus performs two key steps; the first step is called *feature extraction*, where it maps instances to a more informative space. The second step is called *task specific*, where in the case of classification problems it means that the ANN tries to classify all instances to their right class/label.



Another problem for which ANNs can be used is function approximation. Suppose there is a set of n labeled training patterns. These are input-output pairs; $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$. These

⁵ This will be explained later, when dealing with multi-layer neural networks.

input-output pairs are generated from an unknown function $\mu(x)$, but of course, the pairs could include noise. The goal of function approximation is to find an estimation, $\hat{\mu}$, of the unknown function μ . In this problem, x_i can also be a vector. In this case we are searching for a function $\hat{\mu} : \mathbb{R}^n \rightarrow \mathbb{R}$. An example is given in [27].

Above we gave two examples of *supervised* learning. In supervised learning for all training instances, the corresponding output is known. Knowing this corresponding output makes finding a function $f(x) = y$ easier. The network is learning from these input-output pairs. It learns to determine the weights, w . For the example in Figure 3 the weight-vectors are w_1, w_2, w_3 and they are chosen in such a way that the weights correspond to the separating hyperplanes. During training the weights are adjusted in such a way that each adjustment will lead to more correctly classified instances. For the example of function approximation this implies that $\hat{\mu} \approx \mu$.

Another form of learning is *unsupervised learning*. In unsupervised learning it is not necessary for each input instance to have the corresponding output class/label. In unsupervised learning the goal is to find 'similar' input instances, e.g. input instances which have certain input features in common. These 'similar' input instances are then put in the same category.

For example see Figure 3; now for each instance it is not known whether they belong to the class of circles, crosses or triangles. However, in these 30 instances there can be found similarities in input instances. For example the instances corresponding to the triangles all have a low value of x_1 and x_2 . In unsupervised learning these instances are put together in the same category. Unsupervised learning is, at least intuitively, harder than supervised learning.

The last type of learning is *hybrid learning*. In hybrid learning supervised and unsupervised learning is combined. This means that for some input instances the corresponding output is known and for others it is not.

A well known unsupervised learning problem is clustering. Clustering, which is sometimes referred to as unsupervised pattern classification, has no known labels corresponding to the input instances. The goal of clustering is to find similarities between the input instances and place these similar instances in clusters. An application of clustering is data mining. The example given above can be seen as a form of clustering. In [34] and chapter 5 of [35] more can be learned about unsupervised learning. In [36] the focus is on data clustering.

Now we know which types of problems ANNs can solve, we are able to continue discussing the structure of an ANN.

The structure of an ANN

The perceptron of Rosenblatt [29] is one of the first ANNs ever build. It was designed for classification, e.g. saying whether instance x belongs to class one or two $f(x) = y_1$ or $f(x) = y_2$. Nowadays the idea of perceptrons is improved and the ANN structure has become more complicated. The perceptron by Rosenblatt is nowadays called an artificial neuron. An artificial neuron calculates the weighted sum of the input and processes this value through a (non)-linear activation function.

Definition 1 (Artificial neuron): A neuron with input $x \in \mathbb{R}^n$, input bias $b \in \mathbb{R}$, weights $w \in \mathbb{R}^n \setminus 0$ and a known activation function $\phi : \mathbb{R} \rightarrow \mathbb{R}$ and this neuron is defined as the function:

$$\xi(x; b, w, \phi) = \phi(w^T x + b) \quad (2)$$

Where $w^T x + b = \Sigma$ are the scores and the output $a = \phi(\Sigma)$ is referred to as the activation value.

Figure 2 indicates an artificial neuron, as defined in Definition 1. It is represented as an a-cyclic graph. Here the small circles are the inputs, which are multiplied by their corresponding weights, together with the bias, b , to become the score Σ . This is the score of the neuron (big circle). This

score is the input of a (non)-linear activation function which computes the (binary) activation a of the neuron. The perceptron proposed by Rosenblatt is the same as the artificial neuron in Definition 1, where the activation function is a threshold function;

$$\phi(\Sigma) = \begin{cases} 1, & \Sigma \geq 0 \\ 0, & \Sigma < 0 \end{cases}$$

Using Equation 2 and rewriting the above equation gives the following result;

$$\phi(w^T x + b) = \begin{cases} 1, & w^T x + b \geq 0 \\ 0, & w^T x + b < 0 \end{cases} = \begin{cases} 1, & w^T x \geq -b \\ 0, & w^T x < -b \end{cases} \quad (3)$$

i.e. the neuron separates the input space by a separating hyperplane: $w^T x = -b$. In this case the bias determines the threshold of the neuron, which is $-b$. The perceptron of Rosenblatt can only be used to separate the input data linearly. If this perceptron is trained, the weight vector w is updated. This is done in such a way that in the end the hyperplane is constructed which divide the input 'perfectly'.

Using only one perceptron (with threshold activation function) can divide the data into two classes, e.g. $w^T x < b$ and $w^T x \geq b$. Using multiple perceptrons together, e.g. an input layer and a hidden layer with at least two perceptrons can divide the input instances x in four classes; e.g. class 1: $w_1^T x < b_1$ and $w_2^T x < b_2$, class 2: $w_1^T x \geq b_1$ and $w_2^T x < b_2$, class 3: $w_1^T x < b_1$ and $w_2^T x \geq b_2$, class 4: $w_1^T x \geq b_1$ and $w_2^T x \geq b_2$.

In the Appendix Section A this approach, e.g. using multiple perceptrons, will solve the XOR-problem.

To solve the XOR-problem there was already need of a hidden layer. In general, the problems for which neural networks are used nowadays require multiple hidden layers. These complex problems can be solved more efficient using a 'deeper' network, e.g. a network with multiple hidden layers^{[44],[45]}. A neural network with multiple hidden layers is called a *multi-layer neural network*. A multi-layer neural network starts with an input layer and ends with an output layer. In between these two layers are so-called *hidden layers*, the 'heart' of the network. When there are more hidden layers used, the *depth* of the network increases. In the literature a multi-layer neural network with only one hidden layer is called a 'shallow' network. In Figure 1 a visual representation of a shallow network is given. A 'deep' neural network is another name for a multi-layer neural network. Nowadays, with 'deep' learning we mean learning a neural network with a large number (> 50) of layers and corresponding neurons.

Intuitively we can say that each neuron in a layer can be seen as an 'AND' or 'OR' operation, e.g. they tell if some input variables are relevant together or not. Using multiple layers behind each other can combine these 'AND' and 'OR' operations.

The reason why we say that 'deep' neural networks are more efficient will be discussed in Section 4. In short, the efficiency is in the computational speed, a 'deeper' neural network require less free parameters. Furthermore in Section 4 we will discuss the influence of multiple layers and neurons in a layer on the performance of an artificial neural network.

As discussed in Section 3.2.1 we have two main structures for an artificial neural network. These are:

- *feed-forward*: the network is acyclic.
- *recurrent*: the network may have cycles.

In Figure 5 we see multiple different 'names' for the network structures as they appear in the literature. However, all these structures are either a feed-forward network or a recurrent network. In the remainder of this research the focus is on feed-forward networks. In special feed-forward networks with multiple (hidden) layers, which is also called the multi-layer neural network.

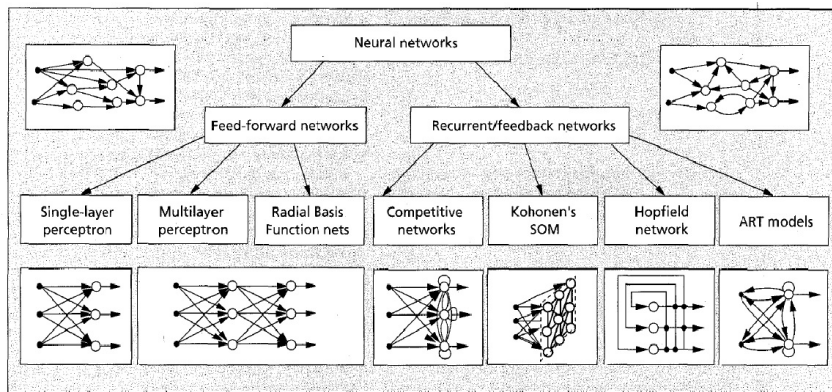


Fig. 5: 'Different' network structures, as given in [27].

We will give the mathematical definition for a multi-layer neural network in this section. However, we first need to discuss some important properties. One of these properties concerns the activation function for multi-layer neural networks. For multi-layer networks the threshold activation function is not used. We require for an activation function, in a multi-layer network, that it is continuous and differentiable. In a multi-layer neural network, the hardest part is training the network, e.g. finding the right set of parameters. In Section 5 we show that learning this right set of parameters can be done by solving an optimization problem. To solve this optimization problem the *back-propagation algorithm*⁶ is introduced. This algorithm uses the derivative of the activation function. Therefore, for multi-layer neural network we require continuous and differentiable activation functions. In Section 6 we discuss different activation functions which can be used. Furthermore, we will discuss some of the advantages and disadvantages they have, when used in a 'deep' network.

As already mentioned, during training the goal is to find the optimal parameters. The optimality is measured in terms of a *cost function*. For *supervised* learning the cost function is positive, e.g. > 0 , if the predicted outcome is not the same as the actual outcome, and zero if they are the same. For a classification problem this can be represented by Equation 4.

$$\text{cost function} = \begin{cases} > 0 & \text{if } \hat{f}(x^{(i)}) \neq y^{(i)} \\ 0 & \text{if } \hat{f}(x^{(i)}) = y^{(i)} \end{cases} \quad (4)$$

Meaning that a lower value of the cost function is desired. In Section 7 different cost functions for supervised learning are given.

Training an artificial neural network can be seen as two different computational processes. The first process is called the *forward pass of the network*. In the forward pass the cost function for a specific set of parameters is calculated. In the second process, e.g. the *backward pass of the network*, the back-propagation algorithm is used to update the parameters. This will be discussed in more detail in Section 5.

Another property we require for a multi-layer neural network is that it is *fully connected*. *Fully-connected* means that every neuron from the previous layer is connected to each neuron in the current layer. Below we give the definition of a *fully-connected layer*:

⁶ Will be discussed in Section 5.

Definition 2 (Fully-connected layer): Suppose we have inputs $x \in \mathbb{R}^{L_j}$ (the previous layer, j , has L_j neurons) and the current layer, $j+1$, has L_{j+1} neurons with corresponding differentiable activation functions⁷ $\phi : \mathbb{R} \rightarrow \mathbb{R}$, weights vectors $w_i \in \mathbb{R}^{L_{j+1}}$ and biases $b \in \mathbb{R}^{L_{j+1}}$. A fully-connected layer is defined by the function:

$$\phi(xW + b) \quad (5)$$

Where $W^{(j+1)} = W := ([w_1, w_2, \dots, w_{L_j}]^T) \in \mathbb{R}^{L_j \times L_{j+1}}$ is the weight matrix between layer j and layer $j+1$. The activation function, $\phi(\cdot)$ is applied point-wise. If we use the notation of Definition 1 and denote the input to the activation function as $\Sigma := xW + b \in \mathbb{R}^{L_{j+1}}$ and in the same way the outputs or activations as $a := \phi(\Sigma) \in \mathbb{R}^{L_{j+1}}$ we can simplify Equation 5 :

$$a = \phi(\Sigma)$$

A popular and simple choice of a differentiable activation function is the sigmoid function: $\phi(\Sigma) = \frac{1}{1+e^{-\Sigma}}$.

As already discussed, training an multi-layer neural network can be seen as learning a mapping into a more representative space. In this space we can perform classifications (e.g. for the classification problem). In general this means that a multi-layer neural network can be seen as a function going from an input space to a label space, consisting of a feature extraction followed by a task specific function. See Definition 3.

Definition 3 (Multi-layer artificial neural network or feed-forward artificial neural network): An artificial neural network is defined as the function $f : \mathbb{R}^n \rightarrow \mathcal{L}$, where f is:

$$f(x) = \Lambda(\mathcal{G}(x; \omega_G); \omega_\Lambda) \quad (6)$$

\mathcal{L} is the label space, $\mathcal{G}(x, \omega_G)$ is the feature extraction function, where ω_G are the weights which parametrize this function. Λ is a task specific function which is parametrized by another set of weights ω_Λ . For classification problems this task-specific function is a classifier.

Both the functions Λ and \mathcal{G} are compositions of multiple layers g_i . If there are in total L layers and N_G layers in the feature extraction the following holds:

$$\mathcal{G}(x; \omega_G) = (g_{N_G} \circ g_{N_G-1} \circ \dots \circ g_1)(x; \omega_G)$$

and

$$\Lambda(y; \omega_\Lambda) = (g_L \circ g_{L-1} \circ \dots \circ g_{N_G+1})(y; \omega_\Lambda)$$

The multi-layer neural network is a composition of multiple fully-connected layers. First, a certain number of fully-connected layers is taken in the feature extractor \mathcal{G} . Afterwards a certain number of fully-connected layers is taken in the task specific function Λ , which is designed to map the outputs of the feature extractor to the real output we strive for. The task-specific function consists mostly of a few fully-connected layers with the final output size equal to the number of outputs the problem has. This means for classification that Λ is a classifier where the output layer has equal size to the number of classes of the problem⁸.

In [37] M. Peemen, B. Mesman and H. Corporaal give an example of how we can interpret the use of feature extraction layers and task specific layers for the problem of handwritten digit recognition. A visualization is given in Figure 6. The goal is to recognize the handwritten digit, in this

⁷ The activation function need to be differentiable, otherwise the network cannot be trained. The *back-propagation algorithm* can not be used then, see Section 5.

⁸ For cancer detection we have two outputs, e.g. cancer or no cancer. For race prediction we have l outputs, where l is the number of racers in that specific race.

case 3. The input is a 32×32 grid, where some grid cells have high intensity if they contain a part of the digit and others have low intensity.

In the feature extraction phase the input grid is divided into smaller grids. Until there are sixteen 5×5 grids. The small grids are the input for the task-specific layers, in this case two. In the task-specific layers classification is performed. The output layer consists of 10 neurons, as there are 10 possible digits. The output of the network is the predicted digit.

Note: In this example the terms convolution and subsampling are used. Handwritten digit recognition is mostly performed by a special class of multi-layer neural networks, the so called *Convolutional neural networks* (CNN). In this report we do not pay attention to CNNs, because we focus on the general theory behind artificial neural networks. However, CNNs are often used nowadays, the majority of face recognition problems is solved by the use of CNNs. Therefore, for the interested reader, we refer to [38], [39]. The idea behind CNNs is to reduce the number of parameters by exploiting the similarity structure in images.

For this specific example the number of layers $L = 6$. If we look at Λ we can write this as $\Lambda(y; \omega_\Lambda) = (n_2 \circ n_1)(y; \omega_\Lambda)$ and $\mathcal{G}(x; \omega_G) = (S_2 \circ C_2 \circ S_1 \circ C_1)(x; \omega_G)$, with $x = \text{the input grid of } 32 \times 32$ and $y = \mathcal{G}(x; \omega_G)$. The parameters in this case are the weights connecting the different layers.

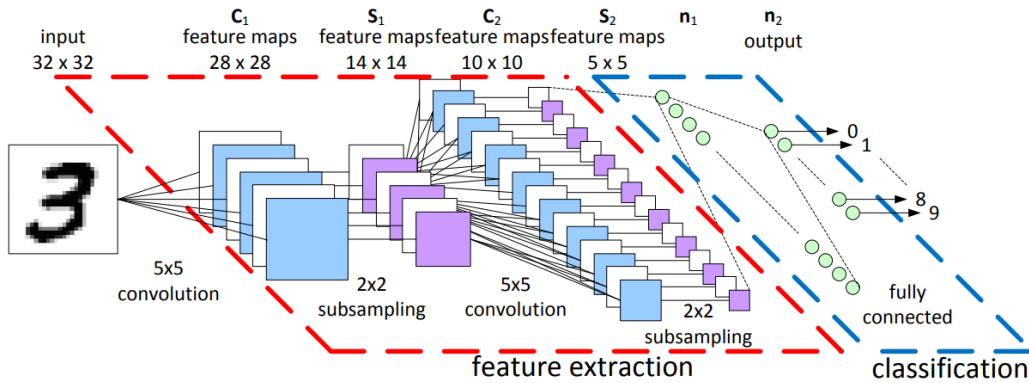


Fig. 1 CNN architecture for a handwritten digit recognition task.

Fig. 6: Example for feature extraction, as given in [37]. As input we have a 32×32 grid with a handwritten digit in it. The output is the predicted digit. First the grid is filtered through a convolutional layer with 5×5 kernels, yielding 6 feature maps of size 28×28 . These feature maps are subsampled through a 2×2 max-pooling layer. Then a convolutional step is performed and afterwards again a subsampling. After the feature extraction phase, finally the classification takes place. Where the output is a predicted probability for each digit.

4 The influence of multiple layers and multiple neurons

In this section the influence of multiple layers and multiple neurons on the performance of the neural network, will be explained. A visualization is given in which the difference of multiple layer can easily be seen. Furthermore, some important rules regarding the number of neurons and the number of layers are discussed. In addition to the rules it will be emphasized that the wrong network structure will lead to bad or wrong performance.

In Figure 9 a geometric interpretation is given, which can help to explain the role of hidden layers and the role of neurons in a hidden layer. In this geometric interpretation the activation function used is the threshold activation function, e.g. $w^T x \geq b$, OR $w^T x < b$. If there are L layers it means there are $L - 1$ hidden layers and one output layer. Thus a 2 layer network has one hidden layer and one output layer.

Neurons in the first hidden layer form a hyperplane in the label space; as already discussed, a hyperplane can be used to divide data in two classes. Neurons in the second hidden layer combine the outputs of the first hidden-layer neurons. This implies that in the second layer a new decision region is obtained. This decision region is obtained by performing logical AND & OR operations on the hyperplanes. To illustrate this; suppose in hidden layer 1 there are two neurons. Neuron 1 is activated, for input x , if $w^T x \geq b_1$ and neuron 2 is activated if $w^T x \geq b_2$. A neuron in the second hidden layer could then be activated if: 'Neuron 1 activated AND neuron 2 activated' ($\forall x$ where: $w^T x \geq b_1 \cap w^T x \geq b_2$); 'Neuron 1 activated OR neuron 2 activated' ($\forall x$ where: $w^T x \geq b_1 \cup w^T x \geq b_2$), e.g. its combines the hyperplanes of hidden layer 1. See also the example of the XOR-problem in Appendix Section A. The output-layer neurons combine these decision regions by performing logical OR operations.

As exemplified in Figure 9, the decision regions which can be obtained by using a multi-layer network are arbitrary. The complexity of the decision regions is limited by the number of neurons in the hidden layer, e.g. more neurons means a decision region which can become more complex. The arbitrary complexity of the single-hidden layer neural network is known as the 'Universal Approximation Theorem' and is proved by several researchers. This theorem can be described as: "a single-hidden layer (feed-forward) network can approximate every continuous function of n real variables with at most n neurons in the hidden layer. Where approximate means that we can get a error/cost $< \epsilon$ ". Here ϵ is the value of the cost-function. The theorem is first proved in 1989 by G. Cybenko [40] where he used the Hahn-Banach theorem to prove this theorem. For the proof, he used the following activation function: *sigmoid activation function*. Another proof uses the Stone-Weierstrass theorem, this proof is from K. Hornik, M. Stinchcombe and H. White [41]. Furthermore, they showed it for every *continuous non-constant activation function*. After this result, other researchers proved that "single-hidden layer feed-forward networks with at most N neurons (including biases) can learn N distinct samples (x_i, y_i) with zero error and the weights connecting the input neurons and the hidden neurons can be chosen "almost" arbitrarily"[42]. This is for the case where the activation function is the signum function⁹. The paper of Guang-Bin Huang and Haroon A. Babri (1998) [43] gives the same result but the difference is that their proof is for every bounded non-linear activation function which has a limit at one.

A visualization of this property is given in Chapter 4 of [55].

Why use a deep network given the 'Universal Approximation Theorem' just discussed? For this we look at results shown by Bengio and Le Cun [44] and Bengio [45]. They show, using complexity theory of circuits, that deep architectures can be much more efficient, sometimes exponentially,

⁹ The signum function or sign function is:

$$\text{sign}(x) := \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

than shallow architectures. This is in terms of computational elements and parameters required to solve a certain (classification) problem.

This sounds a bit counter-intuitive, therefore we will illustrate this. Suppose we have a fully-connected¹⁰ network with two hidden layers, each hidden layer has n neurons. This implies that we have $n \cdot n = n^2$ free weight parameters. Now, instead of two hidden layers, we have four hidden layers with $\frac{n}{2}$ neurons in each layer, that is the total number of neurons is the same. The number of weights connect one layer to another is $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$. In total we thus have $3 \cdot \frac{n^2}{4} = \frac{3n^2}{4}$ weight parameters. Dividing the number of neurons over four instead of two layers gives a decrease of $\frac{n^2}{4}$ in the number of free parameters.

For eight hidden layers with $\frac{n}{4}$ neurons in each layer we have $\frac{n}{4} \cdot \frac{n}{4} = \frac{n^2}{16}$ weights to connect two layers and thus in total $\frac{7n^2}{16}$ weight parameters. For sixteen hidden layers with $\frac{n}{8}$ neurons in each layer we have in total $\frac{15n^2}{64}$ free weight parameters.

To perform AND & OR operations we need at least two neurons in each layer. If we have a network with only two neurons in each layer we have n layers and thus $4n - 4$ free parameters. However, the complexity of the decision regions depends on the number of neurons in the hidden layers. Furthermore, for the problems where we use 'deep' networks we have in general multiple input features and output classes, where the number of input features is bigger than the number of output classes. In most cases the number of neurons in a hidden layer is higher than the number of output classes. A rule of thumb is namely to have at least $\frac{l_0 + l_L}{2}$ neurons in a hidden layer, e.g. the mean of the number of input neurons and output neurons. Therefore, two neurons in a hidden layer is not used most often.

Intuitively, it is not desired to have more hidden layers than number of neurons in a hidden layer, because the complexity depends mostly on the number of neurons in a hidden layer.

Therefore, a choice which can be made is to take $\sqrt{2} \cdot \sqrt{n}$ neurons over $\sqrt{2} \cdot \sqrt{n}$ layers, then we have in total $2 \cdot (\sqrt{2} \cdot \sqrt{n} - 1) \cdot n$ free parameters.

We thus can conclude that by multiplying the number of layers and dividing the number of neurons per layer, we get fewer free weight parameters. In Section 5 we see that for training an ANN we need to calculate the gradient of the cost function with respect to the parameters. Therefore, having less free parameters will also result in less computations.

Throughout the history of ANNs researchers have been trying to find the optimal network structure and the corresponding rules. In [46] an overview is given of methods which can help to choose the number of hidden neurons.

Another reason is given by Zeiler and Fergus in [39] and Larochelle et al in [47]. They cannot claim that deep networks perform better than shallow ones. However, there is evidence that using a deep structure can benefit when performing a complex task, as long as there is enough data available to capture this complexity.

Despite these reasons, 'deep' neural networks were not used most often, e.g. they are harder to train as shallow ones. However, the reason why nowadays 'deep' neural networks can be used has to do with this training part. A lot of research and development has been done to improve the training abilities of an ANN. The first step in this direction was the *back-propagation algorithm*. This algorithm required the derivative of the activation function. Therefore, the threshold function cannot be used. For this the sigmoid function is introduced, the sigmoid activation function is a continuous differentiable function. Furthermore, the sigmoid function 'looks' the same as the threshold function, see Figure 7.

¹⁰ See Definition 2 for the meaning of fully-connected.

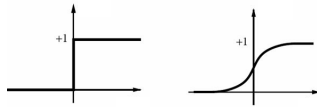


Fig. 7: Threshold and Sigmoid function

In Section 5 we discussed that training a neural network can be seen as solving an optimization problem. We know that we can solve an optimization problem by using *gradient descent*. However, simple learning methods such as *plain gradient descent* will lead to poor behavior for 'deep' networks. This implies that using a 'deep' network structure is still not preferable. However, there are all kinds of variations to gradient descent which allow us to train 'deep' networks. Some of these methods will be given in Section 5. Due to these better optimization methods 'deep' neural networks are the standard nowadays.

If increasing the number of layers and neurons will lead to a network which can solve complex problems (e.g. complex decision boundaries), why should we not always use a deep network. This seems to be a smart idea, having enough layers to solve complex problems should also solve less complex problems. However, this is not true. The structure of an ANN is important, choosing too many or too little layers will lead to problems and even bad performance.

When we have too many layers or neurons, the network may result in errors. These errors are *overfitting* and *overtraining*¹¹. Overtraining has to do with the number of training iterations. Overtraining may happen when we have too few samples and we train the network for too many iterations. Overfitting has to do with the network structure. In Section 8 we will discuss in detail the meaning of overtraining. Here, the meaning of overfitting will be explained based on an example, see Figure 8. In here, the black dots are the input pairs $(x^{(i)}, y^{(i)})$ and the goal is to find the unknown function $\mu(x)$. The input can contain noise, therefore we try to find the approximate function $\hat{\mu}(x)$. For this example $\mu(x)$ is a polynomial of degree 3, but the approximate $\hat{\mu}(x)$ is a polynomial of degree 5. The approximate is thus a more complex function as the actual function. This can happen if we choose a network structure which contains too many layers and neurons. In Section 8 we see that overfitting corresponds to a low bias of $\hat{\mu}(x)$ and a high variance of $\hat{\mu}(x)$. To have the best performance of a network, we need the right number of layers and neurons. The best performance of an ANN is reached when the 'easiest' network structure is used. Here 'easiest' means the network with the fewest number of layers and neurons which still has a good performance. The performance is the outcome of the cost function. For each problem a good performance can be different, e.g. for the problem of cancer detection a good performance should be that if a person has a change of having cancer the ANN predicts that this person has a high probability of having cancer. Or in other words, if a person has cancer and the ANN falsely predicts this person as not having cancer, the performance of the ANN is really bad. In the case of race prediction, it is more important to have the first three riders in the right order than the last three riders, e.g. here the performance is good as we predict the first three positions good. Another example of overfitting; we have a shallow network, i.e. one hidden layer, with P different input samples. In the beginning of this Section we gave the 'Universal Approximation Theorem' for shallow networks. If we use this property we can conclude that if we have more than P neurons in the hidden layer that we have a too complex structure. This comes from the fact that with at most P neurons in the hidden layer we can get a cost value of epsilon. Thus, using more than P neurons implies overfitting.

In general, it is not easy to say how deep the network needs to be and how many neurons there are in each layer. This is often a trial-and-error method. In the second part of this report a trial-and-error method is used to find the number of layers for Formule One prediction, see Section 12. To see if a network structure performs well we need to look at the value of the cost function for

¹¹ Will be discussed in more detail in Section 8

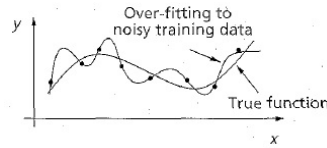


Fig. 8: Example of overfitting

both the training set as the validation set. If we find a network structure which performs well it is always necessary to try a less complex structure, e.g. less layers or less neurons in a layer, to see whether the chosen structure is the right one [49].

A method which tries different network structures is the *dropout method* [53]. This method takes a neuron into account with probability p . This will be discussed in Section 8. Another method is constructively adding layers [48]. As each layer in a multi-layer neural network can be seen as a representation of the input obtained through a learned transformation. By adding layers in this way, for each different network structure the performance, e.g. the value of the cost function, can be calculated. The network with the best value for the cost function is then the desired network. Other methods which can help to find the best structure will be discussed in Section 8.

To conclude this section, we will look more closely at how depth increases complexity. In general, for non-linear activation functions, we can explain the role of the layers as follows. Each (neuron in a) layer is a linear transformation ($w^T x + b$) followed by a non-linear activation function ($\phi(w^T x + b)$). This means that a neural network is a parametrization of a non-linear mapping. Increasing the depth of the network will lead to more complex and more non-linear mappings, which implies that the decision region can become arbitrary complex.

An increasing depth also means more neurons. If we interpret these neurons as decision units, we can explain the extra complexity in the following way. In each layer the neurons make decisions based on the value of the weights and the input of the layer. The input of a layer is the output of the previous layer, this means that we make decisions based on the outcomes of other decisions. In the end our outcome, the last complex decision, is built upon other, simpler decisions. Thus, increasing the number of layers (or neurons in a layer), the number of possible AND & OR paths towards the final decision outcome increases.


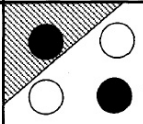

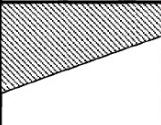

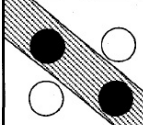

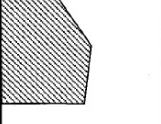

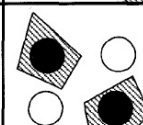

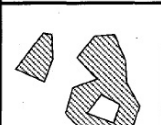
Structure	Description of decision regions	Exclusive-OR problem	Classes with meshed regions	General region shapes
 Single layer	Half plane bounded by hyperplane			
 Two layer	Arbitrary (complexity limited by number of hidden units)			
 Three layer	Arbitrary (complexity limited by number of hidden units)			

Fig. 9: Geometric interpretation of the role of hidden layers in a two-dimensional input space. [27]

5 Training an artificial neural network

At this moment the reader should be familiar with the underlying idea of an ANN and the structure, architecture behind it. Nevertheless, the hardest and most important part of an ANN is not understanding it, but training it. In this section we will explain how to train an ANN. For small networks it will turn out that training is not that hard. It is following the *back-propagation algorithm* in its easiest form. For deeper networks it turns out to be quite difficult.

When training a network, the goal is to find optimal parameters for a specific task. The first thing we need to define is what optimality means in terms of the output of the network, i.e. when is the output of the network bad and when is it good? This judgment is based on the value of a *cost-function*, which is based on a particular set of weights, the parameters. Furthermore, we need the actual output of a sample, this is called the *ground-truth*. We have costs when the output of the network does not equal the *ground-truth*.

In other words, we can train an artificial neural network as a minimization problem;

Definition 4 (The minimization problem of an ANN): The output of the network is the function $f : \Gamma \rightarrow \mathbb{R}^k$. Where $f^{(i)}$ is the output of the network corresponding to input $x^{(i)} \in \Gamma \subset \mathbb{R}^n$ and a specific set of parameters ω ; where ω are the weight matrices $W^{(i)}$ and bias b^{12} . The ground-truth corresponding to sample $x^{(i)} \in S^{train}$ is denoted as $y^{(i)}$.

The optimization of a network which is parametrized by ω , over all (N) samples $x^{(i)}$ can be described by the minimization problem;

$$\min_{\omega} C(\omega) = \frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)}) + \lambda J(\omega) \quad (7)$$

where C denotes the overall cost, $C^{(i)}$ is the cost for sample $x^{(i)}$, $J(\omega)$ is a *regularizer* with *regularization parameter* $\lambda \mid 0 \leq \lambda \in \mathbb{R}$.

Where $C^{(i)}$ could be: $(f^{(i)} - y^{(i)})^2$ for example, which is the quadratic cost formula. In this case the difference between the output of the network and ground-truth is squared to get the cost for instance i .

$J(\omega)$ could be: $\sum_{i=1}^L (W^{(i)})^2$ for example. In this case the regularizer is the sum of squares of the weights.

The different cost-functions will be discussed in Section 7 and the regularizer will be discussed in Section 8.

Training an ANN, which is also called *learning*, is done by solving Equation 7. It is also called learning, because during the training part the network is trained to seek input-output structures in the data. This is the same as saying that it learns how input affects output.

If we look more closely at Equation 7 the goal is to optimize the parameters given some function $C(\cdot)$. Or in other words, we are solving an optimization problem. An optimization problem in standard form is given by:

$$\begin{aligned} \min_x & f(x) \\ \text{subject to} & x \in \mathcal{F} \end{aligned}$$

Where \mathcal{F} denotes the feasible region. Looking at Definition 4 we minimize the overall cost function $C(\omega)$ with respect to the parameters ω . The feasible region in this case is $W^{(i)} \in \mathbb{R}^{L_{i-1} \times L_i}$

¹² See Definition 3 "Multi-layer artificial neural network".

$\forall i = 1, \dots, L$ and $b^{(i)} \in \mathbb{R}^{L_i} \forall i = 1, \dots, L$ ¹³, as ω consists of all weight matrices $W^{(i)}$ and bias-vectors $b^{(i)}$. Thus, training or learning an ANN is solving an optimization problem. This brings us to the property of using all kinds of optimization algorithms for optimization problems. The area of algorithms to solve optimization problems is well known and a lot of research has been done in this area. For example, there are heuristic algorithms which can be used. However, for the optimization of an ANN mostly iterative methods are used. These iterative methods can evaluate the Hessian or the gradient.

If we want to minimize Equation 7 we will use an iterative method called *gradient descent*. Gradient descent is an optimization algorithm which is mostly used for non-linear optimization problems and, as the name already suggests, it evaluates the gradient. The way gradient descent optimization works is as follow; it performs iterative steps on the parameters based on the gradient of the cost-function with respect to the parameters. Mathematically for the k -th iteration:

$$\omega_k = \omega_{k-1} - \eta \nabla_{\omega} C(\omega_{k-1}) \quad (8)$$

η is called the *learning rate*¹⁴. The iterative steps we take are in the direction of the steepest descent of the cost-function, the step size is η . The reason for taking steps in the negative direction of the gradients has to do with the problem we want to optimize. The gradient ($\nabla_{\omega} C$) points in the direction of the greatest rate of increase for function C . The optimization of an ANN is a minimization problem and therefore we need to go in the direction of the greatest rate of decrease, which is the negative gradient.

Instead of normal gradient descent, mostly *stochastic gradient descent* or *batch-wise gradient descent* are used in practice. Let us look why these methods are helpful when using it for non-convex optimization. In Figure 10 we can see the non-convex function $g(w)$. We have two minima, a local one (w') and a global one (w^*). If we start on the left of the graph with gradient descent and we go in the direction of the steepest descent, the algorithm will converge to the local minimum, i.e. the red dot (w'). If we start at the right of the graph, the algorithm will converge to the global minimum, i.e. the blue dot (w^*)¹⁵. Instead of using normal gradient descent and ending up in a local minimum [50] we can use some randomness to avoid getting stuck in a local minimum. This randomness can be accomplished by using stochastic or batch-wise gradient descent.

Another reason why we can use stochastic or batch-wise gradient descent is the computational speed. This will be explained in the next part of this section.

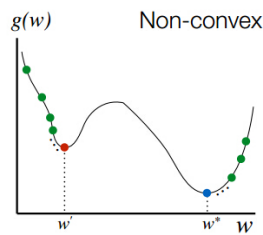


Fig. 10: Non-convex optimization with gradient descent, example.

In stochastic and batch-wise gradient descent the gradient $\nabla_{\omega} C$ is approximated. Since the cost-function is the sum over all training instances, the gradient is just the sum over all individual gradients, i.e. $\nabla_{\omega} C = \frac{1}{N} \sum_x \nabla_{\omega} C^{(x)}$. This is the sample mean of the cost over N samples. If we use some knowledge from statistics we can say the following; if we decrease the number of samples we decrease the confidence of our approximation. This means that we increase the randomness

¹³ See Definition 2 "Fully-connected layer".

¹⁴ At the end of this section we discuss some choices of η .

¹⁵ In this example we consider a learning rate which is smaller than the optimal learning rate. For convergence we need to choose the right learning rate.

in the updates. This is precisely what happens with stochastic/ batch-wise gradient descent, the gradient $\nabla_{\omega}C$ is approximated by the derivative of just one/a batch, randomly chosen, sample(s):

$$\begin{aligned} \text{stochastic: } \nabla_{\omega}C &\approx \nabla_{\omega}C^{(x)} \\ \text{batch-wise: } \nabla_{\omega}C &\approx \frac{1}{B} \sum_{j=1}^B \nabla_{\omega}C^{(j)} \quad \text{where } B \ll N \end{aligned}$$

In expectation the approximation of the gradient is equal to the actual gradient used in gradient descent:

$$\mathbb{E}_x[\nabla_{\omega}C^{(x)}] = \frac{1}{N} \sum_x \nabla_{\omega}C^{(x)}$$

Batch-wise gradient descent is a generalization of *stochastic gradient descent*. Instead of approximating the gradient with just one randomly chosen sample from the training set, the gradient in batch-wise is a batch-average. This average is usually of a small subset $B \ll N$, which is randomly chosen from the training set. If we choose $B = N$ we have normal gradient descent and if we choose $B = 1$ we get stochastic gradient descent.

Using stochastic or batch-wise gradient descent instead of normal gradient descent has two main advantages. The first we already pointed out, namely the benefit of adding stochasticity. We can avoid getting stuck in a local minimum, but actually reach the global minimum. However, this is not a guarantee. Another big advantage is the computational cost. If we use normal gradient descent, the cost function is a function of all samples in the training set. This means that we must process the whole training set before we get the value of the cost for a certain set of weights. In deep learning this becomes a real problem, because training sets for these kinds of problems can consist of more than a million samples. Calculating the cost over all these samples is far from practical. Using batch-wise gradient descent in this case is more efficient. In batch-wise the gradient is calculated only over a small subset. Now take the computational time to calculate the gradient over all examples. In the same amount of time we can calculate multiple batch-wise gradients.

Furthermore, the gradient of the batch is in expectation the same as the actual gradient. Therefore, the new set of parameters, for the k -th iteration, ω_k is in expectation the same for batch-wise gradient descent and normal descent. See Equation 8.

If we compare stochastic gradient descent with batch gradient descent we can give advantages for both methods. The advantages of stochastic learning are; it is usually much faster than batch learning. Next to this, stochastic learning in general leads to better solutions. The advantages for batch learning are; the conditions of convergence are well understood. Many acceleration techniques only operate in batch learning. The last point is that theoretical analysis of the weight dynamics and convergence rates are simpler. The advantages are given by Y. LeCun et al. in [60]. Furthermore in [60] all kinds of acceleration techniques (used for 'deep' networks) are discussed, such as Quasi-Newton (or BFGS) and the conjugate gradient method. These acceleration techniques are one of the main reasons that deep networks can be used nowadays. The acceleration techniques are all optimization algorithms and we see that research in the area of optimization problems is used to improve the performances of neural networks.

To update the parameters, as given in Equation 8, the *back-propagation algorithm* is used. The back-propagation algorithm is an implementation of gradient descent on the cost function. The algorithm uses the chain rule for differentiation to calculate the gradient. $\nabla_{\omega}C$, i.e. the gradient of the cost-function, needs to be taken over all trainable parameters (all weights which we can adjust). To calculate all these derivatives, we start with the parameters in the last layer and continue with the second last layer, etc. We thus move backwards in the network, this is called the *backward pass*. The reason of using the backward pass comes from the dependence of gradients; if we compute the gradient with respect to the parameters in layer l we need to know the gradient with respect to the parameters in layer $l + 1$. For this dependence we need the chain rule. An

explanation with illustration will be given in the next paragraph.

The back-propagation algorithm first computes the gradient of the cost-function with respect to the parameters in the last layer L . After computing this gradient, the gradient with respect to the parameters in layer $L - 1$ is computed. This process is repeated until the gradient of the first layer in the network is computed. The algorithm back-propagates through the network, it back-propagates the gradients from the last layer to earlier layers. That is why the name *back-propagation algorithm* is chosen for this algorithm.

To see why we need the chain-rule for differentiation, let us look closely at the formulas for the derivatives. We look at the stochastic gradient descent, thus $\nabla_{\omega} C$. Where C is the overall cost for sample $x^{(i)}$ and can be divided into the cost for sample $x^{(i)}$ and the regularization term, e.g. respectively $C^{(i)}$ and $J(\omega)$. If we only look at the cost for sample $x^{(i)}$ we have; $C^{(i)}(f^{(i)}, y^{(i)})$. Here $f^{(i)}$ is given by Equation 6 . If we look further in Definition 3 we see that $f^{(i)}$ is a composition all layers hence the cost-function is also a composition of all layers. If we take the derivative with respect to the parameters in layer $L - 1$ we can see in Definition 3 that the chain-rule for derivatives must be used for the derivative with respect to the parameters in layer L .¹⁶

We will will illustrate this by looking at the network in Figure 11.

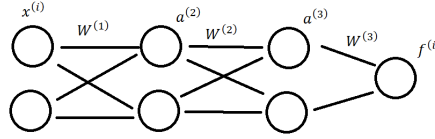


Fig. 11: An example network to show how back-propagation works

Where $x^{(i)}$ is the i -th input sample and $f^{(i)}$ the outcome of the network corresponding to input sample i . We are interested in the gradient of the cost function with respect to the parameters, this is $C^{(i)}(f^{(i)}, y^{(i)})$. In the cost function only $f^{(i)}$ depends on the parameters and therefore we can look at the gradient of the outcome with respect to the parameters. Thus we get: $\nabla_{\omega} f^{(i)}$. The parameters in this case are: $\omega = \{W^{(1)}, W^{(2)}, W^{(3)}\}$. Next to this, following Definition 2, we have; $a^{(2)} = \phi(x^{(i)} \cdot W^{(1)})$; $a^{(3)} = \phi(a^{(2)} \cdot W^{(2)})$ and $f^{(i)} = \phi(a^{(3)} \cdot W^{(3)})$.

Thus first we calculate the gradient with respect to the parameters in the last layer: $W^{(3)}$.

$$\nabla_{W^{(3)}} f^{(i)} = \nabla_{W^{(3)}} \phi(a^{(3)} \cdot W^{(3)}) = a^{(3)} \cdot \phi'(a^{(3)} \cdot W^{(3)})$$

The next step is to calculate the gradient with respect to the parameters in layer $L - 1$, in this case $W^{(2)}$.

$$\nabla_{W^{(2)}} f^{(i)} = \nabla_{W^{(2)}} \phi(a^{(3)} \cdot W^{(3)}) = \nabla_{W^{(2)}} \phi(\phi(a^{(2)} \cdot W^{(2)}) \cdot W^{(3)}) = a^{(2)} \cdot \phi'(a^{(2)} \cdot W^{(2)}) \cdot W^{(3)} \cdot \phi'(a^{(3)} \cdot W^{(3)})$$

In the same way we get for the gradient with respect to $W^{(1)}$:

$$\nabla_{W^{(1)}} f^{(i)} = x^{(i)} \cdot \phi'(x^{(i)} \cdot W^{(1)}) \cdot W^{(2)} \cdot \phi'(a^{(2)} \cdot W^{(2)}) \cdot W^{(3)} \cdot \phi'(a^{(3)} \cdot W^{(3)})$$

We now see that for the gradient in layer $L - 1$ we need the result of the gradient in layer L . Hence the name *back-propagation*.

To train an ANN the following steps are taken; first the parameters are initialized following some rules, see next paragraph. The choice of initialization depends on the chosen activation functions, i.e. the range they have. Then a repeated process starts, this process is repeated until convergence

¹⁶ The function is $(g_L \circ g_{L-1} \circ \dots \circ g_1)$ thus if we are at g_{L-1} we have $f(x) = g_L \circ g_{L-1}$. If we want the derivative of $f(x)$ with respect to g_{L-1} we need the chain-rule.

$(f \circ g)' = (f' \circ g) \cdot g'$

or manual termination. In the process we start with a *forward pass* to compute the cost of this specific combination of weights. After the forward pass we perform the *backward pass*, where we compute all gradients. The next step is to update all weights according to Equation 8. Now we start again with the forward step.

When we initialize the weights, before we start with the first feed-forward step, we need to make sure we follow the rules as given by [60], [61] and [62]. These articles provide rules for initializing weights for distinct activation functions. The weights are initialized based on the number of neurons in a layer. If we start with these initialized weights, it is easier to train the network, which will make the performance of the network better. To illustrate, we will give the rule when we use sigmoid function as activation function. In [62] they give the following rule, for the weights between layer i and $i + 1$:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{l_i} + \sqrt{l_{i+1}}}, \frac{\sqrt{6}}{\sqrt{l_i} + \sqrt{l_{i+1}}} \right]$$

U is the uniform distribution and l_i is the number of neurons in layer i .

There is however one important property which we should not violate. We cannot initialize the weights to zero, thus $w_{i,j} \neq 0 \forall i, j$ where $w_{i,j}^{l+1}$ denotes the weight between neuron i in layer l with neuron j in layer $l + 1$. If $w_{i,j}^{l+1} = 0$ then $\partial_{w_{i,j}^{l+1}} C(\cdot) = 0$. This implies that $w_{i,j}^{l+1} = 0$ for all iterations, see Equation 8. This of course will not improve the learning ability.

A note regarding the learning rate η . The choice of this learning rate is important. Choosing this learning rate wrong can lead to bad performances of the network. The choice of η depends on the problem we are dealing with and the way the optimization is done. Stochastic gradient descent requires a different choice of η as batch-wise gradient descent. In [60] LeCun et al. gave some tricks to choose a good learning rate.

Let us look at some different learning rates, to see how they perform. This is given in Figure 12. If we choose η to be strictly smaller than the optimal learning rate, η_{opt} we will convergence to the optimal value for the parameters, see Figure 12a. However, if we take η too small it can lead to slow or no convergence at all, at each step we come closer to the minimum but also the step we take will be smaller. This can carry on for an infinite number of steps without convergence.

If η is exactly η_{opt} , convergence will take place in one step, see Figure 12b.

Taking η bigger than η_{opt} will lead to 'zigzagging', see Figure 12c. However, if the learning rate η is bigger than $2 \cdot \eta_{opt}$ we do not get convergence but divergence instead, see Figure 12d.

Therefore, when training an ANN, it is good to pay attention to the learning rate and try different choices of η .

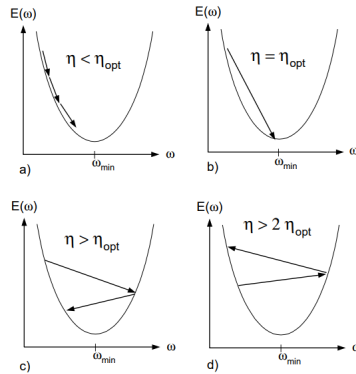


Fig. 12: Different choices of the learning rate in comparison with the optimal learning rate^[60].

6 Activation functions

In this section the idea of the activation function will be discussed and some of the most used activation functions will be given. We will also discuss the different activation functions in terms of usability.

The first known activation function is the threshold function. The threshold function is used in the perceptron as given by Rosenblatt in 1957 [29]. The threshold gives a binary output, where the output is 1 if the input is greater equal a certain threshold c and zero otherwise, as represented by the formula in Equation 9.

$$f(x) = \begin{cases} 1 & \text{if } x \geq c \\ 0 & \text{if } x < c \end{cases} \quad (9)$$

However, nowadays the threshold function is not used anymore as an activation function. Around the 1980's the back-propagation algorithm was introduced^{[31],[32]}. The back-propagation algorithm makes it easier to train 'deep' neural networks. As already discussed in Section 4 using 'deep' networks is desired over 'shallow' networks. The back-propagation algorithm requires the use of continuous differentiable activation functions. The threshold function is not continuous and not differentiable and therefore cannot be used in 'deep' networks.

The activation function which is first used in the back-propagation algorithm is the sigmoid function. The sigmoid function is a continuous and differentiable function. The sigmoid function is given by the formula in Equation 10.

$$f(x) = \theta(x) = \frac{1}{1 + e^{-x}} \quad (10)$$

The derivative of the sigmoid function is (see Appendix Section B for the derivation):

$$\frac{d}{dx}f(x) = f'(x) = \theta(x)(1 - \theta(x)) \quad (11)$$

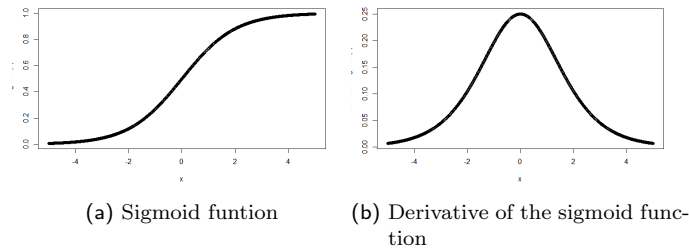


Fig. 13: The Sigmoid function

In Figure 13a we see that the sigmoid function looks like a "smooth step function". The $\lim_{x \rightarrow \infty} \theta(x) = 1$ and $\lim_{x \rightarrow -\infty} \theta(x) = 0$, and $\forall x; 0 < \theta(x) < 1$.

However, the sigmoid activation function is not suitable for the 'deep' networks as used nowadays. The use of the sigmoid activation function in 'really deep' networks can cause the problem of *vanishing gradient*. The problem of vanishing gradient can be described as follows; in deep neural networks the gradient tends to get smaller as we move backward through the network. If we update the parameters in layer l we need the gradient of the cost function with respect to these parameters. The gradient with respect to the parameters in layer l is a product of parts of the gradients of layers $l+1, l+2, \dots, L$, as explained in Section 5. To update the parameters of layer l

we already have $L-l$ multiplications of derivatives of the activation function. In Figure 13b we see that the derivative of the sigmoid function is between $(0, 0.25)$, hence smaller than 1. Therefore, the gradient of the cost-function with respect to the parameters in the lower (first hidden layers) layers will decrease exponentially with the number of layers L and hence tends towards zero. This implies that the parameters in the earlier layers will learn slower than the parameters in later layers, resulting in bad overall performance of the network. For an example and extra information about unstable gradients, we refer to Chapter 5 of [55].

Something quite similar to the sigmoid function is the hyperbolic tangens.

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 \quad (12)$$

The hyperbolic tangens is in fact a scaled version of the sigmoid function; e.g. $\tanh(x) = 2 \cdot \theta(2x) - 1$.

The derivative of the hyperbolic tangens is (see Appendix Section C for the derivation):

$$\frac{d}{dx} f(x) = f'(x) = 1 - \tanh(x)^2 \quad (13)$$

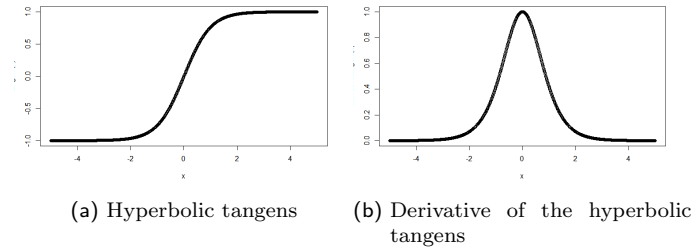


Fig. 14: The Hyperbolic Tangens function

The hyperbolic tangens has similar characteristics as mentioned for the sigmoid function, as it is a scaled version of the sigmoid function. The hyperbolic tangens is bounded between $(-1, 1)$, the $\lim_{x \rightarrow \infty} \tanh(x) = 1$ and $\lim_{x \rightarrow -\infty} \tanh(x) = -1$, and $\forall x; -1 < \tanh(x) < 1$. The hyperbolic tangens is more suitable for 'deep' neural networks than the sigmoid function. In Figure 14b the graph of the derivative of the hyperbolic tangens is given. The derivative reaches value between $(0, 1)$. However, as the derivative is still smaller than 1 the hyperbolic tangens also has the problem of vanishing gradient.

In multi-layer networks, which are not too deep, the hyperbolic tangens is used instead of the sigmoid as activation function.

We discussed that activation functions with derivatives smaller than 1 have the problem of vanishing gradient. On the other hand, if we take activation functions with derivatives larger than one, we get a similar problem. This problem is called *exploding gradient*, e.g. the gradient in lower layers will exponentially grow.

To avoid the problem of vanishing or exploding gradient we want to have an activation function for which the derivative is constant and equal to 1. A function which has a constant derivative is the linear function, as given in Equation 14.

$$f(x) = ax + b \quad (14)$$

The derivative of the linear function is the constant value a . Therefore, if we choose this a to be equal to 1 we have an activation function which has a constant derivative equal to 1.

An activation function which is used in practice and shares this idea is the ReLU (Rectified linear units) activation function [66]. The ReLU function is given by the formula in Equation 15:

$$f(x) = \max\{0, x\} \quad (15)$$

The ReLU function is a continuous activation function with a constant derivative. For positive inputs the derivative is 1 and for negative inputs the derivative is 0. However, this derivative of zero can cause a problem called *dying ReLU problem*. A derivative of zero means that the weights will not be adjusted during training. If the weights are not changing, the neurons will stop to respond to variations in the input. The dying ReLU problem can thus be described in the following way: due to the gradient of zero, some of the neurons will die out and will not respond anymore, which makes a substantial part of the network passive.

We can solve this issue quite easily, with the so called "leaky ReLU" [67]:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases} \quad (16)$$

The goal of this function is to make sure that the gradient is not zero anymore.

We will lose one advantage if we use "leaky ReLU" instead of ReLU. This advantage is called; *sparsity of the activation*, which is caused by "dying ReLU". With this we mean that in a large network almost all neurons will normally have an activation value unequal to zero. In the end the output will be described using all neurons/activations. In other words, the activation is dense and this is highly costly. If we have multiple neurons with activation value equal to zero, these neurons will not be activated and thus the activations are more sparse implying more efficiency. In "leaky ReLU" the derivative is never zero and hence all neurons will give a value.

For the ReLU activation function we now have argued that *dying ReLU* does not only have to cause problems, but might result in an advantage too. Dying ReLU is indeed a problem, but for really 'deep' networks it might help us in a positive way.

As can be seen, choosing the right activation function is quite difficult. All activation functions have their own advantages and disadvantages. However, we can say something about choosing the right activation function. In small networks, e.g. a few hidden layers, the hyperbolic tangens is mostly used. In deep neural network, as used nowadays, the ReLU activation function is mostly used. The reason behind this is that a ReLU activation function does not have the problem of an unstable gradient. Next to this it has the advantage of the sparsity of the network which speeds up the process.

Instead changing the activation function, researchers have also proposed to use a new sort of neural network. This network is called *residual network* [70]. In this new network skip-connections are used to guarantee stable, non-zero gradients.

All activation functions above calculate the activation value only based on their own input, x . There are however also activation functions which calculate the activation value differently, e.g. they calculate the activation value of neuron i based on the inputs for all neurons in that layer. One of these functions is worth mentioning. This is the Softmax activation function, as given by the formula in Equation 17. In this function, i denotes the i -th neuron in the layer:

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^J e^{x_j}} \quad \text{for } i = 1, \dots, J \quad (17)$$

Where (\vec{x}) is a vector which contains the different inputs for all neurons in the layer; $(\vec{x}) = \{x_1, x_2, \dots, x_J\}$.

A property of the Softmax function is that *all the activation values in a layer sum up to 1*. When interested in output probabilities for a certain class, the Softmax function is highly useful as activation function in the output layer. We can interpret the Softmax function as follows

$f_i = P(y = i|x)$. An explanation of this can be found in Appendix Section E. The derivative of the Softmax function is, taken component-wise:

$$\frac{\partial f_i(\vec{x})}{\partial x_j} = f_i(\vec{x})(\delta_{ij} - f_j(\vec{x})) \quad (18)$$

Where δ is the Kronecker delta ¹⁷. The derivation of this derivative is given in Appendix Section D.

When looking at classification problems, the Softmax function is commonly used as activation function. It is used in the output layer to make sure that the output is a probability. In the hidden layers mostly the hyperbolic tangens is preferred as an activation function for 'small' networks and the ReLU for 'deep' networks.

¹⁷ The Kronecker delta is defined as follows:

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

7 Cost functions

In this section we will discuss some cost functions which can be, and are used for Artificial Neural Networks.

For the cost function we introduce the following formula, Equation 19:

$$C = \sum_{i=1}^N C^{(i)}(\vec{x}^{(i)}) \quad \vec{x}^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)}\} \quad (19)$$

In this function C denotes the cost. Next, $C^{(i)}$ denotes the cost of training example i with respect to input vector $\vec{x}^{(i)}$, where the input vector consists of n different input variables: x_j . N is the total number of training examples we have. In this section we denote the output of the network as \hat{y} , where $\hat{y}^{(i)}$ is the output corresponding to input $\vec{x}^{(i)}$ and the set of parameters ω . Thus $\hat{y}^{(i)} = f(\vec{x}^{(i)}, \omega)$.

With $y^{(i)}$ we denoted the actual output, called *ground truth*, corresponding to $\vec{x}^{(i)} \in S^{train}$. If we have l output classes $y^{(i)}$ and $\hat{y}^{(i)}$ are vectors of size l , e.g. $y^{(i)} = \{y_1^{(i)}, y_2^{(i)}, \dots, y_l^{(i)}\}$.

Looking at a classification problem a simple cost function could be, the total number of incorrect classifications. In mathematical notation this is given by Equation 20:

$$C^{(i)}(\vec{x}^{(i)}, y^{(i)}) = \begin{cases} 0 & \text{if } \hat{y}^{(i)} = y^{(i)} \\ 1 & \text{if } \hat{y}^{(i)} \neq y^{(i)} \end{cases} \quad (20)$$

This means that if the predicted outcome is not the same as the ground truth we get a cost of 1. In the end this cost function gives us the total number of wrongly predicted outcomes. We are indeed interested in the total number of good and bad predictions. However, this is mostly not used as a cost function. If we use the sigmoid function to predict in probability whether the input belongs to a certain class or not the following can happen: for input sample i we get an output value of 0.9 and for another input sample j we get an output value of 0.55. Both output values (probabilities) are above 0.5 and thus the input belongs to that certain class. However, 0.9 is a way better prediction in comparison to 0.55, this means we want to have a cost function which uses this kind of knowledge.

Popular cost functions, throughout the history of ANNs, include; quadratic cost, cross-entropy cost. The quadratic cost function is defined as:

$$C^{(i)}(\vec{x}^{(i)}, y^{(i)}) = \frac{1}{2} \|\hat{y}^{(i)} - y^{(i)}\|_2^2 \quad (21)$$

which is the squared Euclidean distance between the output of the network and the ground-truth. Note: the factor $\frac{1}{2}$ is present because we use the derivative of the cost function¹⁸, the factor $\frac{1}{2}$ cancels out the factor 2 which we get extra from the derivative.

The quadratic cost function is nowadays not as popular as it was earlier, as it has several drawbacks when used for 'deep' neural networks. One of the reasons/drawbacks is the slow learning rate for classification problems. The slow learning rate will be explained using the gradient given at the end of this section, see Equation 23, e.g. the gradient is given by: $\nabla_{\omega} C^{(i)} = (\hat{y}^{(i)}(\omega) - y^{(i)}) \cdot \nabla_{\omega} \hat{y}^{(i)}(\omega)$ Where $\nabla_{\omega} \hat{y}^{(i)}(\omega)$ is the gradient of the output with respect to the parameters. In other words, this is the gradient of the activation function in the last layer. In Section 6 we discussed the issue of an unstable gradient. For all activation functions we discussed in Section 6 the derivatives of the activation functions are bounded, e.g. $|\nabla_{\omega} \hat{y}^{(i)}(\omega)| \leq 1 \forall$ activation functions. Furthermore, for classification problems the *ground truth* is either one or zero, e.g. $y_j^{(i)} = 1$ or $y_j^{(i)} = 0 \forall j$.

¹⁸ We use the derivative in the back-propagation algorithm, as we calculate the gradient of Equation 7 to update the parameters.

Next to this, for a classification problem we know, $\hat{y}_j^{(i)} \in [0, 1] \forall j$. Thus $|\hat{y}_j^{(i)} - y_j^{(i)}| \leq 1$ and if $\hat{y}_j^{(i)} \rightarrow y_j^{(i)}$ then $|\hat{y}_j^{(i)} - y_j^{(i)}| \rightarrow 0$.

In the end we have the following result: $|\nabla_{\omega} C^{(i)}| = |(\hat{y}^{(i)}(\omega) - y^{(i)}) \cdot \nabla_{\omega} \hat{y}^{(i)}(\omega)| \leq 1$. If the outcome prediction becomes better, e.g. $\hat{y}^{(i)} \rightarrow y^{(i)}$, we get $|\nabla_{\omega} C^{(i)}| \rightarrow 0$. In words this means that better predictions will lead to a smaller updating step for the parameters¹⁹. This phenomena is called *small learning rate*. In Chapter 3 of [55] the slow learning rate is also discussed and explained.

Another reason, why the quadratic cost function is not used anymore, is that the vectors \hat{y} and y can become high-dimensional if we have a large number of (output) classes. This will lead to a problem, because Euclidean distances in high-dimensional spaces are not very informative. Two instances can be very similar but their Euclidean distance can tell us something quite different. This problem lead to a very unstable cost function and it makes convergence to a good solution hard.

The cross-entropy cost function is defined as, when having l output classes:

$$C^{(i)}(\hat{x}^{(i)}, y^{(i)}) = - \sum_{j=1}^l y_j^{(i)} \log \hat{y}_j^{(i)} \quad (22)$$

Another way to look at the cross-entropy is as the distance between the probability distributions \hat{y} and y . We can interpret it as a (minus) log-likelihood for the data y_j , under a model \hat{y}_j . An explanation can be found in Appendix Section F. This automatically means that the output of the ANN, where we use cross-entropy as cost function, must be a probability, e.g. all outputs must sum up to precisely 1. To achieve this we can use the Softmax activation function, as proposed in Equation 17.

An advantage of the cross-entropy cost function is that it does not have to slow learning problem.

The gradient of the cross-entropy function is given by: $-\sum_{j=1}^l \frac{y_j^{(i)}}{\hat{y}_j^{(i)}(\omega)} \cdot \nabla_{\omega} \hat{y}_j^{(i)}(\omega)$, see Equation

24. The term $\frac{y_j^{(i)}}{\hat{y}_j^{(i)}}$ can be rewritten for classification problems;

$$\frac{y_j^{(i)}}{\hat{y}_j^{(i)}} = \begin{cases} 0 & \text{if } y_j^{(i)} = 0 \\ \frac{1}{\hat{y}_j^{(i)}} & \text{if } y_j^{(i)} = 1 \end{cases}$$

The term $\frac{1}{\hat{y}_j^{(i)}}$ is greater equal 1 for all values of $\hat{y}_j^{(i)}$, as $\hat{y}_j^{(i)} \in [0, 1]$ for classification problems.

Hence, the gradient of the cross-entropy function will not convergence to 0 if $\hat{y}_j^{(i)} \rightarrow y_j^{(i)}$. Therefore, the cross-entropy has not the slow learning problem. This is explained in more depth in Chapter 3 of [55].

Another cost function which can be used in combination with the Softmax activation function is the log-likelihood cost function according to Nielsen in [55]. For the log-likelihood cost function see [56].

In Section 5 we showed that for training an ANN we need to minimize the optimization problem as given in Equation 7. This *total cost function* is separable into two functions; i.e. the cost function and the regularization function. We need to minimize the total cost function with respect to the parameters, ω . To minimize this function, we need to have the gradient of the function with respect to the parameters. As the total cost function is separable into two functions we can also take the gradient separable, e.g. take the gradient of the cost function with respect to the parameters and take the gradient of the regularization function with respect to the parameters. Here we will give the gradient of the cost function with respect to the parameters for the cost

¹⁹ See Equation 8.

functions discussed.

In the cost functions given in this section we have that \hat{y} denotes the output of the network, hence \hat{y} is the network function and thus is a function of the parameters ω ; $\hat{y}(\omega)$ ²⁰.

The gradient for the quadratic cost is given by:

$$\nabla_{\omega} C^{(i)} = \nabla_{\omega} \frac{1}{2} \|\hat{y}^{(i)}(\omega) - y^{(i)}\|_2^2 = (\hat{y}^{(i)}(\omega) - y^{(i)}) \cdot \nabla_{\omega} \hat{y}^{(i)}(\omega) \quad (23)$$

Where $\nabla_{\omega} \hat{y}^{(i)}(\omega)$ is the gradient of the network function with respect to the parameters.

The gradient of the cross-entropy function is defined as:

$$\nabla_{\omega} C^{(i)} = \nabla_{\omega} \left(- \sum_{j=1}^l y_j^{(i)} \log \hat{y}_j^{(i)}(\omega) \right) = - \sum_{j=1}^l \frac{y_j^{(i)}}{\hat{y}_j^{(i)}(\omega)} \cdot \nabla_{\omega} \hat{y}_j^{(i)}(\omega) \quad (24)$$

Again $\nabla_{\omega} \hat{y}^{(i)}(\omega)$ is the gradient of the network function for the j -th output neuron with respect to the parameters.

²⁰ See Definition 3.

8 Regularization

In Equation 7 the term $J(\omega)$ is used. This is the regularization term of the total cost function. There are several regularization functions which could be used and we will discuss some of the most common ones.

First let us explain why there is need for such a regularization function. For this we need to look at *overfitting* and *overtraining*. We will start with a quote, which points out the problem; "The Nobel prizewinning physicist Enrico Fermi was once asked his opinion of a mathematical model which some of his colleagues had proposed for a physics problem. The model gave excellent agreement with experiment, however Fermi was skeptical. He asked for the amount of free parameters in the model. The answer was 'four'. Fermi replied: 'I remember my friend John van Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk.' "

The quote points out that models with a large amount of free parameters can describe a wide range of phenomena, e.g. can come up with complex decision boundaries. The model agrees very well with the available data, but that does not necessary mean that it is a good model. It could be that the amount of freedom in the model is such that it can describe almost any data point of the given set, without even capturing insights of the underlying dependence of input features. This means that the model will work well on the available data but will fail when we validate it on test data (from the same distributions). However, the goal is to make the model perform properly on both available (or train) data and new (or test) data.

There are two different kinds of 'errors' which could occur, namely; *overtraining* and *overfitting*. We define the *training-error* to be the the error (costs) of the training set. The error (costs) of the testing set, is define to be the *testing-error*. We can say that our model is *overtraining* when the training-error is decreasing for each iteration (after a specific number of iterations U) and that the testing-error is increasing at that point. In other words, the network is memorizing the training set instead of learning the underlying process. As a consequence the ANN predicts samples from the training set better each following iteration but the samples from the testing set are predicted worse. The samples from the testing set are from the same data set as the training ones, but they are different from one another, i.e. $S^{train} \in \Gamma, S^{test} \in \Gamma$ and $S^{train} \cap S^{test} = \emptyset$.

Overtraining, as the name already mentions, can occur if we train for too many iterations, but it can also occur when we have limited training data. If we train a network with limited training data even after a few iterations the model will start overtraining.

Suppose, to illustrate the concept of overtraining, we have m training samples, e.g. $|S^{train}| = m$ and we train the ANN for T iterations, where $T \gg m$. We use stochastic gradient descent to update the parameters. In each iteration a sample $x^{(i)} \in S^{train}$ is picked to update the parameters. We denoted ω_k to be the parameters after k iterations. If we update the parameters using sample $x^{(i)}$, e.g. we get ω_{k+1} , this will imply that: $|f^{(i)}(\omega_{k+1}) - y^{(i)}| \leq |f^{(i)}(\omega_k) - y^{(i)}|$. Here $f^{(i)}(\omega_k)$ is the output of the ANN for input sample i and parameters (ω_k) and $y^{(i)}$ is the ground-truth corresponding to input i . Throughout the training iterations we will have sample $x^{(i)}$ multiple times as input and eventually, after U iterations, $f^{(i)}(\omega_u) - y^{(i)} \approx 0 \forall u \geq U$. This will happen for all m input samples in S^{test} . Thus, after U iterations the parameters are not updated significantly anymore because the ANN 'remembers' the input- ground-truth pairs. The parameters $\omega_{u \geq U}$ will most likely not perform well on the testing set, as they learn the input-output pairs instead of the underlying structure.

Overfitting is based on *Occam's Razor*, or the principle of parsimony. This says that models and procedures only need to contain all that is necessary for modeling and nothing more. Overfitting is the use of models or procedures which violates this parsimony. This means that they include more terms than necessary or uses a more complicated approach than necessary. As a result the network will learn complicated relationships out of the training data. However, in the test data these complicated relationships will not occur, even if it is drawn from the same distribution as the training data.

Overfitting can also be described as the bias/variance dilemma. This is explained in [68] and [69]. Here we describe it in a short paragraph, but the interested reader should read one of the two articles provided.

Suppose we have input-output pairs (x_i, y_i) , where $y_i = \mu(x_i) + \epsilon$ with $\mu(x)$ an unknown function and ϵ is a noise term. The noise is distributed normally with mean 0 and variance σ^2 . The goal is to find an approximate function $\hat{\mu}(x)$.

We use the squared error cost function to find the approximate function, e.g. $(y - \hat{\mu}(x))^2$, see Equation 21.

We are now interested in the expected mean squared error (this give us an expectation on how well this network behaves) for this network, $E[(y - \hat{\mu}(x))^2]$. Rewriting this expectation we get:

$$E[(y - \hat{\mu}(x))^2] = E[y^2] - E[2 \cdot y \cdot \hat{\mu}(x)] + E[\hat{\mu}(x)^2]$$

Here $E[y^2] = Var[y] + E[y]^2$, $E[\hat{\mu}(x)^2] = Var[\hat{\mu}(x)] + E[\hat{\mu}(x)]^2$ and $E[2 \cdot y \cdot \hat{\mu}(x)] = 2 \cdot \mu(x) \cdot E[\hat{\mu}]$. Inserting this in the original equation, we get:

$$\begin{aligned} E[(y - \hat{\mu}(x))^2] &= Var[y] + E[y]^2 - 2 \cdot \mu(x) \cdot E[\hat{\mu}] + Var[\hat{\mu}(x)] + E[\hat{\mu}(x)]^2 \\ &= Var[y] + Var[\hat{\mu}(x)] + (E[y]^2 - 2 \cdot \mu(x) \cdot E[\hat{\mu}] + E[\hat{\mu}(x)]^2) \\ &= Var[y] + Var[\hat{\mu}(x)] + (\mu(x)^2 - 2 \cdot \mu(x) \cdot E[\hat{\mu}] + E[\hat{\mu}(x)]^2) \quad E[y] = \mu(x) \\ &= Var[y] + Var[\hat{\mu}(x)] + (\mu(x) - E[\hat{\mu}(x)])^2 \end{aligned}$$

The last step is to rewrite this equation in terms of bias and variance of the approximate function. The bias is given by; $Bias[\hat{\mu}(x)] = E[\hat{\mu}(x) - \mu(x)] = E[\hat{\mu}(x)] - E[\mu(x)] = E[\hat{\mu}(x) - \mu(x)]$. Furthermore $Var[y] = E[(y - E[y])^2] = E[(y - \mu(x))^2] = E[(\mu(x) + \epsilon - \mu(x))^2] = E[\epsilon^2] = Var[\epsilon] + E[\epsilon]^2 = \sigma^2$. Using this, gives the final result:

$$E[(y - \hat{\mu}(x))^2] = Var[\hat{\mu}(x)] + Bias[\hat{\mu}(x)]^2 + \sigma^2$$

Approximating an input set thus gives us a bias/variance trade-off, where overfitting corresponds to a high complexity model. A high complexity model(network) has a low bias but a high variance. The opposite of overfitting is *underfitting*, where the model(network) has a low complexity and thus a high bias and low variance.

Figure 15 illustrates the effect of the bias/variance trade-off on the expected error.

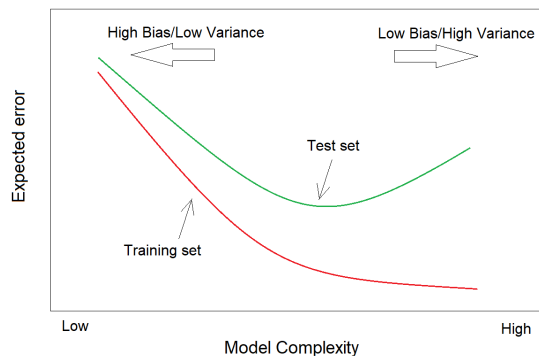


Fig. 15: The effect of the bias/variance trade-off on the expected error

Overfitting and overtraining are a major problem in artificial neural networks nowadays. A lot of parameters are present in modern networks. To train such a network efficiently we need a way to detect at which point U overtraining starts. Furthermore, we want methods which make the

model less sensitive for overfitting.

One of the 'easiest' and best methods to prevent our model from overfitting/overtraining is increasing the amount of training data. If there is enough training data available it is hard for the network to overfit/overtrain. However, in most cases not enough data is available. This means that we need other methods. These methods are called *regularization methods*. The goal of these methods is to make sure that the model, e.g. the parameters, does not become too complex. This is done by giving penalties to the parameters, in such a way that in the end only the parameters are used which give actual value to the data.

Popular regularization methods are; L^1 regularization, L^2 -norm regularization, soft weight sharing [51], Kullback-Leibler (KL) divergence [52], the dropout-technique [53] and data-augmentation. L^1 , L^2 and soft weight sharing are methods which introduce weight penalties. These methods prefer weights which have a small value or with have a value of zero.

L^1 regularization is given by the function:

$$\lambda J(\omega) = \lambda \sum_{i=1}^L |w_i| \quad \text{where } w_i \text{ is the weight vector for layer } i \quad (25)$$

This functions sums up all the absolute values of the weights. The regularization term is added to the cost function to give total costs²¹. So for L^1 regularization it is best if the (absolute value of the) weights are as low as possible. In that case the total costs will be namely lower. Hence small weights are preferred over high weights.

L^2 -norm regularization is given by:

$$\lambda J(\omega) = \lambda \sum_{i=1}^L w_i^2 \quad \text{where } w_i \text{ is the weight vector for layer } i \quad (26)$$

This is the sum of squares of the weights. Again lower weights make the total costs lower and hence they are preferred.

Nevertheless, there is a drawback with this method. It can favor two weak interactions over one strong one, as can be seen below. If a neuron receives input from two neurons which are highly correlated with each other, its behavior will be similar if we have weights w and 0 or weights $\frac{w}{2}$ and $\frac{w}{2}$. The penalty term favors the latter:

$$\left(\frac{w}{2}\right)^2 + \left(\frac{w}{2}\right)^2 < w^2 + 0^2$$

Instead we want as many zero weights as possible. If we have many zero-weights we can remove these connections and we get a simpler network structure. The simpler the network the less chance of overfitting and overtraining.

The idea of soft weight sharing is based on the idea of weight sharing. In weight sharing a single weight is shared among many connections in a network [33], [57]. The purpose is to get less adjustable weights (parameters) than the number of connections. Le Cun et al. [58] showed that this approach is effective for some problems. If parts of the problem are already understood, it is 'easier' to specify, in advance, which weights should be identical.

In soft weight sharing the distribution of weight values is modeled as a mixture of multiple gaussians. The model itself learns which weights should be tied together. This is done by adjusting the means and variances of the gaussians.

The soft weight sharing method tackles the problem of L^2 -norm regularization, as the gaussian idea will preform many near-zero weights without forcing large weights away from there needed

²¹ The total costs is given by: $C(\omega) = \frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)}) + \lambda J(\omega)$

values.

A formula as given in [51], for a mixture of a narrow (n) and a broad (b) gaussian :

$$p(w) = \pi_n \frac{1}{\sqrt{2\pi\sigma_n}} e^{-\frac{(w-\mu_n)^2}{2\sigma_n^2}} + \pi_b \frac{1}{\sqrt{2\pi\sigma_b}} e^{-\frac{(w-\mu_b)^2}{2\sigma_b^2}} \quad (27)$$

Here π_n and π_b are the mixing properties of the two gaussians and are therefore constrained to sum to 1. Furthermore σ^2 and μ are respectively the variance and mean of the gaussian.

The soft weight sharing regularization term will become, if we have j different gaussians:

$$-\sum_{i=1}^L \log \left[\sum_j \pi_j p_j(w_i) \right] \quad (28)$$

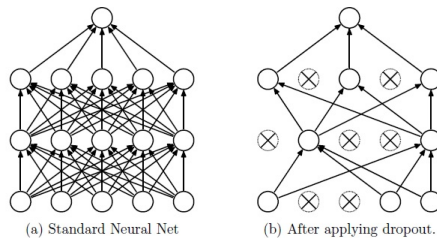
Kullback-Leibler divergence is given by:

$$J(\omega) = \sum_j y_j^{(i)} \ln \left(\frac{y_j^{(i)}}{\hat{y}_j^{(i)}} \right) \quad (29)$$

This function can be seen as a measure of how much information is lost whenever we approximate $y^{(i)}$ with $\hat{y}^{(i)}$. In other words, it is a measure of how one probability distribution diverges from a second expected probability distribution. A Kullback-Leibler divergence of 0 indicates that we can expect similar behavior (or the same) of the two distributions. Whereas a KL-divergence of 1 indicates that the two distributions behave in a different manner.

Before we explain the dropout-technique we need to explain something else. When we have unlimited computational expenses, the best way to make sure overfitting/overtraining does not occur (for a fixed-sized model) is to average the predictions of all possible settings of the parameters. Model combinations nearly always improve the performance of machine learning methods. However, if we are dealing with large neural networks, the idea of averaging the outputs of many separately trained networks is prohibitively expensive. Furthermore combining several models is most helpful when they are tested on different data. But we know, as stated earlier, that there is not enough data available.

Dropout is a technique which 'tackles' both issues [53]. It prevents the network from overfitting/overtraining and provides a way for combining exponentially many different neural network architectures in an efficient way. In the dropout technique, as expected by the name, we are dropping out units (neurons) in the neural network. Both the units in the hidden layers and in the visible (input, output) layers could be dropped. If we drop a unit, we temporarily remove the unit from the network, along with all its incoming and outgoing connections. An example is given in the article of Srivastava et al. [53]. See Figure 16 for this example.



Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Fig. 16: An example of the dropout method [53].

The units which are dropped are chosen randomly. In the simplest case we train a unit with probability p and drop it with probability $1 - p$, where $0 \leq p \leq 1$. If we apply dropout we get a "thinned" network. It consists of all the units and corresponding connections which survived the dropout, see Figure 16(b). If we have in total n units in our neural network, there are in total 2^n possible thinned neural networks. Because all these networks share weights, the total number of parameters is still $O(n^2)$.

Training a neural network with the use of dropout can be seen as training a collection of 2^n thinned networks, where the networks share their weights. Each network is trained rarely, or sometimes not at all.

For testing the network becomes an approximate average of all trained networks, because for testing we only use one neural network. The weights of this single network are calculated as follows; if a unit is retrained with probability p in the training-phase, the outgoing weights of that unit are multiplied by p for the network used in the testing-phase.

In [53] it is shown that using dropout gives a significantly lower error on a wide variety of classification problems compared to other regularization methods. By training on less complex network structures the chance of overfitting decreases, e.g. the number of free parameters is lower. By combining all these less complex networks we reconstruct the original network. Combining these networks will give the new parameters for the actual network, which still solve the problem we are dealing with. Moreover as given in [53] combining several methods nearly always improves the performance of machine learning methods. Therefore dropout is a good method to solve a classification problem and making sure overfitting occurs to a lesser extent [53].

The last method is data-augmentation. In this method we do not work with a fixed set of training instances, but instead we perform random transformations on each instance before processing it. These transformations, such as rotation and translation, will lead to the following; for the network it will be harder to overfit the training data. In other words data-augmentation can be seen as a way to learn specific in-variances in the network. The transformation is done on the input instances but the labels (output) stay the same. For classification problems, for example, this leads to the fact that we can force insensitivity to the type of transformation used in the data-augmentation.

Part II. Experimental Research

To illustrate the usefulness of artificial neural networks in race prediction we will make predictions for Formula One. Formula One is the highest race-podium for car racing.

First we will explain the data we used and how we prepared it. Next we will tell on which data-sets we tested it.

We also explain how to initialize an ANN. This is done by showing the influence of each step throughout initialization on the prediction for a given data-set.

We will finish the experiments with predicting the outcome of several races. In this section, we will also compare the results of the ANN with some other predicting methods.

9 Data preparation

For the required Formula One data we looked at the season 16/17. This season consists of 21 races. We only look at four different racers, because we needed to make this data-set by hand and these results only illustrate the power of ANNs. These racers are; *Lewis Hamilton*, *Max Verstappen*, *Felipe Massa* and *Jenson Button*. The following public features are chosen to be considered in our model:

Tab. 1: Features used for Formula One race prediction

Feature	Description
Circuit length	This represents the length of a circuit in kilometers
Number of laps	This represents the total number of laps of which the race consists
Weather	This represents the weather at the race day
Start grid	This represents the start position of each racer before the race
Recent form racer	This represents the last results of a racer
Recent form others	This represents the last results of the contestants
Best qualification lap	This represents the best lap-time of the racer during the qualification in seconds
Race result	This represents the actual outcome of the race

These features are chosen because they are public available, i.e. they can be found on the internet²². Furthermore, these are features which can be found for all races. Next to this, they can be found for races that happened before the season 16/17, which implies that our model can be tested on championships in the past and also for races in the present. Another reason is that these features are known before the race starts, which is needed to predict the race beforehand. Lastly, we believe these features are definitely needed to predict race results.

How are the features from Table 1, used in the race prediction model:

First we make a new feature called: *Total race length*, which is the *Circuit length* times the *Number of laps*. This new feature represents the duration of a race in kilometers. The feature will be normalized, we divide it by the maximum Total race length. We normalize the total race length due to its big number, looking at the other features the total race length is way bigger. This will lead to unfair weights and corresponding activations.

The weather is a binary variable; $Weather \in \{0, 1\}$. Where $Weather = 1$ means a rainy race and $Weather = 0$ means that it is not a rainy race. This structure is chosen because different circumstances are to be considered when there is rain. Difference in temperature will be less significant. The *Best qualification lap* is the best lap of a racer during his classification. The feature is a number which will be given in seconds. In our model this feature is used to get *Difference in qualification*, which is the difference between the best qualification lap of all 4 racers and the best qualification lap of a specific racer. This is calculated in the following way; let bq^r be the best qualification lap for racer r , $r = 1, 2, 3, 4$. With dbq^r we denote the difference in qualification for racer r and with $bestlap$ we denote the best qualification lap overall. Where $bestlap = \min\{bq^1, bq^2, bq^3, bq^4\}$. Then $dbq^r = |bq^r - bestlap|$.

The *Start grid* and *Race result* are given in the same way. We represent the start grid and the race result with binary variables. For each racer we have an array of size 4 which represents the start grid and an array of size 4 which represents the race result. At each position p , $p = 1, 2, 3, 4$ in the array it holds that the value $i(p)$ it is either 0 or 1. Furthermore $\sum_{p=1}^4 i(p) = 1$. If $i(p) = 1$ it holds that the racer start/finish at this position. An illustration of this is given in Table 2.

The only features which still need to be explained are the form features. For the form we look at the past 3 races. We choose this number because we believe that this number represents a racers

²² All this information can be found on: Wikipedia:FormulaOne2016

Tab. 2: Position example

1	0	0	0	Means: Position 1
0	1	0	0	Means: Position 2
0	0	1	0	Means: Position 3
0	0	0	1	Means: Position 4

form in a good way. Furthermore, if a racer gets a DNF^{23} which results in an end position of 4, this result will only be taken into account the next three races. Taking a bigger number of races for form will imply that a DNF will be taken into account for all those races. We believe this is not fair, therefore the choice for three races is made. Next to this, in horse-racing and dog-racing some articles also use a past 3-race average^([23],[24]).

The next step is to calculate the form; we will provide formulas for this. We have four racers; $r = 1, 2, 3, 4$ and 21 races; $t = 1, 2, \dots, 21$. Where $t = 1$ means race 1, $t = 2$ means race 2, etc. With $CurrentForm_t^r$ we denote the form of racer r at race t and with $CurrentFormOthers_t^r$ we denote the form of the contestants for racer r at race t . Next to this with $finish_t^r$ we denote the finish position of racer r at race t . This lead to the following formulas:

$$CurrentForm_t^r = \begin{cases} 0 & \text{if } t = 1 \\ finish_1^r & \text{if } t = 2 \\ \frac{\sum_{i=1}^2 finish_i^r}{2} & \text{if } t = 3 \\ \frac{\sum_{i=1}^3 finish_{t-i}^r}{3} & \text{if } t \geq 4 \end{cases}$$

$$CurrentFormOthers_t^r = \begin{cases} 0 & \text{if } t = 1 \\ \frac{\sum_{j \neq r} finish_1^j}{3} & \text{if } t = 2 \\ \frac{\sum_{j \neq r} \sum_{i=1}^2 finish_i^j}{6} & \text{if } t = 3 \\ \frac{\sum_{j \neq r} \sum_{i=1}^3 finish_{t-i}^j}{9} & \text{if } t \geq 4 \end{cases}$$

Let us look how all these features work for an actual race. For this we pick the *Australian Grand Prix*. The general race data for this grand prix is:

Tab. 3: General race data Australian Grand Prix

Circuit length	5.303 km
Number of laps	57
Weather	Partly cloudy; 22-23 °C

This is the first race of the season, therefore we do not have any numbers for the form of the racer itself and neither for form of the contestants. For this race they will zero, as stated by the formula. The qualification results, which determine the start grid are given in Table 4.

We need to mention something regarding the best qualification lap. It could be possible that the order of the best qualification lap does not correspond to the order of the start grid. In the Formula One season 16/17 the qualification consists of three different parts, called 'Q1, Q2 and Q3'. Where in Q1 all racers participate, in Q2 the best 16 racers of Q1 participate and in Q3 the best 10 racers of Q2 participate. The order of Q3 determines the start grid for the first 10 racers, the order in Q2 determines the start position for the positions 11 till 16 and start position 17-22 is determined from the order in Q1. This means that a racer can drive the best qualification lap of all racers but is not starting at pole position.

²³ DNF means: did not finish

Tab. 4: Qualification results Australian Grand Prix

Name	Start Position	Best qualification lap
Lewis Hamilton	1	83.837 seconds
Max Verstappen	2	85.434 seconds
Felipe Massa	3	85.582 seconds
Jenson Button	4	86.304 seconds

The end result for this race is given in Table 5.

Tab. 5: End result Australian Grand Prix

Name	End result
Lewis Hamilton	1
Max Verstappen	3
Felipe Massa	2
Jenson Button	4

The input of the Australian Grand Prix for the ANN-model is then:

Tab. 6: Input Australian Grand Prix as used for training the ANN

	Start grid				Race data		Form		Qualification
	Pos 1	Pos 2	Pos 3	Pos 4	Track length	Weather	Racer	Other	Best lap difference
Hamilton	1	0	0	0	0.97379	0	0	0	0
Verstappen	0	1	0	0	0.97379	0	0	0	1.597
Massa	0	0	1	0	0.97379	0	0	0	1.745
Button	0	0	0	1	0.97379	0	0	0	2.467

Note: The maximum total race length for all races is 310.408 kilometer. All track lengths will be divided by this number.

The corresponding output of this race is given in Table 7.

Tab. 7: Final Result Australian Grand Prix as used for training the ANN

	Race result			
	Pos 1	Pos 2	Pos 3	Pos 4
Lewis Hamilton	1	0	0	0
Max Verstappen	0	0	1	0
Felipe Massa	0	1	0	0
Jenson Button	0	0	0	1

To illustrate how the current form works in practice we will give an example. We consider race six of the championship, this is the Grand Prix of Monaco. To get the current form for all the riders, we need the end results of grand prix 5, 4 and 3. These results are given in Table 8.

The input for the Monaco Grand Prix then becomes: see Table 9.

Tab. 8: Past results

Name	Chinese Grand Prix	Russian Grand Prix	Spanish Grand Prix
Lewis Hamilton	2	1	4
Max Verstappen	3	4	1
Felipe Massa	1	2	2
Jenson Button	4	3	3

Tab. 9: Input Monaco Grand Prix as used for training the ANN

	Start grid				Race data		Form		Qualification
	Pos 1	Pos 2	Pos 3	Pos 4	Track length	Weather	Racer(*)	Other(**)	Best lap difference
Ham	1	0	0	0	0.83853	1	2.33333	2.55556	0
Ver	0	0	0	1	0.83853	1	2.66667	2.44444	8.525
Mas	0	0	1	0	0.83853	1	1.66667	2.77778	1.443
But	0	1	0	0	0.83853	1	3.33333	2.22222	1.41

We show the calculation of the form for Lewis Hamilton:

$$(*) \frac{2+1+4}{3} = \frac{7}{3} \rightarrow CurrentForm_6^1$$

$$(**) \frac{\frac{3+4+1}{3} + \frac{1+2+2}{3} + \frac{4+3+3}{3}}{3} = \frac{\frac{8}{3} + \frac{5}{3} + \frac{10}{3}}{3} = \frac{23}{9} \rightarrow CurrentFormOther_6^1$$

10 Different data-sets

The season 16/17 of the Formula One consists only of 21 races. All these races are different, so no input sample is the same. In total we have $21 \cdot 4 = 84$ different input samples, e.g. the information of a racer is an individual input sample, thus per race we have four input samples. This comes from the fact that the developed ANN can only learn per racer and not per race.

The races represent how well the different racers performed during the season. However, the data can sometimes not always fully represent the interrelation between the different riders. For example, if one of the riders has an engine failure, it will not finish the race and thus finish last. However, this rider could have been the best of the four. This makes working with only 21 races difficult, because the data can contain noise. This is not impossible though. We will still try to find an optimal network initialization for predicting these results based on the noisy data.

Another data-set which we want to investigate can be described by; the actual data-set where we add another 21 races. In this data-set we let them, e.g. the racers, race again on each track, thus total race length stays the same. We only adjust the qualification time, start-grid and end result. How this is done will be explained in Section 10.1. We thus simulate 21 new races.

From the four chosen riders, Hamilton is by far the best racer according to the results in the season 16/17. To illustrate the learning abilities of an ANN, we adjusted the end-results in the data-set with 42 races. In the newly made data-set Hamilton is a bad rain driver. This implies that whenever there is rain, Hamilton will finish in the back of the field. Meaning fourth place, unless somebody has a *DNF*. In this case Hamilton will finish in a higher position.

The data needs to be divided into a training set and a validation set, where the predictions on the validation set will tell us if the artificial neural network makes good predictions or not. Next to this, we need the validation set to make sure that our model is not overtraining itself, we do not want that it remembers the training input and predict the training results perfectly but when we use it on unseen data it predicts worse. Looking at the predictions in the validation set we can see if the model overtrains itself. Furthermore, the validation set is added to make sure that we do not overfit, e.g. choosing a too complex network structure.

When splitting the data in training and validation data, there needs to be a certain ratio between training data and validation data. If there are not enough training inputs, the ANN will most likely overtrain, but there must be enough validation input to test the predictions. We choose to train on the first 17 races of the season and to validate it on the last 4 races of the season. This implies a balance of approximately 80% training instances and 20% validation instances.

For the bigger data-sets we also use the last four races of the actual season 16/17 as validation and all others as training. This gives an division of: 90% training and 10% validation. We choose to have the same validation set, and not bigger, because we want to see if the 'randomly'²⁴ added races actually improve the prediction or not. Therefore, we need to validate it on the same set. Next to this, we want to validate our model on races which have actually occurred, and not on 'fake' races.

The validation set consists on the following grand prix:

- United States Grand Prix (Race 18)
- Mexican Grand Prix (Race 19)
- Brazilian Grand Prix (Race 20)
- Abu Dhabi Grand Prix (Race 21)

²⁴ The races we added according to Section 10.1.

The real results for the last four races of the season 16/17 are given in Table 10. We give this table because we will refer to it multiple times throughout the results.

Tab. 10: Results of the last four Grand Prix's of the season 16/17

	United States GP	Mexican GP	Brazilian GP	Abu Dhabi GP
Lewis Hamilton	1	1	1	1
Max Verstappen	4	2	2	2
Felipe Massa	2	3	4	3
Jenson Button	3	4	3	4

In the validation set the only rain race is the Brazilian Grand Prix. If we train and validate the data-set where Hamilton is a bad rain racer we need to adjust the end result of the Brazilian GP. In this case, the new outcome will become; Verstappen ends first, in front of Button and Massa and finishing last is Hamilton.

10.1 Enlarging the data-set

The following steps are done to enlarge the data-set:

First of all, as already said, we race again on each track. This means that we added another 21 races. In this new race, we leave the Circuit length and Number of laps the same, which means that the Total race length is equivalent. Instead of only racing once on a circuit we 'let' them race twice at each circuit. If we do this multiple times, it could show if a racer is good at a certain track or not. Another reason to add the same tracks again is due to the Formula One itself. Most of the Formula One seasons are raced on the same 21 tracks as they did in the season 16/17. We thus could have chosen to add the season 15/16 and even earlier seasons, however due to time constraints this is not done.

For the weather we look at how many times it has rained in the actual 21 races, say that this is z times. Then in the newly added races the probability of rain is $\frac{z}{21}$ and the probability of no rain is thus $1 - \frac{z}{21}$.

The best qualification lap for racer r on race/track t is taken from a normal distribution with mean= μ and variance= σ^2 . Where μ is the best qualification lap of racer r on track t in the actual data-set; bql^r . The variance is chosen to be 1 percent of the best qualification lap, thus $\sigma^2 = 0.01 \times \mu$. It applies for the newly added races that: $bgl^r(new) = \mathcal{N}(\mu, \sigma^2) = \mathcal{N}(bql^r, bql^r \times 0.01)$.

For the race result and the start grid of the race we have in total 24 different possibilities of ordering. We namely have 4 possible positions for each racer, which leads to a total of $4! = 24$ different orders. For each race we have 2 orders, namely the start grid order and the end result order. An order could for example be, Verstappen, Hamilton, Massa, Button. This means that Verstappen is on position one, Hamilton on position two, etc. In total we have 42 orders of *Hamilton, Verstappen, Massa, Button*. If we look more closely at the 42 orders we see that the following different orders occur, see Table 11. In the column 'Order' number 1 refers to Hamilton, 2 to Verstappen, 3 to Massa and 4 to Button.

Each of these different orders will be presented as a probability. Where the probability is given by: $\frac{\text{Number of times occur in total}}{42}$. If we now add races to the data-set with these probabilities the race will have a certain order for start position and for end result. Adding the start and finishing position in this way will give in expectation for Hamilton the same amount of first place starts and finishes as in the actual 21 races.

Tab. 11: Different start and finish orders in the actual 21 races of the season 16/17.

Order 1, 2, 3, 4 refers to Hamilton first, Verstappen second, Massa third and Button fourth. Similar order 2, 3, 1, 4 refers to Verstappen first, Massa second, Hamilton third and Button fourth.

Order	Number of times occur in total	# occur in qualification standings	# occur in race result
1,2,3,4	20	11	9
1,2,4,3	6	3	3
1,3,2,4	3	1	2
1,3,4,2	1	1	0
1,4,2,3	2	0	2
1,4,3,2	2	1	1
2,1,3,4	1	0	1
2,3,1,4	1	0	1
3,2,1,4	2	2	0
4,1,2,3	2	1	1
4,1,3,2	2	1	1

The recent form of a racer and the recent form of the others will be calculated exactly the same as before, we do not need to add some randomness to these features. This is because these features depend on the race-results of previous races.

If we take all steps explained above, we can make the data-set with each step bigger. Still the steps explained will leave the underlying structure of the data-set the same. In this way we hope that when we train an ANN on the bigger data-set it will find the same structure for the ANN as can be used for the actual data-set of 21 races.

11 Results structure

To show how the trained ANN performed we took the last four races of the season 16/17 as reference, see Table 10. This implies that training is done on all other races. Section 12 shows the steps taken to initialize the ANN. The purpose of that section is to show how different choices of activation functions, cost functions, etc. influence the prediction abilities of the ANN. In Section 13 the optimal network for each data-set is used to predict the outcome of the races in the validation set.

Results structure

We will first tell how we structure the results. It will contribute to the understanding of the results and how we derived them.

The first part, e.g. Section 12, is used to find the optimal network structure. In this section we try to find the optimal number of layers and corresponding number of neurons in each layer. We find the optimal number of layers by using a trail-and-error method. We start with 1 layer and train the model to see what the training costs are and the number of wrongly predicted outcomes in the training set. The next step is to do this for 2 layers, 3 layers etc. Using this trail-and-error method, where we consider all the different number of layers, will eventually give us the number of layers which perform best for the training set.

The second step is to find the best combination of activation function and cost function. For all possible combinations we use the network structure derived in the previous step. Again, comparison is based on the training costs and number of wrongly predicted outcomes for each combination. We choose the combination with the lowest costs.

After finding the best combination of activation and cost function, the next step is to look at a good regularization function and learning rate. In this step we also look at the results of the network for the validation set. It is important that these choices have a good impact on the predicted results in the validation set.

When we have found the optimal network structure, optimal functions and optimal learning rate we will give the end results. The (end) results obtained from the trained and validated ANN will be given in the following way:

First, we give some general results, see Table 12.

Tab. 12: Result-structure for the general results

Running time	Will be given in seconds
Number of iterations	Is a number, which represents the number of iterations to get the smallest cost
Number of wrong predictions	Gives the number of miss predictions in the training set for the 'optimal' network
Total Costs	Is the total cost, which is cost for miss predictions and cost given from the regularization function
Smallest value of wrong predictions	Gives the smallest number of miss predictions in the training set reached

Note: We have 'Number of wrong predictions' and 'Smallest value of wrong predictions'. They are not the same. The optimal network will be reached with the lowest 'Total Costs', so it could be possible that somewhere during training, we have less wrongly predicted outcomes than in the end. However the total costs could be higher, this will be explained with an example later²⁵.

We will give 'Number of wrong predictions', 'Total Costs' and 'Smallest value of wrong predictions'

²⁵ See Section 12, Tables: 19 and 20

for both the training set and the validation set.

Next to this table we give a graph where we set the number of iterations against the Total costs. In this graph we only show points where we decrease in total costs. At all these points we also calculated the validation costs. We show these costs also in a graph.

The most important part of the results is of course how well we perform on the reference races. We want to have a predicted finish position for each racer and in the end a predicted finishing order.

The ANN output for each racer are its individual probabilities on a certain position. The outcome of the ANN is given in Table 13.

Tab. 13: Probabilities of each racer to finish at a certain position

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	$p_h(1)$	$p_h(2)$	$p_h(3)$	$p_h(4)$
Max Verstappen	$p_v(1)$	$p_v(2)$	$p_v(3)$	$p_v(4)$
Felipe Massa	$p_m(1)$	$p_m(2)$	$p_m(3)$	$p_m(4)$
Jenson Button	$p_b(1)$	$p_b(2)$	$p_b(3)$	$p_b(4)$

As we are working with probabilities, the sum of each row needs to be exactly 1, thus as example: $p_h(1) + p_h(2) + p_h(3) + p_h(4) = 1$.

Note: this table gives individual probabilities; therefore $p_h(1) + p_v(1) + p_m(1) + p_b(1) = 1$ is not per definition the case. In Section 13.1 we will propose a method on how we go from these individual probabilities to an end result prediction.

Furthermore, the predicted results of the ANN will be compared with other methods which can be used for prediction. This is done to see if using a complex method such as the ANN really has benefits. The end result will be compared with two 'simple' prediction methods and one other machine learning approach.

The two simple prediction models are based on the start position and the recent form. The 'start position' prediction means the following: the start gird order is the finish order, e.g. the start position of racer r is the finish position of racer r . Thus, if racer r starts at position 2, then the end prediction will be position 2.

The 'recent form' prediction predicts the end results based on the order of the current form. First the current form of the racers will be ordered. The lowest current form correspond with the best racer ²⁶. Thus based on the ordering of the current form the end result is predicted.

The machine learning approach is: "Multiclass logistic regression". For this we use the "Azure Machine Learning Studio", where the multiclass logistic regression is a built-in function. This built-in function is used for classification problems with multiple output classes. More information about multiclass logistic regression can be found in [63], Chapter 4.3.4 of [64] and Chapter 7 in [65].

²⁶ See Equation 9

12 Explanation of how to initialize an Artificial Neural Network

The first step is to find a good network structure, in terms of number of layers. We test this in terms of error in training set, which is in this case only the cost function. We do not yet add the regularization function, because we first want to find a good network structure which will lead to good predictions. Once we have this network, we add the regularization function to optimize the network in terms of neurons.

This initialization is based on the data-set which consists of 42 races, e.g. the actual 21 races plus the simulated 21 races.

We initialize our model in the following way:

- Maximum number of iterations = 100000
- Activation function = Hyperbolic tangens
- Cost function = Cross-entropy
- Regularization function = Not used, because we are only interested in the predicted result
- Learning rate = 0.001
- Regularization rate = 0.01 Not used, because we do not have a regularization function

We have chosen to initialize in this way for the following reasons; as already mentioned the euclidean distance cost function does not always perform good. To be sure that we do not get any problems, we choose the cross-entropy cost function. The activation function is hyperbolic tangens. We think this is the best activation function that can be chosen.

To make sure that the last layer outputs a probability we use the Softmax function as activation function for the last layer.

Furthermore, we choose a descent amount of iterations and a small learning rate to make sure we do not end up in a local minimum.

Now we start adding layers to the network and see how it behaves in terms of number of wrongly predicted outcomes and corresponding costs.

The total cost is in this case not divided by the total number of samples. This is not necessary because we do not use a regularization function. So we get:

$$\text{Total costs} = \sum_{i=1}^N C^{(i)} \quad \text{where} \quad C^{(i)} = - \sum_{j=1}^4 y_j^{(i)} \log(\hat{y}_j^{(i)})$$

$y^{(i)}$ (the actual output) is an array of size 4, where at one position, j , a 1 is located and on the other positions a 0. This 1 corresponds to the end results of that specific rider. Therefore, costs only exist when $y_j^{(i)} = 1$.

$\hat{y}^{(i)}$ is the output of the ANN and is given as a row of Table 13. $\hat{y}_j^{(i)}$ corresponds to the j -th element of the row. This is the value given by the Softmax function.

To illustrate this; suppose we have as input sample Verstappen in the Australian Grand Prix. We know that Verstappen has finished the Australian Grand Prix as third. Thus, $y^{(Verstappen, Australian)} = [0, 0, 1, 0]$. The output of the ANN will be given by $\hat{y}^{(Verstappen, Australian)} = [p_1(v), p_2(v), p_3(v), p_4(v)]$. Thus, $C^{(Verstappen, Australian)} = \log(p_3(v))$.

From the results in Tables 14 and 15 we can expect that the 'optimal' network consists of 4,5,6 or 7 layers. After 7 layers the number of wrong predictions is rising again and the total costs are

Tab. 14: Behavior for different number of (hidden) layers (1)

	1-Layer	2-Layers	3-Layers	4-Layers	5-Layers
# Wrong predictions (Smallest # Wrong predictions)	70 (68)	52 (51)	47 (46)	43 (42)	42 (41)
Total costs	156.2834	136.7398	135.2950	123.1804	113.0944
# Iterations to reach smallest total costs	31491	70909	99972	94897	94967

Tab. 15: Behavior for different number of (hidden) layers (2)

	6-Layers	7-Layers	8-Layers	9-Layers	10-Layers
# Wrong predictions (Smallest # Wrong predictions)	44 (44)	42 (39)	62 (60)	73 (62)	67 (64)
Total costs	128.6641	120.7266	143.0668	159.3978	144.1272
# Iterations to reach smallest total costs	85742	47720	21906	11539	26217

rising as well. Furthermore, looking at the number of iterations, we can see that it stops learning relatively fast. It is searching for underlying structures which do not exist in this data-set. Next to this, we do not want to investigate a too complex network. Therefore, the choice is easily made when deciding to look at networks with at most 7 layers.

The next step is to investigate there performance of these networks for more iterations, i.e. we will let the ANN learn for a maximum of 200000 iterations. This lead to the following results:

Tab. 16: Behavior for different number of (hidden) layers (more iterations)

	4-Layers	5-Layers	6-Layers	7-Layers
# Wrong predictions (Smallest # Wrong predictions)	38 (37)	37 (37)	38 (38)	42 (42)
Total costs	124.9528	114.4131	128.6290	124.6389
# Iterations to reach smallest total costs	66855	93115	68719	66932

Giving the results from Table 16, we choose to focus mainly on a neural network with 5 layers, thus 4 hidden layers. For this network we will try different functions and choices for learning rates etc. From the table we can notice that it is best to choose a learning rate which is smaller. We can conclude this from ”# iterations to reach smallest costs” where all the values are far below the maximum number of iterations. It is namely possible that the learning rate is too big and the network diverges afterwards. Choosing a smaller learning rate makes this less predictable.

We choose this structure, e.g. 5 layers, because we can see that the number of wrong predictions is the lowest and the costs are lower than in other network structures. This implies that the distance between the probability distributions is the lowest when using 5 layers. This lets us believe that we can improve a 5-layered network the most.

Now that we have a 5-layered network, the next step is to find the best functions ²⁷ for the problem we are dealing with.

For the activation functions we can choose between; linear, sigmoid and hyperbolic tangens.

For the cost functions we can choose between; euclidean distance and cross-entropy.

For the regularization method we can choose between; L^1 regularization, L^2 regularization and KL divergence.

The first step in finding the best function is to find the best combination of activation and cost function. If we have the best combination of these functions we will add the regularization function to see which regularization method will improve the performance of the network the most.

We have 6 different combinations between activation and cost functions. We will test them with the following initialization:

²⁷ See Sections: 6, 7 and 8 for the formulas of all the functions.

- Maximum number of iterations = 200000
- Regularization function = Not used, because we are only interested in the predicted result
- Learning rate = 0.0005
- Regularization rate = 0.01 Not used, because we do not have a regularization function

We have chosen to take a learning rate which is two times smaller as in the last experiment. Again, we test for the smallest error in the training set, where we do not use any of the regularization methods yet.

The results are given below:

Tab. 17: Results for different combinations of activation and cost function (1)

	linear(*)/euclidean	linear(*)/cross-entropy	sigmoid/euclidean
# Wrong predictions (Smallest #)	50 (50)	52 (52)	68 (68)
Total costs	37.2434	131.5434	55.4306
# Iterations to reach smallest total costs	199959	90131	199994

Tab. 18: Results for different combinations of activation and cost function (2)

	sigmoid/cross-entropy	tangens/euclidean	tangens/cross-entropy
#Wrong predictions(Smallest#)	64 (64)	40 (39)	32 (32)
Total costs	172.7031	30.7138	114.5540
# Iterations to reach smallest total costs	199950	199915	177215

(*) for the linear activation function in combination with euclidean distance as cost function we needed to adjust the learning rate. The linear function is not bounded which leads to really big values after the activation. Therefore, we changed the activation function, we choose $a = 0.1$ instead of $a = 1$. Therefore, the derivative is smaller and we choose to increase the learning rate to 0.1. In the end this leads to a change of $0.1 \times 0.1 = 0.01$ each iterative step.

Note: we cannot compare the cost of the euclidean cost function with the cross-entropy cost function. These are different cost functions and therefore the costs are in a different range. We can however compare the number of wrongly predicted outcomes.

Furthermore note: The optimal number of layers can depend on the chosen combination of activation function/cost function. In our case we find the optimal number of layers with the hyperbolic tangens and cross-entropy. This can imply that other combinations will perform better on a different number of layers. However, following the theory discussed in the literature research of this work, the combination of hyperbolic tangens/cross-entropy works best for multi-layer networks which are not too 'deep'.

The conclusion which can be based on Table 18 is; using the hyperbolic tangens as an activation function and the cross-entropy as the cost function, is for this problem the optimal combination. The hyperbolic tangens activation function has the lowest "Total costs" and "Wrong predictions" for both cost functions. Another conclusion which can be made is that most models are still improving with more iterations. This implies that we can choose a higher maximum for the number of iterations. Next to this, we can look at the behavior for a different range of learning rates.

Now we have chosen the right structure and functions for this problem we will focus on the learning rate and the regularization function.

We will give a plot where we set the total costs against the learning rate. Besides this, for the regularization function we also take into account the results on the validation set. We are interested in both the training error and the validation error. Next to this, we are interested in the total number of wrongly predicted outcomes. However, we will give the most attention to costs. In general, the costs give a better insight in the performance of the network than the number of wrong predictions.

We will illustrate this behavior by the use of an example:

We have three racers and thus three different outcome positions; 1, 2, 3. We computed the probability that a racer will finish on one of these positions. These are the individual probabilities²⁸. The highest probability will imply that the racer will finish there. We give two different outcomes and show the difference. We use the cross-entropy function as cost function.

Tab. 19: Example 1

Predicted			Actual outcome			Correct
0.3	0.3	0.4	0	0	1	Yes
0.3	0.4	0.3	0	1	0	Yes
0.1	0.2	0.7	1	0	0	No

The cross-entropy cost for example 1 is: $-(1 \log(0.4) + 1 \log(0.4) + 1 \log(0.1)) = 4.14$. Whereas the number of wrongly predicted outcomes is 1.

Tab. 20: Example 2

Predicted			Actual outcome			Correct
0.1	0.1	0.8	0	0	1	Yes
0.4	0.3	0.3	0	1	0	No
0.4	0.5	0.1	1	0	0	No

The cross-entropy cost for this example is: $-(1 \log(0.8) + 1 \log(0.3) + 1 \log(0.4)) = 2.34$. Whereas the number of wrongly predicted outcomes is 2.

Looking at example 1 and example 2 we can see that in example 1 the two good predicted outcomes are not strong. They are below 0.5 and just 0.1 higher as the probabilities for the other positions. Next to this, the wrongly predicted outcome is bad, we predicted only 0.1 instead of 1. In example 2 the correctly predicted outcome is strong, as we predicted 0.8.

For most problems it is preferable to have strong outcomes, e.g. outcomes which are close to either one class or another class. Therefore, the focus is on the cost function and less on the number of wrongly predicted outcomes. The cost function (cross-entropy) tells us how well the probability distributions, of the predicted outcome and the real output, are the same, as stated in Section 7.

Now we will continue with finding the optimal learning rate for the Formula One problem. The model is now initialized as follows: 5-layers, hyperbolic tangens as activation function and cross-entropy as cost function. We will learn the model for 200000 iterations with different choices for the learning rate. The results are given in Figures 17 and 18. In these figures we show, for both the training and validation set, the (minimum) costs and the number of wrongly predicted outcomes per learning rate. Note that the number of samples in the training set is 152 and the

²⁸ see Table 13.

number of samples in the validation set is 16²⁹.

Looking at both figures the following conclusions can be made; for both the training set as the validation set a high learning rate (bigger as 0.01) means bad predictions, in terms of costs and number of wrong predictions. Therefore, our focus is at learning rates smaller than 0.01. However, we can also conclude that a learning rate too small, e.g. 0.0001 will lead to bad results. At all four graphs the minimum is at a learning rate of 0.0005. Therefore, we choose to use a learning rate of 0.0005. If we use a huge number of training iterations we can always choose a bigger learning rate in the beginning to speed up the process and in the end choose a smaller learning rate to optimize the parameters.

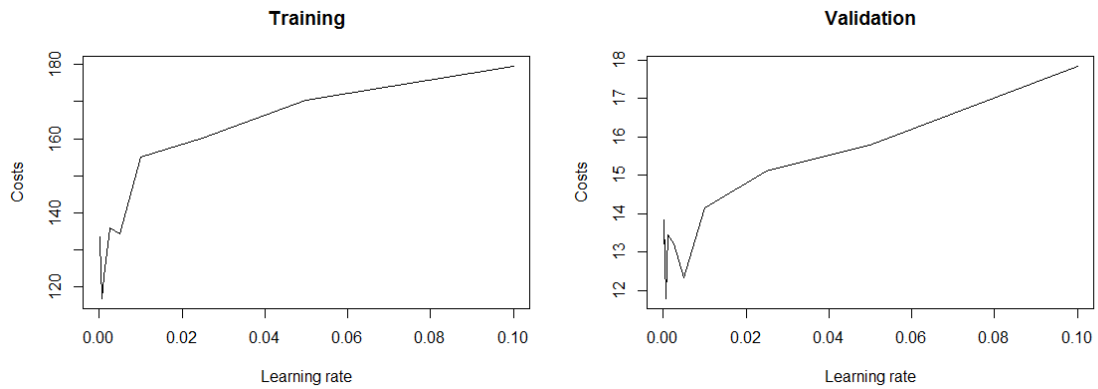


Fig. 17: Total costs for different learning rates. The costs are given by: $\sum_{i=1}^N C^{(i)} = -\sum_{i=1}^N \sum_{j=1}^4 y_j^{(i)} \log(\hat{y}_j^{(i)})$

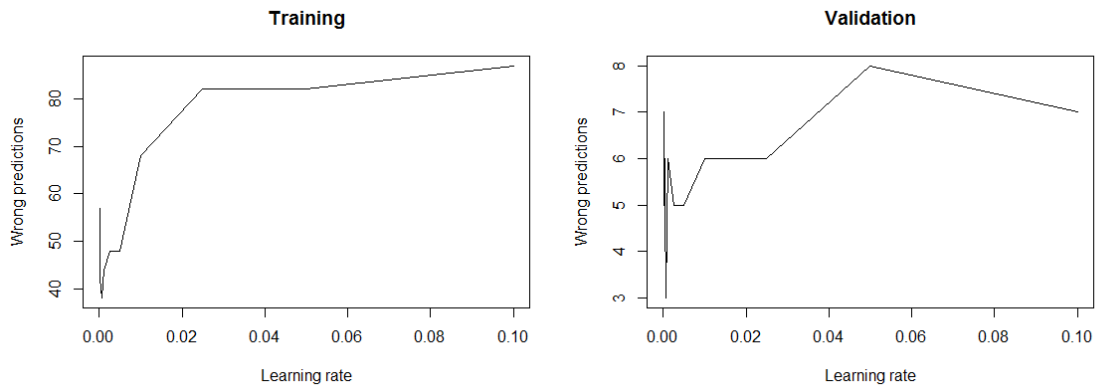


Fig. 18: Number of wrongly predicted outcomes for different learning rates

The next and final step is to find a suitable regularization function. Furthermore, we need to think about whether a regularization function is needed or not. We only have a small neural network, so it may not be necessary to use a regularization method. To see if we need a regularization

²⁹ In total we have 42 races, we have 38 races in the training set and 4 races in the validation set. Each race has four different input samples, hence $38 \cdot 4 = 152$ and $4 \cdot 4 = 16$.

method, we look at the predictions for the validation set. We compare the prediction results of the models with regularization with the model without regularization.

The results of this comparison are given in Table 21.

In this table 'Total Costs' is given by Equation 7:

$$\frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)}) + \lambda J(\omega)$$

and 'Costs per sample' is given by: $\frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)})$. The 'Costs per sample' are added because we want to compare the different regularization methods. The regularization methods all have a different range of output values and therefore we cannot compare these methods based on the 'Total Costs'.

Where $C^{(i)}(f^{(i)}, y^{(i)}) = -\sum_{j=1}^4 y_j^{(i)} \log(f_j^{(i)})$ and $J(\omega)$ the regularization function.

Looking at Table 21 we can conclude that, for the training set, KL divergence works best as regularization function. However, this gives the worst 'Costs per sample' for the validation set. The same holds the other way around, L^1 performs the best for validation set but the worst for the training set.

If we compare the results of the model without regularization to the model with regularization we see that it looks quite the same. 'KL divergence' and 'No regularization' have almost the same results. In comparison with L^1 and L^2 regularization, the model without regularization function performs significantly better at the training set. Therefore, we prefer a model without regularization. This will speed up the process, as the regularization function does not need to be calculated and the derivative of the regularization function does not need to be taken into account.

Tab. 21: Results using different regularization functions

		L^1	L^2	KL divergence	No regularization
Training	Total Costs	4.4755	5.3817	1.8552	0.7225
	Costs per sample	0.9242	0.9218	0.7362	0.7225
	Wrong predictions	59	56	35	31
Validation	Total Costs	4.3280	5.3481	1.0547	0.8999
	Costs per sample	0.8506	0.8883	0.9093	0.8999
	Wrong predictions	5	8	5	5

In the end we have made the following choices for the initialization of the ANN:

- Number of layers: 5
- Number of neuron per layer: 10 neurons per hidden layer, 4 neurons in the output layer
- Activation function: hyperbolic tangens
- Cost function: cross-entropy
- Regularization function: non
- Learning rate: 0.0005
- Maximum number of iterations: 350000

Fig. 19: Initialization Artificial Neural Network

13 Results Formula One

13.1 Results for data-set consisting of 42 races

In the previous sector we showed how to initialize the Artificial Neural Network for this data-set. The end initialization is given in Figure 19. The end results, using this initialization, are given in Table 22. These results correspond to the minimum validation costs.

We choose to use the weights/parameters which belong to the minimum value of the validation set, because we do not want that the model starts overtraining.

Tab. 22: General Results

Running time	3620.66
Number of iterations	139019
Number of wrong predictions (training)	35
Total Costs per sample (training)	0.7675
Smallest value of wrong predictions (training)	32
Number of wrong predictions (validation)	5
Total Costs per sample (validation)	0.7692
Smallest value of wrong predictions (validation)	5

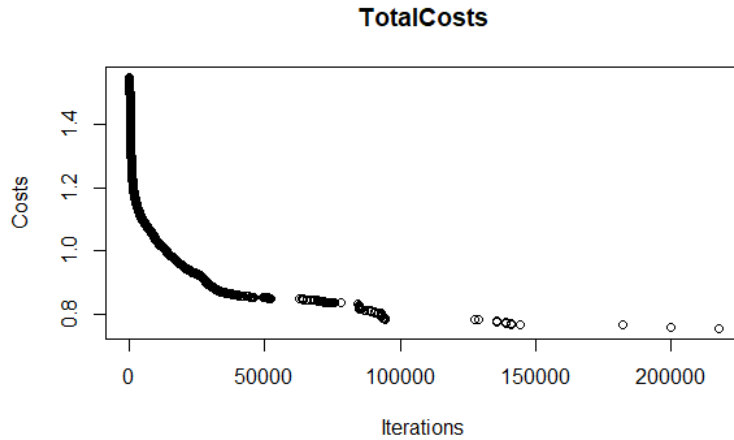


Fig. 20: Total training costs

Looking at both graphs, Figures 20 and 21³⁰, we can see that the learning algorithm of the ANN does not need the maximum number of iterations to reach the lowest costs. However, we let the ANN learn for all iterations, we can namely see multiple parts where the costs do not decrease. Therefore, it is hard to say if we should stop the algorithm before its maximum number of iterations. The same holds when looking at validation costs. We see that it increases around 50000 iterations, which could mean overtraining. However, around 125000 iterations it is decreasing again. Thus it is best to let the algorithm learn for all iterations.

³⁰ The total costs are given by: $\frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)}) + \lambda J(\omega) = \frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)})$ as we do not use a regularization method.

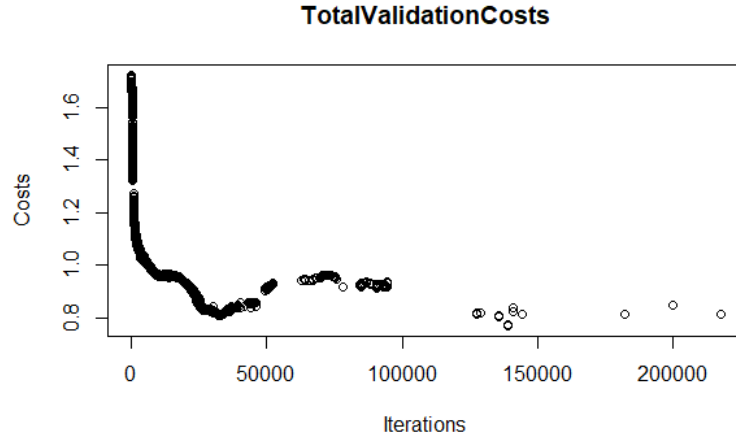


Fig. 21: Total validation costs

The fact that the costs per sample for both the training and the validation set are quite the same; 0.7675 and 0.7692, is a good thing. This means that the trained model works the same on the validation set as on the training set, e.g. the ANN has similar performance on the training set as on the validation set.

Now it is time to show the actual predictions for the races in the validation set. The first race we are looking at is the *United States Grand Prix*, according to Table 10 Lewis Hamilton wins this race, in front of Felipe Massa and Jenson Button. In last finished Max Verstappen.

The predicted individual probabilities, as predicted by the trained ANN, are given in Table 23. If we look at the individual probabilities we can say the following:

The expected finishing for Hamilton is first. For Verstappen the expected finishing position is first as well. For Massa third and Button fourth. These expected finishing positions are those with the highest probability.

Tab. 23: Probabilities of each racer for different positions United States Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123444	0.09625511	0.09625511	0.09625533
Max Verstappen	0.65485424	0.16789600	0.08862488	0.08862488
Felipe Massa	0.06631157	0.18898324	0.48997991	0.25472528
Jenson Button	0.06274518	0.06319911	0.41042806	0.46362765

However, now we have two racers who are predicted to finish first, therefore we need a model which turns these individual probabilities into an expected race result. To do this we propose the following method: first we take the sum of all the probabilities for each position. For position 1 this is: $0.7112344 + 0.65485424 + 0.06631157 + 0.06274518 = 1.49514543$, the probability that Hamilton will finish first given the opposition of Verstappen, Massa and Button is then: $\frac{0.7112344}{1.49514543} = 0.47569583$. In words this means that in a race consisting of Lewis Hamilton, Max Verstappen, Felipe Massa and Jenson Button the probability that Lewis Hamilton finishes at position 1 is the individual probability that Lewis Hamilton finishes first divided by the sum of the probabilities that each individual racer finishes first

If we do this for all positions and all riders we get the following predicted race result: Hamilton first, Verstappen second, Massa third and Button fourth. See Appendix Section G for the calculated ranking. The actual end result for this race is: Hamilton first, Massa second, Button third

and Verstappen last, as given by Table 10. So we only predicted Hamilton right. However, if we look more closely at the actual results we can see that Verstappen has a *DNF* due to engine failure. Taking this into account we can say that we predicted the outcome order right, Hamilton before Massa before Button.

Lets continue with the predicted results for the *Mexican Grand Prix*. According to Table 10 the actual outcome here is: Hamilton first, Verstappen second before Massa in third and last is Button.

The individual probabilities, the outcome of the trained ANN, are given in Table 24.

Tab. 24: Probabilities of each racer for different positions Mexican Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123446	0.09625512	0.09625512	0.09625530
Max Verstappen	0.09625492	0.71123302	0.09625531	0.09625675
Felipe Massa	0.06589593	0.18685370	0.48690874	0.26034163
Jenson Button	0.08563357	0.08564247	0.19597271	0.63275125

Following the proposed method, the predicted outcome for this race is: Hamilton first, Verstappen second, Massa third and last Button.

The output of our ANN for the *Brazilian Grand Prix* is given in Table 25.

Tab. 25: Probabilities of each racer for different positions Brazilian Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123443	0.09625511	0.09625511	0.09625534
Max Verstappen	0.08922853	0.65931460	0.14345704	0.10799983
Felipe Massa	0.08628710	0.24690843	0.40690874	0.25700534
Jenson Button	0.08118000	0.08240500	0.42435253	0.41206247

The prediction for this race is quite hard. The first two positions are clear; Hamilton will win and Verstappen will be second. But for the positions three and four it is not yet clear. We see that both Massa and Button have an individual result of finishing third. Our proposed method cannot really predict this battle either, but because Massa has an higher probability of finishing second we predict Massa on position three.

The actual end result for this race is: Hamilton, before Verstappen, before Button and in last Massa.

The last validation race is the *Grand Prix of Abu Dhabi*. The outcome of the ANN is given in Table 26.

Tab. 26: Probabilities of each racer for different positions Abu Dhabi Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71108102	0.09623435	0.09623435	0.09645027
Max Verstappen	0.09110787	0.67320118	0.10965993	0.12603102
Felipe Massa	0.06623365	0.18781291	0.48940418	0.25654926
Jenson Button	0.06695407	0.06707801	0.37124053	0.49472739

Here we predict the following outcome: Hamilton first, Verstappen second, Massa third and Button fourth.

The actual output is the same as our predicted output, see Table 10.

Now let us look how well the proposed ANN performed on the validation set in terms of costs and wrongly predicted outcomes.

Tab. 27: Number of wrongly predicted positions and costs for validation set

	# Wrong predictions	(Mean) Costs per sample
United States Grand Prix	3	1.3302
Mexican Grand Prix	0	0.4647
Brazilian Grand Prix	2	0.7433
Abu Dhabi Grand Prix	0	0.5388
Total	5	0.7692

From Table 27 we can concluded that we did a terrible job predicting the United States Grand Prix, we not only predicted three racers wrong, but also the costs per sample are high. If we do not take Verstappen into account, due to his *DNF*, and calculate the (mean) costs per sample for the other three we still get 0.9658. For the other three validation races we scored a better (mean) costs per sample than for the training set, the training set namely has a (mean) costs per sample of 0.7675. Thus for the other three races the proposed ANN did a descent job predicting the outcome.

Looking at how well we scored in terms of right predicted outcomes we can say the following: In the training set we wrongly predicted 35 outcomes, on a total of 152 outcomes, this is: $\frac{35}{152} = 0.23$, so we predicted 77% of the outcomes right. For the validation set this percentage is $\frac{16-5}{16} \cdot 100 = 69\%$.

Note: In the validation set for each race we predicted the outcome Hamilton first, Verstappen second, Massa third and Button fourth. If we look at the outcomes for the training set this is not true. The 'optimal weights' also give different orders as predicted outcome for races in the training set. Therefore, we can say that the ANN can predict different race outcomes.

To see how well the ANN predicts the outcome of races, we compare it with other methods. We look at the number of races predicted perfectly and the number of positions predicted right. The corresponding results are given in Table 28.

Tab. 28: Comparison with other methods

		Start Position	Current Form(*)	ANN
Training	Races predicted perfectly	11	10	18
	Positions predicted good	80	77	117
Validation	Races predicted perfectly	2	1	2
	Positions predicted good	11	8	11

(*) The current form only exists after the first race has been raced. So we do not predict an outcome based on the current form for the first race.

Concluding from the comparison with other simpler methods we can see that using an ANN indeed has benefits. For the validation set the model where we predicted the outcome based on start position gets the same results, but we can see that in the training set, the ANN predicts 47% of the races perfectly and 77% of the positions, in comparison with 28% and 53% using the start position. Therefore, in the training set the ANN predicts more accurate.

Next to a comparing with simpler methods, we also compare the performance of the ANN with

another machine learning approach. The results for the multiclass logistic regression model are given in Table 29. We see that the proposed ANN method and multiclass logistic regression have the same results for the validation set. For the training set the ANN predicts 117 outcomes good and the multiclass logistic regression only 99, a difference of 18.

Tab. 29: Results multiclass logistic regression

Training	Races predicted perfectly	17
	Positions predicted good	99
Validation	Races predicted perfectly	2
	Positions predicted good	11

To concluded this subsection:

- An ANN can be used to predict Formula One results.
- The proposed ANN predicted 77% of the training instances right and 69% in the validation set. Where it predicts in total 17 training races right and 2 validation races.
- The ANN performs better as simple prediction methods.
- The ANN performs better as the multiclass logistic regression method.

13.2 Results for data-set consisting of 42 races with Hamilton as a bad rain driver

In this part we will give the predicted race results for the data-set where we say that Hamilton is a bad rain driver. As can be seen, from the four chosen riders, Hamilton is by far the best racer. To show that an ANN can also learn different scenarios; we turn Hamilton into a bad rain driver.

We obtained the following network structure; we have 5 layers, where each layer, excluded the output layer, has 10 neurons. The output layer has 4 neurons, which corresponds to the number of possible outcome positions.

The following choices are made regarding the functions; the activation function is hyperbolic tangens, the cost function is the cross entropy and the regularization function is KL-divergence regularization.

Other choices that are made are; regularization parameter $\lambda = 0.01$ and learning rate $\alpha = 0.0005$ for the first 250000 iterations and $\alpha = 0.0001$ for the other iterations.

In total there are 350000 iterations allowed. This will give a total running time of about a hour. Getting this settings is done in the same way as discussed in Section 12. In the end this was the network initialization which predicted the best race results.

The general results belonging to this network are given in Table 30 and the graphs following this table.

Total Costs (per sample) is calculated with the following formula: $\frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)})$ and Total Costs Regularization is: $\frac{1}{N} \sum_{i=1}^N C^{(i)}(f^{(i)}, y^{(i)}) + \lambda J(\omega)$. Where $C^{(i)}$ is the cross entropy cost function and $J(\omega)$ is the KL divergence regularization function.

Tab. 30: General results for the rain data set

Running time	3572.01 seconds
Number of iterations	329420
Number of wrong predictions (training)	30
Total Costs per sample (training)	0.7280
Smallest value of wrong predictions (training)	30
Number of wrong predictions (validation)	5
Total Costs per sample (validation)	0.7955
Smallest value of wrong predictions (validation)	4

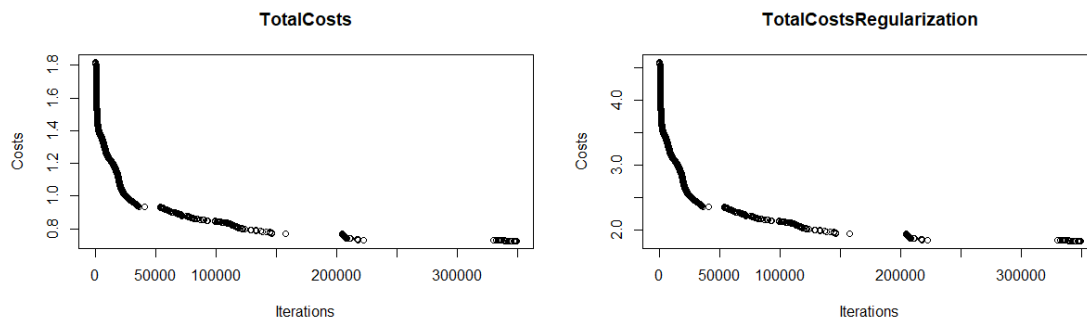


Fig. 22: Total costs per sample and total costs with regularization for the training set

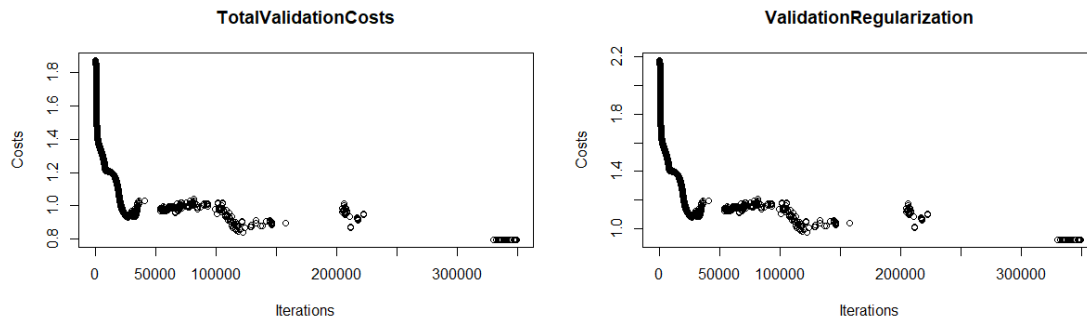


Fig. 23: Total costs per sample and total costs with regularization for the validation set

This network gives the following predicted individual probabilities for the races in the validation set: see Tables 31, 32, 33 and 34.

Tab. 31: Probabilities of each racer for different positions United States Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123191	0.09625855	0.09625477	0.09625477
Max Verstappen	0.07296270	0.53912546	0.11193716	0.27597468
Felipe Massa	0.07347508	0.30852477	0.54291152	0.07508862
Jenson Button	0.08090184	0.10032903	0.22098090	0.59778823

Tab. 32: Probabilities of each racer for different positions Mexican Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123191	0.09625855	0.09625477	0.09625477
Max Verstappen	0.07774602	0.57446968	0.12297882	0.22480549
Felipe Massa	0.06919993	0.34770365	0.51132214	0.07177428
Jenson Button	0.07995421	0.09989487	0.22936475	0.59078616

Tab. 33: Probabilities of each racer for different positions Brazilian Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.08408079	0.62127769	0.11694756	0.17769395
Max Verstappen	0.71123050	0.09626034	0.09625458	0.09625458
Felipe Massa	0.06028210	0.43400793	0.44542785	0.06028211
Jenson Button	0.08040165	0.24366081	0.59409231	0.08184523

Tab. 34: Probabilities of each racer for different positions Abu Dhabi Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123208	0.09625833	0.09625479	0.09625479
Max Verstappen	0.07732936	0.57139100	0.12657298	0.22470666
Felipe Massa	0.07350481	0.30829677	0.54313119	0.07506723
Jenson Button	0.08023788	0.10000798	0.22687199	0.59288216

The predicted race results for each race, using the individual probabilities and the proposed method, are then:

- United States GP: Hamilton before Verstappen, Massa third and Button last.
- Mexican GP: Hamilton first, Verstappen second, Massa third and fourth Button.
- Brazilian GP: Verstappen first, Massa second, Button third and last Hamilton.
- Abu Dhabi GP: Hamilton before Verstappen, in third Massa and last Button.

For the Brazilian GP, the only rain race in the validation set, the prediction was hard. It is easy to see that Verstappen will finish first. However, the other three positions are hard, we can see that the individual probability of Hamilton will correspond to position two, but his individual probability of finishing last is way higher than that of the others. Therefore, it is hard to predict Hamilton's position, but our proposed method gives probabilities where Massa and Button have higher chances to finish in the top three than Hamilton has, that is why we predict Hamilton in fourth.

Let us look how well we did in terms of costs and wrongly predicted outcomes for each race in the validation set, see Table 35.

From this table we can conclude that we predicted the Mexican and Abu Dhabi Grand Prix well. Not only did we predict each racer in the right place, we also have low costs per sample. For the proposed network it was hard to predict the rain race right. This is probably caused by the fact that we have a low amount of rain races in the training set.

For the training set we predicted 87% of the positions right. For the validation set this was:

Tab. 35: Number of wrongly predicted positions and costs for validation set

	# Wrong predictions	(Mean) Costs per sample
United States Grand Prix	3	1.0785
Mexican Grand Prix	0	0.5230
Brazilian Grand Prix	2	1.0723
Abu Dhabi Grand Prix	0	0.5084
Total	5	0.7955

69%.

Again we compare the results of the ANN with the other methods. This comparison is given in Table 36. From this table we can conclude that the ANN for both the training and validation set works better than some other 'simpler' methods. The ANN predicts 50% of the races in the validation set perfectly and in total 69% positions right, the best other method gives here 50% and 63%. For the training set the ANN gives the following percentages; 50% and 87%. The best order method gives; 45% and 64%.

Tab. 36: Comparison with other methods

		Start Position	Current Form(*)	Logistic Regression	ANN
Training	Races predicted perfectly	11	10	17	19
	Positions predicted good	80	77	97	132
Validation	Races predicted perfectly	2	1	2	2
	Positions predicted good	10	7	10	11

To conclude this subsection:

- An ANN can learn the difference between good and bad rain racers. However, the predicted contrast between good and bad rain drivers is not as good as expected. If there are more rain races the ANN can learn this contrast between good and bad rain racers better.
- The proposed ANN predicted 80% of the training instances right and 69% in the validation set, where it predicts in total 19 training races right and 2 validation races.
- The ANN performs better as simple prediction methods for both training and validation races.
- The ANN performs better as the multiclass logistic regression method for both training and validation races.

13.3 Results for the actual data-set, only consisting of 21 races

Here we show the results of the ANN when we only use the races raced in the season. We thus have 17 training races and 4 validation races. Thus, we have 68 training samples and 16 validations samples.

For this we obtained the following network structure, see Figure 24. The general results corresponding to this network are given in Table 37 and Figures 25 and 26.

- Number of layers: 5
- Number of neurons per layer: 10 neurons per hidden layer, 4 neurons in the output layer
- Activation function: hyperbolic tangens
- Cost function: cross-entropy
- Regularization function: KL-divergence
- Regularization rate: 0.1
- Learning rate: 0.001
- Maximum number of iterations: 200000

Fig. 24: Initialization Artificial Neural Network

Tab. 37: General results for the Formula 1 season 16/17

Running time	1043.17 seconds
Number of iterations	54140
Number of wrong predictions (training)	15
Total Costs per sample (training)	0.6767
Smallest value of wrong predictions (training)	12
Number of wrong predictions (validation)	4
Total Costs per sample (validation)	0.8192
Smallest value of wrong predictions (validation)	4

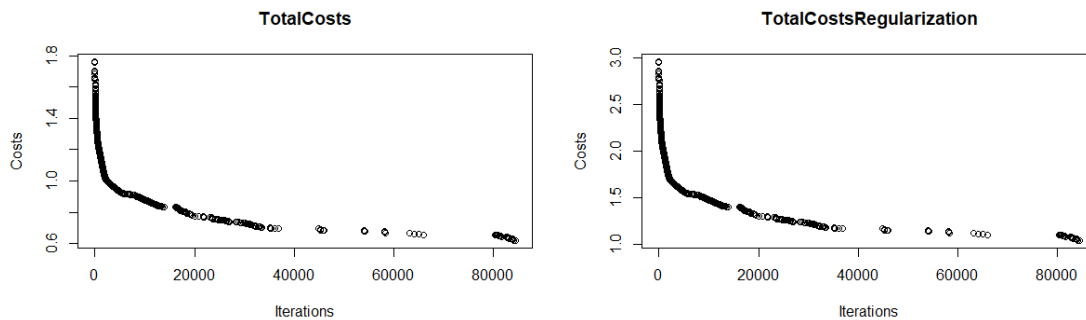


Fig. 25: Total costs per sample and total costs with regularization for the training set

We can see from this results that even with the use of a regularization function this network is probably overtraining. The costs per sample for the training set are decreasing until almost 0.6, however the minimum costs per sample for the validation set are 0.8192. This implies most definitely that the training set is too small to prevent overtraining. Ideally the costs per sample for the training and validation set are the same.

Now let us look at the predictions for the validation races, in the Tables 38, 39, 40 and 41 the individual probabilities are given.

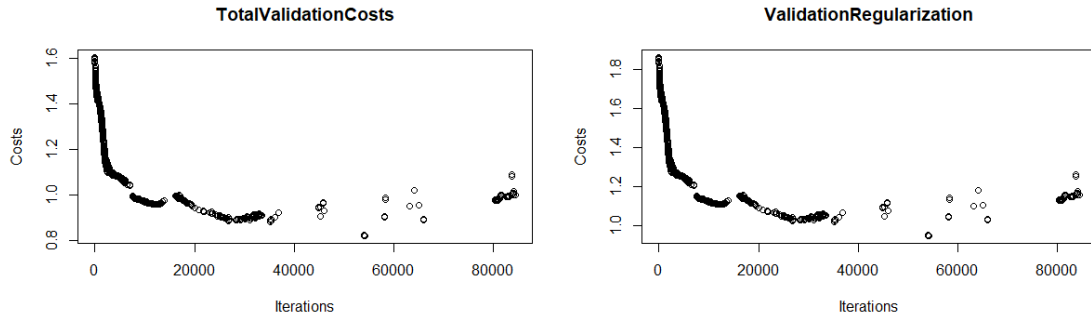


Fig. 26: Total costs per sample and total costs with regularization for the validation set

Tab. 38: Probabilities of each racer for different positions United States Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.68584817	0.12851279	0.09281946	0.09281959
Max Verstappen	0.05961818	0.05961606	0.44050641	0.44025934
Felipe Massa	0.08859197	0.08859197	0.65461100	0.16820506
Jenson Button	0.31879606	0.28086476	0.04777756	0.35256162

Tab. 39: Probabilities of each racer for different positions Mexican Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.68585841	0.12849977	0.09282084	0.09282097
Max Verstappen	0.12309109	0.69011507	0.093396922	0.09339693
Felipe Massa	0.08832020	0.08832020	0.65260293	0.17075666
Jenson Button	0.09589082	0.09589890	0.099667625	0.70854266

Tab. 40: Probabilities of each racer for different positions Brazilian Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.67317043	0.14462041	0.09110371	0.09110545
Max Verstappen	0.11662051	0.69520732	0.09408608	0.09408609
Felipe Massa	0.08837377	0.08837377	0.65299871	0.17025376
Jenson Button	0.05990208	0.05990214	0.43757592	0.44261986

Tab. 41: Probabilities of each racer for different positions Abu Dhabi Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.67313795	0.14466115	0.09109932	0.09110158
Max Verstappen	0.15256633	0.66691846	0.09025760	0.09025760
Felipe Massa	0.08855726	0.08855726	0.65435458	0.16853090
Jenson Button	0.09586538	0.09587300	0.09990695	0.70835467

The predicted race results for each race, using the individual probabilities and the proposed method, are then:

- United States GP: Hamilton before Button, Massa third and Verstappen last.
- Mexican GP: Hamilton first, Verstappen second, Massa third and fourth Button.
- Brazilian GP: Hamilton first, Verstappen second, Massa third and last Button.
- Abu Dhabi GP: Hamilton before Verstappen, in third Massa and last Button.

The costs and number of wrongly predicted outcomes³¹ per race are given in Table 42.

Tab. 42: Number of wrongly predicted positions and costs for validation set

	# Wrong predictions	(Mean) Costs per sample
United States Grand Prix	2	1.6656
Mexican Grand Prix	0	0.3798
Brazilian Grand Prix	2	0.8391
Abu Dhabi Grand Prix	0	0.3925
Total	4	0.8192

Looking at this table we can see that the ANN predicted the Mexican and Abu Dhabi Grand Prix real good. However, the United States Grand Prix is predicted really bad.

In Table 43 we compare the prediction results of the ANN with the other methods. Again the

Tab. 43: Comparison with other methods

		Start Position	Current Form(*)	Logistic Regression	ANN
Training	Races predicted perfectly	5	5	7	9
	Positions predicted good	35	38	43	53
Validation	Races predicted perfectly	2	1	2	2
	Positions predicted good	11	9	11	12

ANN performs the best of all methods for both the validation as the training set.

The conclusions which can be made based on this subsection are:

- Looking at only 21 races will probably lead to overtraining.
- The proposed ANN predicted 78% of the training instances right and 75% in the validation set, where it predicts in total 9 training races right and 2 validation races.
- In the validation set the ANN performs good on two races and really bad at one race.
- The ANN performs better as simple prediction methods for both training and validation races.
- The ANN performs better as the multiclass logistic regression method for both training and validation races.

³¹ See Table 10 for the real outcome of each race.

14 Conclusion

In this research we have focused ourselves on Artificial Neural Networks. We have acknowledged ourselves with the principles regarding ANNs. In which we have studied and discussed multiple important parts in the world of ANNs. An example is the influence of multiple layers and different numbers of neurons. We also discussed how to train an artificial neural network. Furthermore, we have emphasized some different activation- and cost functions. Lastly, we looked at regularization. Regularization is used to prevent the model from becoming too complex.

This knowledge is used to predict race results for the last four races of the Formula One season 16/17. We compared the predicted results of the ANN with some other prediction ideas.

We tested the performance of the ANN on three different data-sets, where the differences in data sets are the additional races added to the original 21 races of the season 16/17. The other data set is created to show the ability of the ANN to separate good and bad rain drivers.

The artificial neural network structure for this problem is obtained by testing different network structures and different combinations of cost- and activation functions. Furthermore, the model is tested for different learning rates and the use of a regularization function.

We obtained a relatively simple neural network with only 5 layers, e.g. 4 hidden layers. Following the theory, we could see that the hyperbolic tangens works best as activation function and the cross-entropy function as a cost function. Depending on which data-set we want to predict, using a regularization function can help us finding optimal weights and prevent us from overtraining.

For the two bigger data-sets the learning rate is twice as low as for the smaller data-set.

Looking at the first question, e.g. can ANNs be used to predict Formula One results, the answer is yes. ANNs can be used to predict race results and for the data-sets we considered the ANN outperforms the other methods tested. We showed that using a machine learning approach outperforms simple prediction models such as only looking at the start position or the last three race results. This implies that race results are not only determined by start position and recent form, but multiple other features are also important. We also compared the performance of the ANN with another machine learning approach called: multiclass logistic regression. The results support that ANNs are a better approach for race prediction as multiclass logistic regression.

Next the following points can be concluded. First of all, we see that the ANN always performs better on the training set than on the validation set. Both the costs per sample and the number of wrongly predicted outcomes are lower for the training set. This can imply overtraining on the training set and for the smallest data-set this is probably true. Here the costs per sample really difference for the bigger data-sets this difference is smaller. This brings us directly to another interesting result of our research. We see that adding another 21 races to the actual 21 races will lead to better results in the validation set. The added 21 races share a similar input structure, as discussed in Section 10.1. Therefore, we think that training an ANN only on one season with around 20 races is too hard. This also has to do with the fact that the data is noisy. Races that contain crashes influence the end results. Adding races, in this specific case however improved the prediction abilities. The data-set where we turned one of the racers a bad rain driver showed us that the ANN can indeed learn the difference between good and bad rain drivers. However, the result was not as strong as we would have expected. Probably this has to do with the fact that we only had three rain races in the training set. We therefore think that we need more rain races to emphasize this difference.

To conclude this section, we can see that Artificial Neural Networks can be used to predict Formula One results and it even outperforms some other methods. However, we cannot say that ANNs are the best method to use for race prediction. The prediction power of ANNs really depends on the number of samples and the chosen network structure plus initialization.

15 Discussion and further research

In this report the focus has been the use of artificial neural networks for Formula One prediction. Not available without agreement of the author and confidential company.

Another interesting research area in race prediction could be lap prediction. Meaning that we want to predict the order of the race at each lap. For these predictions, we can also take into account events happening during the race. The lap predictions are namely made during the race, so called real-time predictions. A little more on this topic is discussed in [12].

Looking more to the area of real-time predictions, a nice question could be to investigate whether artificial neural networks can be used to predict a racers strategy. Where of course the strategy of pit stopping can be investigated, e.g. at which lap does the racer need to make a pit-stop. Or even if the racer needs to speed up or even speed down.

In the experimental part of this research we only considered two cost functions and three regularization functions. There are however more cost functions and regularization functions. Some more attention to them could also be a good direction for further research.

As already discussed in this report, the data of the Formula One is a little noisy. We need to deal with accidents and penalties, which influence the outcome of the race. On a training data-set which only consists of seventeen races, one race which is influenced by an accident is already a big percentage. We could choose to filter the data in such a way that we only have a data-set with races where there are no accidents. For four racers this is doable, but if we take into account all racers it is impossible. In each Formula One race there are accidents. Therefore, we need to come up with a clever way of taking into account this 'noise'.

During this research we used multiple public available feature as inputs. However, there are of course more public available features which could tell us something about the race. For example we could take into account the brand of the car.

We could test the ANN with more input features, however more convenient would be doing a data pre-process step. In this pre-process step we take all features which could influence the race outcome and perform a regression test on them. With this test we can already see whether some of the features really have an influence on the outcome and which do not have any influence, e.g. we test for correlation. In [10], [11] a correlation test is done for NASCAR-races. This pre-process step will eventually speed up the learning abilities of the ANN.

In this research, the goal was to predict the last four races of the season 16/17. However, it would be nice to predict each race and not only the last four. For this, we could propose to train the network on a previous season and then predict the first race of the new season. Afterwards we put the first race of the new season in the training set and train the model again. We can repeat this for each race. A similar approach is used in [21] for the Dutch football league.

The main focus of this research has been on Artificial Neural Networks and in this research a lot is discovered and explained. However, a really important part, maybe even the most important part of ANNs, is not yet understood. There is no mathematical proof why ANNs can be used and why they can predict some problems in a good way. Focusing on this important question would be a rather hard but beautiful research, which will eventually solve a lot of unanswered questions in the world of ANNs.

Appendices

A The XOR-problem

The XOR, or Exclusive OR, is a logical operation that outputs true only when inputs differ. Or in other words; one of the inputs is true and the other is false.

The XOR-problem which is solved by the multi-layer perceptron has two inputs x_1 and x_2 . With corresponding truth table, see Table 44, where 1 means true and 0 corresponds to false.

Tab. 44: XOR truth table

Input		Output
x_1	x_2	
0	0	0
1	0	1
0	1	1
1	1	0

The following network structure is used to solve this problem: 2 – 2 – 1. The activation function ϕ is the threshold function. This is visualized in Figure 27a.

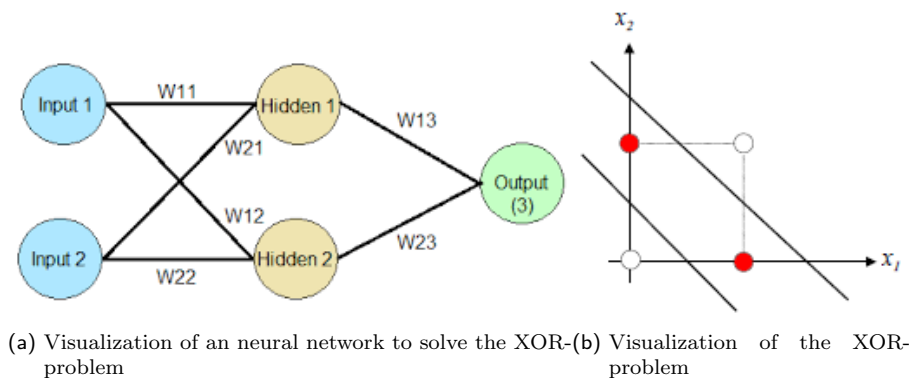


Fig. 27: The XOR-problem

In here we have three neurons with an activation function, e.g. 'Hidden 1', 'Hidden 2' and 'Output 1'. In 'Hidden 1' the threshold $b_1 = 0$ and in 'Hidden 2' the threshold $b_2 = 0$. In the 'Output' the threshold $b_3 = 0$. Choosing the weights in the following way will solve the XOR-problem: $w_{11} = 1$, $w_{12} = -1$, $w_{21} = -1$, $w_{22} = 1$, $w_{13} = 1$, $w_{23} = 1$.

For example look at input: $x_1 = 1$, $x_2 = 0$. This gives for 'Hidden 1': $x_1 \cdot w_{11} + x_2 \cdot w_{21} = 1 \cdot 1 + 0 \cdot -1 = 1$ the threshold here is 0, $1 > 0$ which implies that the value after the activation function is 1. For 'Hidden 2' the activation value is 0 as; $x_1 \cdot w_{12} + x_2 \cdot w_{22} = 1 \cdot -1 + 0 \cdot 1 = -1$ and $-1 > 0$ is false. The output is then: $Hidden1 \cdot w_{13} + Hidden2 \cdot w_{23} = 1 \cdot 1 + 0 \cdot 1 = 1$ which is bigger than the threshold 0 therefore the output is 1. All the calculations are given in Table 45.

Concluded can be that even for a simple logical operation as the XOR there is already need of a network which has multiple perceptrons and at least one hidden layer, as proven by Minsky and Papert in [30]. The XOR has only two classes, but there are three regions; as can be seen in Figure 27b. Thus only one perceptron cannot solve the XOR-problem.

Tab. 45: XOR-results

Input		Hidden 1	Hidden 2	Output
x_1	x_2			
0	0	$(0 \cdot 1 + 0 \cdot -1 = 0), 0 > 0 \rightarrow 0$	$(0 \cdot -1 + 0 \cdot 1 = 0), 0 > 0 \rightarrow 0$	$(0 \cdot 1 + 0 \cdot 1 = 0), 0 > 0 \rightarrow 0$
1	0	$(1 \cdot 1 + 0 \cdot -1 = 1), 1 > 0 \rightarrow 1$	$(1 \cdot -1 + 0 \cdot 1 = -1), -1 > 0 \rightarrow 0$	$(1 \cdot 1 + 0 \cdot 1 = 1), 1 > 0 \rightarrow 1$
0	1	$(0 \cdot 1 + 1 \cdot -1 = -1), -1 > 0 \rightarrow 0$	$(0 \cdot -1 + 1 \cdot 1 = 1), 1 > 0 \rightarrow 1$	$(0 \cdot 1 + 1 \cdot 1 = 1), 1 > 0 \rightarrow 1$
1	1	$(1 \cdot 1 + 1 \cdot -1 = 0), 0 > 0 \rightarrow 0$	$(1 \cdot -1 + 1 \cdot 1 = 0), 0 > 0 \rightarrow 0$	$(0 \cdot 1 + 0 \cdot 1 = 0), 0 > 0 \rightarrow 0$

B Derivative of the sigmoid activation function

Derivation of the derivative of the sigmoid activation function:

$$\begin{aligned}
 \frac{d}{dx}f(x) &= \frac{d}{dx}\sigma(x) \\
 &= \frac{d}{dx} \frac{1}{1 + e^{-x}} \\
 &= \frac{e^{-x}}{(1 + e^{-x})^2} \quad \text{quotient rule} \\
 &= \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} \\
 &= \sigma(x) \cdot \frac{1 + e^{-x} - 1}{1 + e^{-x}} \\
 &= \sigma(x) \cdot \left(\frac{1 + e^{-x}}{1 + e^{-x}} + \frac{-1}{1 + e^{-x}} \right) \\
 &= \sigma(x) \cdot (1 - \sigma(x))
 \end{aligned}$$

C Derivative of the hyperbolic tangens activation function

Derivation of the derivative of the hyperbolic tangens activation function:

$$\begin{aligned}
 \frac{d}{dx}f(x) &= \frac{d}{dx}\tanh(x) \\
 &= \frac{d}{dx} \frac{e^x - e^{-x}}{e^x + e^{-x}} \\
 &= \frac{(e^x + e^{-x}) \cdot (e^x + e^{-x}) - (e^x - e^{-x}) \cdot (e^x - e^{-x})}{(e^x + e^{-x})^2} \quad \text{quotient rule} \\
 &= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\
 &= 1 - \tanh(x)^2
 \end{aligned}$$

D Derivative of the Softmax activation function

If we look at Equation 17, this is the Softmax function, we can see that the input x is a vector. So we need to take the derivative with respect to each of those components, we are thus looking

for the partial derivatives.

$$\frac{\partial f_i(\vec{x})}{\partial x_j} = \frac{\partial \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}}}{\partial x_j}$$

Again we will use the quotient rule. For this we rename some terms:

$$g_i = e^{x_i}$$

$$h_i = \sum_{k=1}^K e^{x_k}$$

If we look at the derivatives of g_i and h_i with respect to x_j we can conclude the following:

$$\frac{\partial g_i}{\partial x_j} = \begin{cases} e^{x_i} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

$$\frac{\partial h_i}{\partial x_j} = e^{x_j} \quad \forall j$$

We will split this derivative up into two cases the first case is where $i = j$ and the second is where $i \neq j$.

- $i = j$

$$\begin{aligned} \frac{\partial f_i(\vec{x})}{\partial x_j} &= \frac{\partial \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}}}{\partial x_j} \\ &= \frac{\partial \frac{g_i}{h_i}}{\partial x_j} \\ &= \frac{e^{x_i} h_i - e^{x_j} g_i}{h_i^2} \quad e^{x_i} = g_i \quad \text{and} \quad e^{x_j} = g_j \\ &= \frac{g_i}{h_i} \cdot \frac{h_i - g_j}{h_i} \\ &= f_i(\vec{x})(1 - f_j(\vec{x})) \end{aligned}$$

- $i \neq j$

$$\begin{aligned} \frac{\partial f_i(\vec{x})}{\partial x_j} &= \frac{\partial \frac{e^{x_i}}{\sum_{k=1}^K e^{x_k}}}{\partial x_j} \\ &= \frac{\partial \frac{g_i}{h_i}}{\partial x_j} \\ &= \frac{0 \cdot h_i - e^{x_j} g_i}{h_i^2} \quad e^{x_j} = g_j \\ &= \frac{-g_j g_i}{h_i^2} \\ &= -\frac{g_j}{h_i} \frac{g_i}{h_i} \\ &= -f_j(\vec{x}) f_i(\vec{x}) \end{aligned}$$

Combining both cases and making use of the Kronecker delta gives us exactly in Equation 18.

E Explanation of the probabilities

Suppose we have a classification problem with two classes; y_1 and y_2 . We can express the posterior probability of y_1 using Bayes' theorem:

$$P(y_1|x) = \frac{P(x|y_1) \cdot P(y_1)}{P(x|y_1) \cdot P(y_1) + P(x|y_2) \cdot P(y_2)}$$

Dividing top and bottom of the right hand side by $P(x|y_1) \cdot P(y_1)$ gives:

$$P(y_1|x) = \frac{1}{1 + \frac{P(x|y_2) \cdot P(y_2)}{P(x|y_1) \cdot P(y_1)}}$$

Now we define z as the ratio of log posterior probabilities:

$$z = \ln \left(\frac{P(x|y_1) \cdot P(y_1)}{P(x|y_2) \cdot P(y_2)} \right)$$

Using this we can define the following:

$$P(y_1|x) = f(z) = \frac{1}{1 + e^{-z}}$$

Thus for a two class problem, where we have as output the sigmoid function we can say the following; using the sigmoid function as output can be seen as providing a posterior probability estimate.

If we have more than two classes we consider the following function, related to the log posterior probability of class y_k :

$$z_k = \ln(P(x|y_k)) \cdot P(y_k)$$

where z_k is the activation value of output k . Substituting this into Bayes' theorem, gives us the following expression for the posterior probability:

$$P(y_k|x) = \frac{e^{z_k}}{\sum_{j=1}^J e^{z_j}}$$

This function is precisely the Softmax function.

Using this as the activation function in the output layer for a multi-class problem, will guarantee that the J output values sum up to 1. That is a necessary condition for probability estimation.

Thus:

$$f_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

$$z_k = (w^T x)_k \quad \text{e.g. the input for neuron } k$$

F Explanation of the cross-entropy function

The cross-entropy function can be interpreted as the (minus) log-likelihood for the data y_j under a model \hat{y}_j .

Suppose we have some "hypothesis", which predicts for l classes $\{1, 2, \dots, l\}$ their hypothetical occurrence probabilities $\hat{y}_1, \hat{y}_2, \dots, \hat{y}_l$. In the end we observed (in reality) k_1 instances of class 1, k_2 instances of class 2, etc. According to the model the likelihood of this is:

$$P(\text{data}|\text{model}) = \hat{y}_1^{k_1} \hat{y}_2^{k_2} \dots \hat{y}_l^{k_l}$$

Now taking the (minus) logarithm gives:

$$-\log P(\text{data}|\text{model}) = -k_1 \log \hat{y}_1 - k_2 \log \hat{y}_2 - \dots - k_l \log \hat{y}_l = -\sum_{j=1}^l k_j \log \hat{y}_j$$

If we now divide by the number of observations $N = k_1 + k_2 + \dots + k_l$ and denote the empirical probabilities as $y_j = \frac{k_j}{N}$, we get precisely the cross-entropy function:

$$-\frac{1}{N} \log P(\text{data}|\text{model}) = -\frac{1}{N} \sum_{j=1}^l k_j \log \hat{y}_j = -\sum_{j=1}^l y_j \log \hat{y}_j$$

G Example of calculating the predicted race results

First the outcome table is given by Table 46:

Tab. 46: Probabilities of each racer for different position

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	$p_h(1)$	$p_h(2)$	$p_h(3)$	$p_h(4)$
Max Verstappen	$p_v(1)$	$p_v(2)$	$p_v(3)$	$p_v(4)$
Felipe Massa	$p_m(1)$	$p_m(2)$	$p_m(3)$	$p_m(4)$
Jenson Button	$p_b(1)$	$p_b(2)$	$p_b(3)$	$p_b(4)$

Because we are working with probabilities the sum of each row need to sum up to exactly 1, thus as example: $p_h(1) + p_h(2) + p_h(3) + p_h(4) = 1$.

The proposed model calculates the probabilities which are given in Table 47:

Tab. 47: Race Prediction, given all probabilities

	Lewis Hamilton	Max Verstappen	Felipe Massa	Jenson Button
Position 1	$p_1(h)$	$p_1(v)$	$p_1(m)$	$p_1(b)$
Position 2	$p_2(h)$	$p_2(v)$	$p_2(m)$	$p_2(b)$
Position 3	$p_3(h)$	$p_3(v)$	$p_3(m)$	$p_3(b)$
Position 4	$p_4(h)$	$p_4(v)$	$p_4(m)$	$p_4(b)$

Where:

$$p_i(j) = \frac{p_j(i)}{\sum_{t=\{h,v,m,b\}} p_t(i)} \quad \forall i = 1, 2, 3, 4 \ \& \ j = h, v, m, n$$

Let us now look how this works in practice. For this we take the predicted individual probabilities for the US Grand Prix. See Table 48.

Tab. 48: Probabilities of each racer for different positions United States Grand Prix

	Position 1	Position 2	Position 3	Position 4
Lewis Hamilton	0.71123444	0.09625511	0.09625511	0.09625533
Max Verstappen	0.65485424	0.16789600	0.08862488	0.08862488
Felipe Massa	0.06631157	0.18898324	0.48997991	0.25472528
Jenson Button	0.06274518	0.06319911	0.41042806	0.46362765

Following the proposed method Table 49 is made:

Tab. 49: Race Prediction United States Grand Prix, , given all probabilities

	Lewis Hamilton	Max Verstappen	Felipe Massa	Jenson Button
Position 1	0.47569583	0.43798698	0.04435125	0.04196594
Position 2	0.18642045	0.32516970	0.36601006	0.12239980
Position 3	0.08869085	0.08166025	0.45147456	0.37817434
Position 4	0.10656753	0.09811961	0.28201498	0.51329787

Now we look per position for the highest probability. For position 1 this implies Hamilton. For position 2 and 3 Massa, however for position 3 this probability is higher and Verstappen has a higher probability of finishing first. Thus position 3 is Massa and therefore position 2 is Verstappen. Position four is Button.

The predicted outcome is now thus: Hamilton, Verstappen, Massa, Button.

References

- [1] D.A. Harville; *Assigning Probabilities to the Outcomes of Multi-Entry Competitions*; Journal of the American Statistical Association Vol. 68, No.342, pp. 312-316, Jun. 1973.
- [2] R. Sauer; *The Economics of Wagering Markets*; Journal of Economic Literature Vol. 36 No. 4, pp. 2021-2064, Dec. 1998
- [3] W.T. Ziemba and D.B. Hausch; *Beat the Racetrack*; Harcourt Brace Jovanovich, 1984
- [4] J.W.Pratt; *Risk Aversion in the Small and in the Large*; Econometrica, Vol. 32, No. 1/2 (Jan.-Apr., 1964), pp. 122-136
- [5] K.J. Arrow; *Aspects of the theory of Risk Bearing*; Helsinki, Yrjo Jahnssonin Saatio, 1965
- [6] R. Sobel and T. Raines; *An examination of the empirical derivatives of the favourite-longshot bias in racetrack betting*; Applied Economics Vol. 35, No. 4, pp. 371-385, 2003.
- [7] M. Cain, D. Law and D. Peel; *The Favourite-Longshot Bias, Bookmaker Margins and Insider Trading in a Variety of Betting Markets*; Bulletin of Economic Research Vol. 55, No. 3, pp. 263-273, July 2003.
- [8] T. Graves, C.S. Reese and M. Fitzgerald; *Hierarchical models for permutations: Analysis of auto racing results*; Journal of the American Statistical Association 2003; 98:44282–44291.
- [9] C. Depken and L. Mackey; *Driver success in the Sprint Cup series: The impact of multi-car teams*; Social Science Research Network. Available online at: <http://ssrn.com/abstract=1442015> (Last accessed on 29 October, 2017).
- [10] M. Allender; *Predicting The Outcome Of NASCAR Races: The Role Of Driver Experience*; Journal of Business & Economics Research, Vol. 6, No. 3, pp. 79-84, March 2008
- [11] C.B. Pfitzner and T.D. Rishel; *Do Reliable Predictors Exists for the Outcomes of NASCAR Races?*; The Sport Journal, Vol. 8, No. 2, 2005
- [12] T. Tulabandhula and R. Cynthia; *Tire Changes, Fresh Air, and Yellow Flags: Challenges in Predictive Analytics for Professional Racing*; Big Data 2.2 (2014): 97–112. © 2012 Mary Ann Liebert, Inc
- [13] H. Drucker, C.J.C. Burges, L. Kaufman, et al.; *Support vector regression machines*; Advances in Neural Information Processing Systems 1997, 155–161.
- [14] R. Tibshirani; *Regression shrinkage and selection via the LASSO*; Journal of the Royal Statistical Society Series B (Methodological) 1996; 58:267–288.
- [15] M. Haghighat, H. Rastegari and N. Nourafza; *A Review of Data Mining Techniques for Results Prediction in Sports*; Advances in Computer Science: an International Journal, Vol. 2, Issue 5, No. 6, pp. 7-12, November 2013
- [16] C.K. Leung and K.W. Joseph; *Sports data mining: predicting results for the college football games*; Procedia Computer Science 35, pp. 710-719, 2014
- [17] K.Przednowek. J.Iskra and K.H.Przednowek; *Predictive modeling in 400-metres hurdles races*; Proceedings of the 2nd International Congress on Sports Sciences Research and Technology Support 2014, Pages 137-144
- [18] R.P. Schumaker and J.W. Johnson; *An Investigation of SVM Regression to Predict Longshot Greyhound Races*
- [19] M.C. Purucker; *Neural network quarterbacking*; IEEE Potentials, 15(1996), pp. 9-15

- [20] J. Kahn; *Neural network prediction of NFL football games*; World Wide Web Electronic Publication, 2003 (2003), pp. 9-15
- [21] N. Tax and Y. Joustra; *Predicting The Dutch Football Competition Using Public Data: A Machine Learning Approach*; Transactions on knowledge and data engineering, Vol. 10, No. 10, pp. 1-13, 2005
- [22] R.P. Bunker and F. Thabtah; *A machine learning framework for sport result prediction*; Applied Computing and Informatics, issn. 2210-8327, 2017
- [23] H. Chen et al.; *Expert Prediction, Symbolic Learning, and Neural Networks: An Experiment on Greyhound Racing*; 1994-12, 9(6):21-27 IEEE Expert
- [24] U. Johansson and C. Sonstrod; *Neural Networks Mine for Gold at the Greyhound Track*; International Joint Conference on Neural Networks, Portland, OR. (2003)
- [25] J. Williams and Y. Li; *A Case Study Using Neural Networks Algorithms: Horse Racing Predictions In Jamaica*; Proceedings of the 2008 International Conference on Artificial Intelligence, ICAI 2008, July 14-17, 2008, Las Vegas, Nevada, USA
- [26] E. Davoodi and A.R. Khanteymoori; *Horse racing prediction using artificial neural networks*; NN'10/EC'10/FS'10 Proceedings of the 11th WSEAS international conference on neural networks and 11th WSEAS international conference on evolutionary computing and 11th WSEAS international conference on Fuzzy systems; pp. 155-160, Jun. 2010
- [27] Anil K.Jain, Jianchang Mao and K.M.Mohiuddin; *Artificial Neural Networks: A Tutorial*; Computer March 1996
- [28] W. S. McCulloch and W. Pitts; *A logical calculus of the ideas immanent in nervous activity*; The bulletin of mathematical biophysics, vol. 5, no. 4, pp. 115-133, 1943
- [29] F. Rosenblatt; *The perceptron, a perceiving and recognizing automaton Project Para*; Cornell Aeronautical Laboratory, 1957
- [30] M. Minsky and S. Papert; *Perceptrons*; MIT press, 1988.
- [31] P. Werbos; *Beyond regression: New tools for prediction and analysis in the behavioral sciences*; Ph.D. dissertation, 1975.
- [32] P. Werbos; *Applications of advances in nonlinear sensitivity analysis*; System modeling and optimization. Springer, 1982, pp. 762–770.
- [33] D.E. Rumelhart and J.L. McClelland; *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*; MIT Press, Cambridge, Mass., 1986.
- [34] J. Schmidhuber; *Deep learning in neural networks: An overview* Neural networks, Vol. 61, pp. 85–117, 2015.
- [35] P.Melin and O.Castillo; *Book: Hybrid Intelligent Systems for Pattern Recognition Using Soft Computing: An Evolutionary Approach for Neural Networks and Fuzzy Systems - Chapter 5: Unsupervised Learning Neural Networks*; Springer Berlin Heidelberg, pp. 85-107, 2005
- [36] A.K. Jain, M.N. Murty and P.J. Flynn; *Data clustering: A review*; ACM Computing Surveys Vol. 31, No. 3, pp. 264-323, 1999
- [37] M. Peemen, B. Mesman and H. Corporaal; *Speed sign detection and recognition by convolutional neural networks*; Proceedings of the 8th International Automotive Congress, pp. 162-170
- [38] S. Mallat; *Understanding deep convolutional networks*; Phil. Trans. R. Soc. A, Vol. 374, No. 2065, 2016

- [39] M. D. Zeiler and R. Fergus; *Visualizing and understanding convolutional networks*; in European conference on computer vision. Springer, 2014, pp. 818–833.
- [40] G. Cybenko; *Approximation by Superpositions of a Sigmoidal Function*; Mathematics of Control, Signals, and Systems , pp. 303-314, 1989
- [41] K. Hornik, M. Stinchcombe, H. White; *Multilayer Feedforward Networks are Universal Approximators*; Neural Networks, Vol. 2, pp 359-366, 1989
- [42] M. A. Sartori and P. J. Antsaklis; *A simple method to derive bounds on the size and to train multilayer neural networks*; IEEE Transactions on Neural Networks, Vol. 2, pp. 467–471, 1991.
- [43] Guang-Bin Huang and Haroon A. Babri; *Upper Bounds on the Number of Hidden Neurons in Feedforward Networks with Arbitrary Bounded Nonlinear Activation Functions*; IEEE Transactions on Neural Networks, Vol. 9, No. 1, 1998
- [44] Yoshua Bengio and Yann Le Cun; *Scaling learning algorithms towards AI*; In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, Large Scale Kernel Machines. MIT Press, 2007.
- [45] Yoshua Bengio; *Learning deep architectures for AI*; Technical Report 1312, Universite de Montreal, dept. IRO, 2007.
- [46] K.G. Sheela and S.N. Deepa; *Review on Methods to Fix Number of Hidden Neurons in Neural Networks*; Mathematical Problems in Engineering Volume 2013
- [47] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio; *An empirical evaluation of deep architectures on problems with many factors of variation*; In Zoubin Ghahramani, editor, Twenty-fourth International Conference on Machine Learning (ICML 2007), pages 473–480. Omnipress, 2007. URL <http://www.machinelearning.org/proceedings/icml2007/papers/331.pdf>.
- [48] Hugo Larochelle, Yoshua Bengio, Jerome Louradour and Pascal Lamblin; *Exploring Strategies for Training Deep Neural Networks*; Journal of Machine Learning Research 1 (2009) 1-40
- [49] Douglas M. Hawkins; *The Problem of Overfitting*; J. Chem. Inf. Comput. Sci. 2004, 44, pp. 1-12
- [50] Peter Auer, Mark Herbster, and Manfred K. Warmuth; *Exponentially many local minima for single neurons*; In M. Mozer, D. S. Touretzky, and M. Perrone, editors, Advances in Neural Information Processing System 8, pp. 315–322. MIT Press, Cambridge, MA, 1996.
- [51] S.J. Nowlan and G.E. Hinton; *Simplifying neural networks by soft weight-sharing*; Neural Computation, 4(4), 1992
- [52] S. Kullback and R. A. Leibler; *On information and sufficiency*; The annals of mathematical statistics, Vol. 22, No. 1, pp. 79–86, 1951.
- [53] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov; *Dropout: a simple way to prevent neural networks from overfitting*; Journal of Machine Learning Research, Vol. 15, No. 1, pp. 1929–1958, 2014.
- [54] Igor V. Tetko, David J. Livingstone and Alexander I. Luik; *Neural Network Studies. 1. Comparison of Overfitting and Overtraining*; J. Chem. Inf. Comput. Sci. 1995, 35, pp. 826-833
- [55] Michael A. Nielsen; *Neural Networks and Deep Learning*; Determination Press, 2015
- [56] M.D. Richard and R.P. Lippmann; *Neural Network Classifiers Estimate Bayesian a posteriori Probabilities*; Neural Computation, Vol. 3, No. 4, pp. 461-483, 1991
- [57] Y. LeCun; *Generalization and Network Design Strategies*; Tech. Rep. CRG-TR-89-4

- [58] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard and L.D. Jackel; *Handwritten digit recognition with a back-propagation network*; Advances in Neural Information Processing Systems 2, pp. 396-404, 1990
- [59] Y.LeCun, J.Denker, S.Solla, R.E. Howard and L.D. Jackel; *Optimal Brain Damage*; Advances in Neural Information Processing Systems 2, pp. 598-605, 1990
- [60] Y.LeCun, L.Botton,G.B. Orr, K. Muller; *Efficient BackProp*; Neural Networks: Tricks of the Trade pp. 9-50 1998
- [61] Y. Bengio; *Practical recommendations for gradient-based training of deep architectures*; Neural Networks: Tricks of the Trade. Springer Berlin Heidelberg, 2012. pp. 437-478
- [62] X. Glorot and Y. Bengio; *Understanding the difficulty of training deep feedforward neural networks*; International conference on artificial intelligence and statistics. 2010
- [63] W.A. Bergerud; *Introduction to Regression Models: with worked forestry examples*; Biom. Info. Hand. 7. Res. Br., B.C. Min. For., Victoria, B.C. Work. Pap. 26/1996
- [64] C.M. Bishop; *Pattern Recognition and Machine Learning*; Springer, 2006
- [65] D. Jurafsky and J.H. Martin; *Speech and Language Processing*; 3rd ed. draft
- [66] V.Nair and G.E. Hinton; *Rectified linear units improve restricted boltzmann machines*; Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807-814, 2010
- [67] A. L. Maas, A. Y. Hannun and A. Y. Ng; *Rectifier nonlinearities improve neural network acoustic models*; Proc. ICML, Vol. 30, No. 1, 2013.
- [68] S. Geman, E. Bienenstock and R. Doursat; *Neural networks and the bias/variance dilemma*; Neural Computation 4, pp. 1-58, 1992
- [69] G.M. James; *Variance and Bias for General Loss Functions*; Machine Learning 51, pp. 115-135, 2003
- [70] K. He, X. Zhang, S. Ren and J. Sun; *Deep residual learning for image recognition*; in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770-778, 2016
- [71] A. Krizhevsky, I. Sutskever, and G. E. Hinton; *Imagenet classification with deep convolutional neural networks*; in Advances in neural information processing systems, pp. 1097-1105, 2012