# Prediction of total cloud cover over Europe

Project 3 - FYS-STK4155

Paulina Tedesco and Hanna Svennevik

December 2018

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Abstract

In this article, we describe the first attempt on predicting total cloud cover fraction over the Europa based on pressure, temperature, specific- and relative humidity data measured at the ground. The data was obtained from European centre for medium range weather forecasting, ECMWF.

We use a wide range of models; Linear regression, self-implemented fully connected feed forward neural networks, neural networks from the Python libraries Keras and TensorFlow and Random forest trees in a attempt to find the most suitable model. All the codes (along with necessary environments) and data used in this article can be found in the GitHub repository (https://github.com/hannasv/Project3).

The self-implemented neural network performance worsened for a increasing number of hidden layers. The best performance with one layer was $MSE \approx 0.75$. Using one layer, all combinations of nodes converged toward this value, and we did not really notice a clear difference. The best performance using four layers was a $MSE \approx 1$, this is worse than linear regression (OLS) which had a $MSE \approx 0.9$. The reason might be that we did not find the correct numbers of nodes, but most likely it is a too simple model to describe such a complex problem. Keras performed much better than the self-implemented network conforming that increased complexity in a neural network increases performance of the prediction. Keras showed the best performance for 2 layers with an $MSE \approx 0.48$.

Random forest performed best, with an MSE of approximately 0.45. This had a larger tendency to underestimate than decision trees. Decision trees had a MSE around 0.6, a R2-score of 0.4, which tells us that the model explains roughly 40% of the variation in the data. This is actually a good result in a meteorological context.

# Chapter 2

# Introduction

## 2.1 Cloud formation

The formation of clouds is a very complex process. It depends on a wide range of physical properties. Some of the most important ones are the temperature, the stability of the atmosphere, moisture content, number concentrations of aerosols and the chemical composition of these aerosols. An aerosol in this context is a particle suspended in air.

Clouds can consist of liquid water, ice crystals or both. If the cloud has a temperature higher than $0^oC$ it is referred to as a warm cloud. These clouds can only consist of water droplets. A cold cloud has a temperature lower than $0^oC$, this can consist of ice crystals or water droplets. A water droplet at $T < 0^oC$ is called a supercooled droplet. The third category of clouds is mixed-phase clouds which consist of both ice crystals and water droplets. Ice and water are not in thermodynamic equilibrium, this means they cannot coexist in a cloud over a long time period. After a while, everything is transformed to either ice crystals or water droplets.

Aerosols are necessary for the formation of clouds since homogeneous nucleation of water droplets or ice crystals from vapour doesn't occur in nature. At a temperature lower than $-38^oC$ water droplets will spontaneously freeze, this is called homogeneous ice nucleation. Water droplets are formed on the surface of a cloud condensation nuclei (CNN). A CNN has properties which allow vapour to condense on the surface. At first, the droplet continues to grow by diffusion, when their size is sufficiently large they keep growing due to collision and coalescence with other droplets. Collisions can be caused by turbulence or a difference in fall speeds. After approximately ten thousand collisions the droplet is large enough to make it to the surface as precipitation. Ice nuclei particles (INP) are particles that provide a surface onto which molecules are likely to absorb, bond together and form aggregates with ice-like structure. When the ice embryo is formed the ice crystal will continue by either diffusion (deposition of vapour), collision and coalescence with other ice particles or supercooled droplets which freeze upon impact. The latter one produces hail. Lohmann, Lüönd, and Mahrt (2016)

### 2.1.1 The climate effect

louds have both a warming and cooling effect on climate. Low-level clouds have a tendency to cool. The dense clouds provide a white surface, which reflects the incoming solar radiation back into space. This is called the albedo effect of clouds. Solar radiation travel through the atmosphere more or less unattenuated. Aerosols affect some of the radiation by scattering it back into space. This partly offsets the greenhouse gas warming. It has been suggested to inject aerosol particles deliberately into the atmosphere to reduce the impacts of climate change. The low-level clouds are warm and the molecule concentration is high. Longwave radiation emitted by a low-level cloud will, therefore, be absorbed by other molecules and re-emitted by them at lower temperatures. The radiation is absorbed and re-emitted so many times that the presence of the cloud doesn't have an effect for the radiation which makes it back into space.

High-level clouds are thin and cold. The clouds trap heat by absorbing radiation and re-emitting at low temperatures. Molecular concentrations are low, the radiation escapes back into space. This is often referred to

as the greenhouse effect of clouds. This is shown schematically in figure 2.1. Lohmann, Lüönd, and Mahrt (2016)

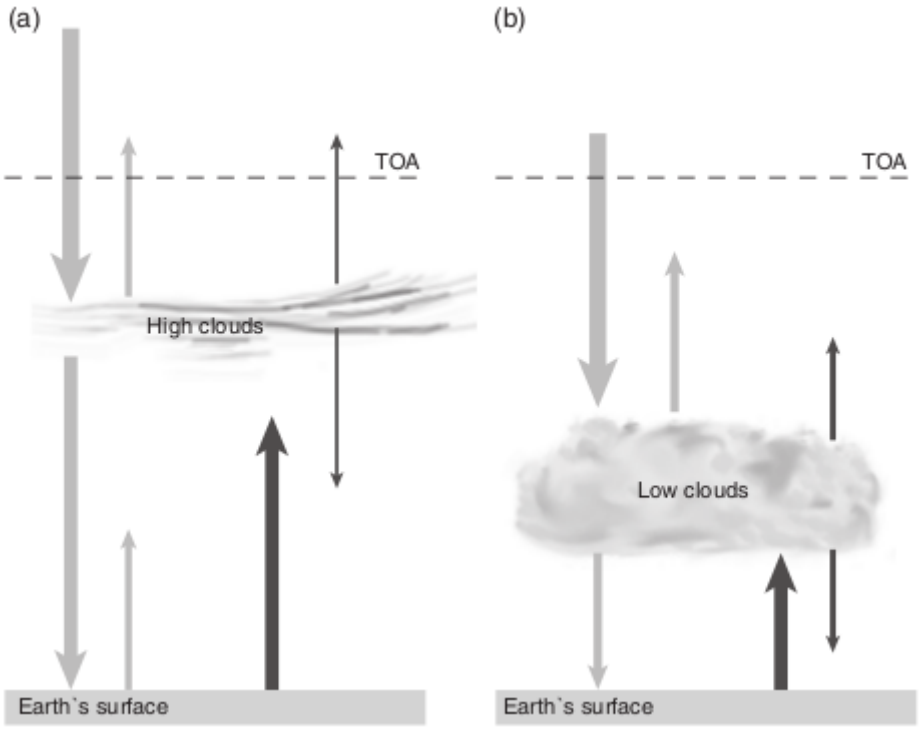


Figure 2.1: Schematic of the radiative of (a) high-level clouds and (b) low-level clouds. Short wave fluxes are illustrated by light grey and long wave fluxes are shown in dark grey. Lohmann, Lüönd, and Mahrt

In this article, we have chosen to look at the total cloud cover fraction. It is possible to download data sets for low-, medium- and high-level clouds. The idea is to first study the possibility to predict total cloud cover. When we have done this if this is possible we can start to look into predicting different levels of cloud cover.

### 2.1.2 Machine learning and cloud physics

After many decades of research, the field is still at a low level of scientific understanding. Despite the fact that computer models of Earth's climate have multiplied in number, complexity, and computational power. Cloud formation happens on a scale of 10-100m, most climate models have grids cells on the scale of 1000km. In traditional models, clouds are represented by parameterizations which get tuned, and not by the exact physical equations. Scientists at California Institute of Technology (CalTech) have started a project where they wish to use the latest breakthroughs in artificial intelligence (AI), satellite imaging, and high-resolution simulation, to build new climate model. They want to build it around Frank Giraldo's non-hydrostatic Unified Model of the Atmosphere. Its design is based on modern parallel computing. The core is so flexible it is able to solve equations to various degrees of accuracy. This should allow the earth machine to give a low-resolution overview of the planet while zooming into the clouds.

Stratocumulus clouds are a low-level dense cloud which have a cooling effect on climate. The large uncertainty associated with the extent of these stratocumulus decks can turn the global temperature up or down by a couple of degrees within this century. The current models are unable to predict which way it goes. [23]

# Chapter 3

# Methods

## 3.1 Data

In this article, we use ERA-Interim reanalysis data. This is the latest in a series of global reanalysis produced by the European Centre for Medium Range Weather Forecast, ECMWF. Climate reanalysis combines historical observations with models. One can argue that it is the closest we have to observations in multivariate, spatially complete and coherent records of meteorological data. The data is available in the period from the first of January 1979 to real time with a couple of month delay. The idea with reanalysis is to learn how to make better use of observations by analyzing the data multiple times. [5] This reanalysis is created based on models and assimilation systems. It takes archived observations as input. The final result is data sets describing the recent history of the atmosphere, land surface, and oceans. In total there are 48 parameters available for downloading. [4]

ERA-Interim uses mainly the assimilations made for ERA-40. The most notable differences are more data are subjected to bias corrections. These are described below. ERA-40 uses satellite-, radiosonde and *in situ* measurements. ECMWF used their own operational data archive but they were also supplied by other institutions where the largest contributors were the National Centre of Environmental Prediction, NCEP, and the Japan Meteorological Agency. The National Centre of Atmospheric Research, NCAR provided copies of extensive holdings of *in situ* and satellite data. Uppala et al. (2006)

In this section, we discuss some of the most important components to data assimilation systems. Observational data needs to be carefully evaluated and the errors need to be removed. An error in a sparse region will significantly reduce the quality of the data assimilation and forecasts. The coordinates of the observations need to be at the exact grid point represented by the model. This requires a 3-dimensional interpolation in order to project the data onto the model grid. The best choice in initial state has proven to be a combination of the current observations and the past observations carried forward with the model. Bengtsson (1982)

ERA-Interim reanalysis is produced using a sequential data assimilation scheme. Each cycle lasts for 12h. Information from a short-term forecast model is combined with the observation available in the current cycle. This is followed by separate variational analysis of the atmosphere and the underlying surface. These analyses are again used to initiate the next short range model forecast which is used in the next cycle. The forecast model has three fully coupled components; the atmosphere, the land surface, and the oceans When generating a forecast, the model also estimates unobserved parameters based on locally observed parameters and physical relations. This information is also carried forward in time. The accuracy of these model generated estimates are dependant on the model physics. Cloud properties are one of these features that is not directly observed but constrained by the other observations used to initiate the forecast. The archive currently contains four analysis gridded estimates of 3-dimensional meteorological variables per day. These are produced for synoptic hours 00, 06, 12 and 18 UTC. Synoptic hours is a global agreement that all meteorological observations are conducted simultaneously.[22] [5]

### 3.1.1 Area of interest

We used the scripts available in GitHub to download the data. For simplicity, we made one script per feature. After our experience, this makes for a faster download since one can send multiple data request at the same time. A too large data request is denied, so even if its possible to download all the files at the same time using one script, this will become inefficient or impossible for large data set. It also becomes easier to include new features if we keep things separate.

The spatial resolution in this article is 0.75 latitude and 0.75 longitude. The data describes the weather conditions the first week of December in 1990. We choose this week since we knew it was strong weather and clouds present.

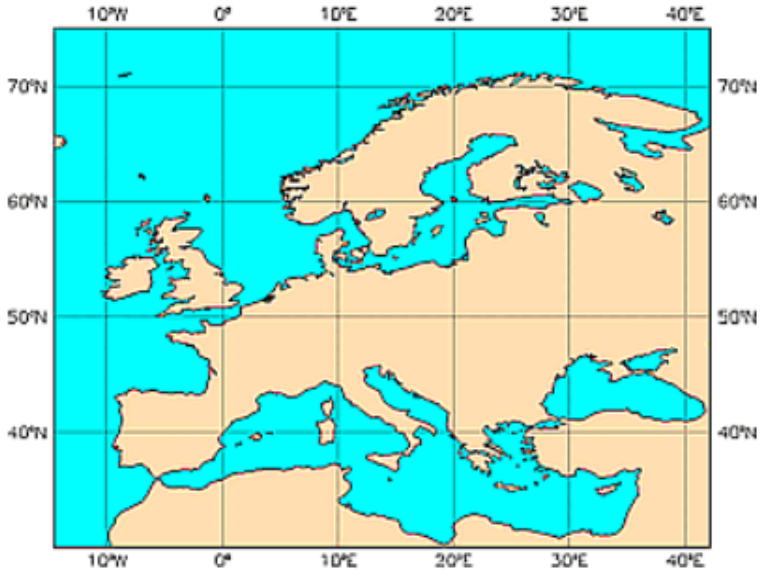


Figure 3.1: Map showing the area of interest. [2]

### 3.1.2 Features

In this article, we have chosen the features based on two criteria. It needs to be important for the physical process and to be well predicted by other climate models. We are interested in accessing the performance of our models on output from climate models. In 2.1 we discussed the importance of the presence of aerosols in cloud formation. Over land we have substantial emissions of aerosols. Which makes it a good assumption that humidity is the limiting factor. The ocean covers 71% of the surface area of the globe. Most of this is far away from emission sources and therefore air can be saturated (relative humidity is 100%) but no cloud will be formed in the absence of aerosols. Ships release aerosols when travel across oceans and ship tracks are visible in satellite images. See figure 3.2. This might make it difficult to build a model which can predict cloud cover over both land and ocean.

#### Pressure

The pressure comes in units of Pascal, Pa and in order of $10^5$. Navier-Stokes equation is known as the equation of motion. The equation is displayed in 3.1. Applying large-scale geophysical balances to Navier-Stokes equation reveals that the wind velocities are highly dependent on the pressure gradient. Thus the pressure contains information about the winds. Lohmann, Lüönd, and Mahrt (2016)

$$\frac{d\vec{u}}{dt} + f\omega \times \vec{u} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}F - \hat{k}g \tag{3.1}$$

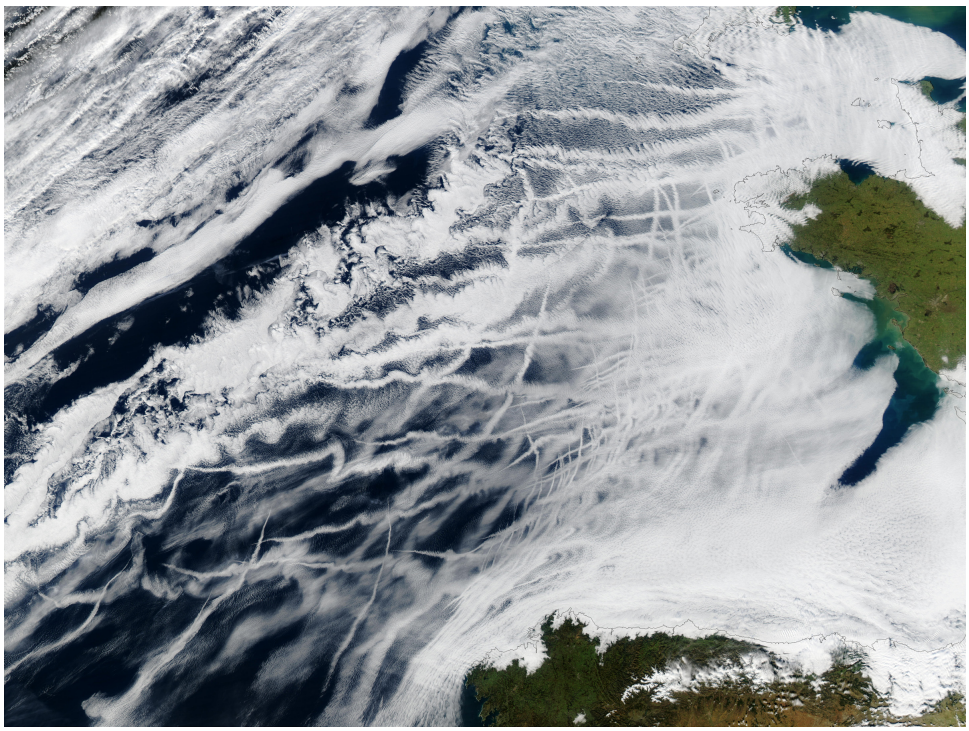

Figure 3.2: Satellite images of ship tracks of Europe's Atlantic coast. [18]

**Temperature**

The temperature is sampled at a height of 2m, this is the convention when one is conduction temperature measurements. The unit of temperature is in Kelvin, K and the scale is $10^2$. Figure 3.3 illustrates how the

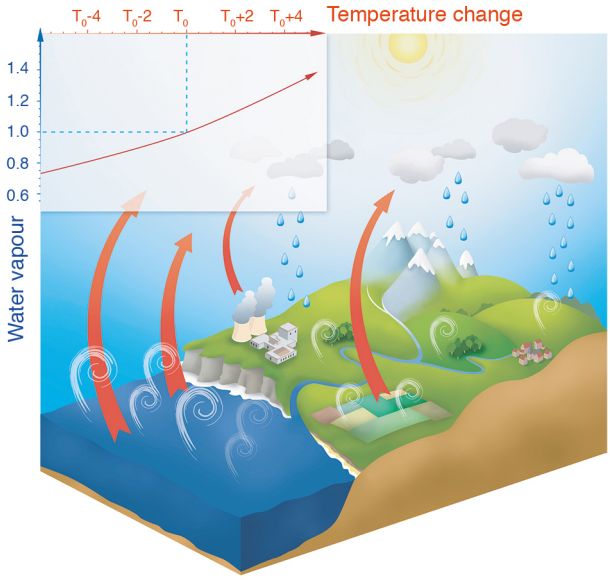


Figure 3.3: Illustration over humidity sources. [11]

clouds are formed at a height where the moist air has been cooled to the dew point temperature, $T_d$. This is the temperature at which a parcel reaches saturation.

**Relative humidity**

Both relative and specific humidity is available for six different pressure surfaces. In this article we have chosen to look at humidity's at a pressure surface 1000 hPa affects the presence of clouds. It is reasonable to assume that the moisture at the 1000 hPa resembles the moisture at the surface. Most of the humidity sources are confined close to the ground. Examples of sources are evaporation from surfaces and transpiration from vegetation.

Relative humidity is a ratio between the partial water vapour pressure, e, and the saturation water vapour pressure, $e_{s,w}$. The expression is shown in equation 3.2. This parameter gives us information about the saturation ratio for a given temperature.

$$RH = \frac{e}{e_{s,w}} * 100 \tag{3.2}$$

The unit is in percentage so its values range from 0 to 100. By assuming the latent heat of vaporisation is constant, $L_v$ one can obtain a analytical expression for saturation vapour pressure as a function of temperature by solving the Clausius-Clapeyron equation 3.3. Lohmann, Lüönd, and Mahrt 2016

$$\frac{de_{s,w}}{dT} = \frac{L_v e_{s,w}}{R_V T^2} \tag{3.3}$$

**Specific humidity**

Specific humidity, $q_v$ is unitless, but in order to be reminded that is a mass ratio one usually states that the unit is $kgkg^{-1}$. It is the ratio of the mass of water vapour, pr. mass of moist air. $q_v \in [10^{-4}, 10^{-2}]$.

$$q_v = \frac{M_v}{M_m} \tag{3.4}$$

## 3.2 Preprocessing

### 3.2.1 Feature scaling

When we are doing regression it can become problematic with features of very different scales. In this article, we have the pressure of an order hundred thousand and the specific humidity which is of the order of one hundredth.

$$x_n = \frac{x - \mu}{\sigma} \tag{3.5}$$

where $\mu$ is the mean and $\sigma$ denotes the standard deviation. This is the same expression which is used in sci-kit learns Standard Scalar. [6]

### 3.2.2 Data Transformation - Logit function

In this section log refers to the natural logarithm. To avoid issues with a small predictor space we introduce the logit function in equation 3.6. This function, f will transform values from $f : [0, 1] \to [-\infty, \infty]$. [12]

$$f(x) = log\left(\frac{x}{1-x}\right) \tag{3.6}$$

$$f^{-1}(y) = \frac{e^y}{e^y + 1} \tag{3.7}$$

## 3.3 Linear regression

### 3.3.1 Ordinary least squares

In the Ordinary Least Squares (OLS), $\hat{Y}$ denotes the predicted model given the input $\mathbf{X}$, and $\hat{\beta}_0$ is the intercept also known as the bias in machine learning. The best fit is given by the $\hat{\beta}$-values that minimise the residual sum of squares, which is the cost function in this case. The minimum is found by differentiating the RSS with regard to $\beta$ and setting it equal to zero. Finally, solving for $\hat{\beta}$ results in equation 3.10. This has a unique solution when $\left(\mathbf{X}^T\mathbf{X}\right)$ is non-singular. A matrix is non-singular when $X^TX$ has full rank, then all the columns are linearly independent and the matrix is invertible Hastie, Tibshirani, and Friedman (2001) [8].

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j \tag{3.8}$$

$$RSS(\hat{\beta}) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) \tag{3.9}$$

$$\hat{\beta} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T\mathbf{y} \tag{3.10}$$

### 3.3.2 Ridge regression

When the amount of predictors is large relative to the number of observations, a penalization term is often added to penalize those features that add little to the prediction, by adding a penalty term, $\lambda$, to the diagonal of the matrix. The ridge regression imposes a size constraint on the coefficients and reduces the number of correlated coefficients which could result in a singular matrix. Hastie, Tibshirani, and Friedman (2001)

$$RSS(\hat{\beta}) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta \tag{3.11}$$

$$\hat{\beta}^{ridge} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1} \mathbf{X}^T\mathbf{y} \tag{3.12}$$

### 3.3.3 The LASSO regression

The difference between the LASSO and ridge regression is that the regularization term in the cost function is in absolute value. The $\sum_{j=0}^{p} \beta_j^2$ is replaced with $\sum_{j=0}^{p} |\beta_j|$. Lasso is non-linear and there is no closed expression for $\hat{\beta}$ in this case. [8] The expression for Lasso on Lagrangian form is:

$$\hat{\beta}^{lasso} = \underset{\beta}{argmin} \left\{ \frac{1}{2} \sum_{i=1} \left( y_i + \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=0}^{p} |\beta_j| \right\} \tag{3.13}$$

Another substantial difference between the ridge and the LASSO is that the former shrinks the coefficients, but not to zero, whereas the latter does not only penalize the coefficients but actually set them to zero if they are irrelevant. Hastie, Tibshirani, and Friedman (2001)

### 3.3.4 Performance metrics

In order to quantify the performance of a regression model we use metrics. In this article, we use both the mean square error, MSE, and the $R^2$-score. MSE is the average squared distance between the predicted and the true

value. The $R^2$-score is a measure on how well future samples are likely to be predicted by the model. They both result in a good score when the models' performance approaches the true value. Then the MSE goes to zero and the $R^2$ goes to one. The metrics can be calculated by the following equations. [9]

$$MSE(\hat{y}, \hat{\tilde{y}}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 \tag{3.14}$$

$$R^2(\hat{y}, \hat{\tilde{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2} \tag{3.15}$$

where mean value of $\hat{y}$ is defined as $\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i$. In this article, we have normalized the response in order to get comparable results, when using both logit transformation and normal data. This is necessary since MSE depends on the scale of the predictor we can't compare our results after using different transformations of the data. R2 is normalized, so this gives already comparable results.

## 3.4 Neural networks

The objective function in this regression problem is minimizing the squared distance between the data and the fit. The latter is also known as the residual sum of squares, RSS. This objective function allows for the addition of a penalty, $\lambda$. The purpose of the penalty is to get a good fit while the weights are retained to low values.

### 3.4.1 The perceptron rule

Neurons are interconnected nerve cells in the brain that transmit chemical and electrical signals. The first model of a simplified brain cell was published in 1943 by Warren McCulloch and Walter Pitts[16]. They described the neurons as a logic gate with a binary output; the signals reach the dendrites, and are transmitted to the axon; only if the signals exceeds a certain threshold, it will generate and pass an output by the axon. See the model of a neuron in the following figure 3.4.

A perceptron rule based on the previous model was published in 1957 by Frank Rosenblatt[16]. The algorithm

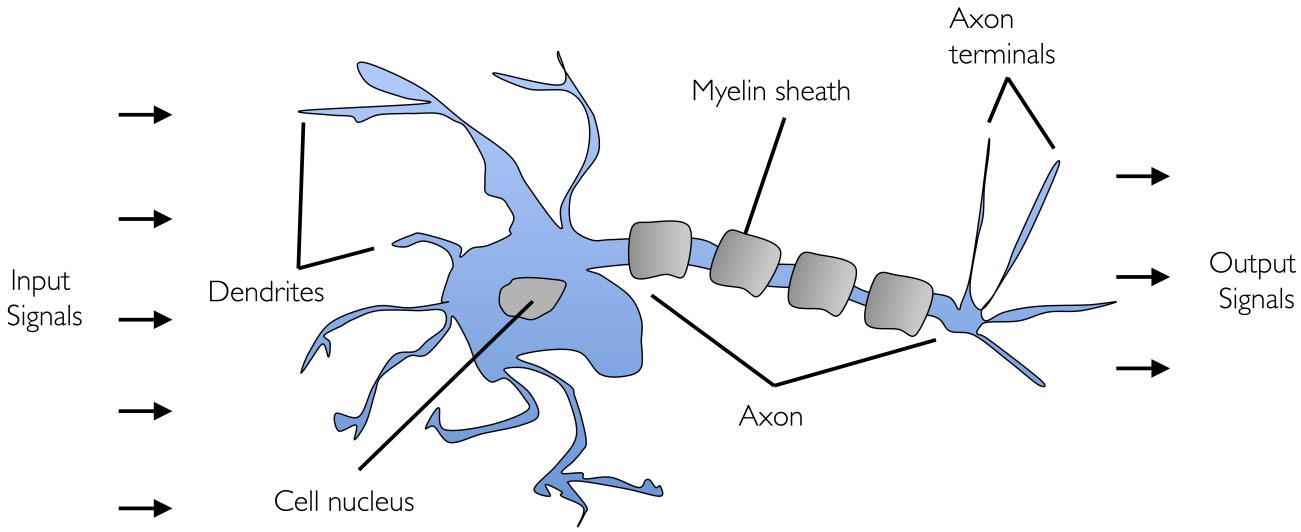


Figure 3.4: Model of a neuron [16].

he proposed learns automatically the optimal weight coefficients involved in the decision of whether the neuron fires or not. This algorithm can be used to predict to which class the sample belongs. A schematic representation of the perceptron for three inputs $x_1, x_2, x_3$ is shown in figure [15], but it can have more or fewer outputs.

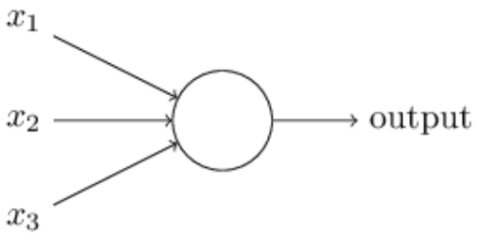


Figure 3.5: Perceptron model [15].

More formally, this model can be written as follows:

$$y = f(\sum_{i=1}^{n} w_i x_i) = f(u) \tag{3.16}$$

where the output, $y$, is the value of the activation function applied to the weighted sum of the signals $x_i, i = 1 : n$ received from other neurons, which can be written as, $z = w_1 x_1 + w_2 x_2 + ... + w_n x_n$. [10] If the net input of a sample $x^{(i)}$ is greater than the defined threshold $\theta$ , we predict class 1, and class 0 otherwise:

$$y = \begin{cases} 0, & \text{if } \sum_{i=1}^{n} w_i x_i \leq \theta \\ 1, & \text{if } \sum_{i=1}^{n} w_i x_i > \theta \end{cases}$$

The system above can be simplified by writing the sum as a dot product of vectors $W.X$, and moving the threshold to the other side of the inequality (known as the bias, b).

$$y = \begin{cases} 0, & \text{if } X.W + b \leq 0 \\ 1, & \text{if } X.W + b > 0 \end{cases}$$

Figure 3.6 shows, in the left panel, the binary output, 1 or 0, after applying the activation function (represented in the figure with the letter $\phi$) to the weighted sum of signals; the right panel shows how the model can be used to discriminate between two linearly separable classes.

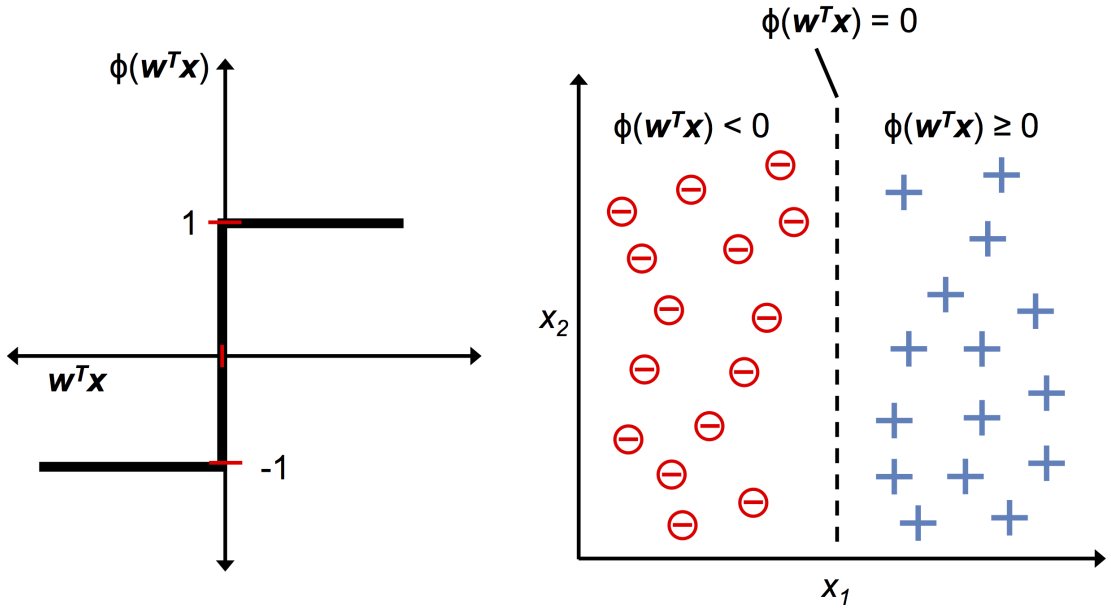


Figure 3.6: Representation of the binary output after applying the decision function (left panel), and how it is used to discriminate classes (right panel) [16].

The output value is the class label predicted by the step function defined above. In the algorithm, the simultaneous update of the weights can be written as:

$$w_i := w_i + \Delta w_i \tag{3.17}$$

And $\Delta w_i$ is calculated by the perceptron learning rule:

$$\Delta w_i = \eta(y^{(j)} - \hat{y}^{(j)})x_j^{(i)} \tag{3.18}$$

Here, $\eta$ is the learning rate, $y^{(j)}$ is the true class label of the training sample $j$, and $\hat{y}^{(j)}x_j^{(i)}$ is the predicted class label.

The general idea of the models is to mimic how the neurons in the brain behave when it receives many signals from the neurons it is connected to, this is, it fires or not. The convergence of the perceptron is only guaranteed if the classes are linearly separable and the learning rate is small enough. The following diagram summarises the perceptron model. It illustrates how the net input is calculated from the combination of the inputs of the sample $x$ with the weights $w$. The net input is passed to the threshold function in order to generate the binary output 0 or 1. Then, the prediction error is calculated from the output and the weights are updated during the learning phase.

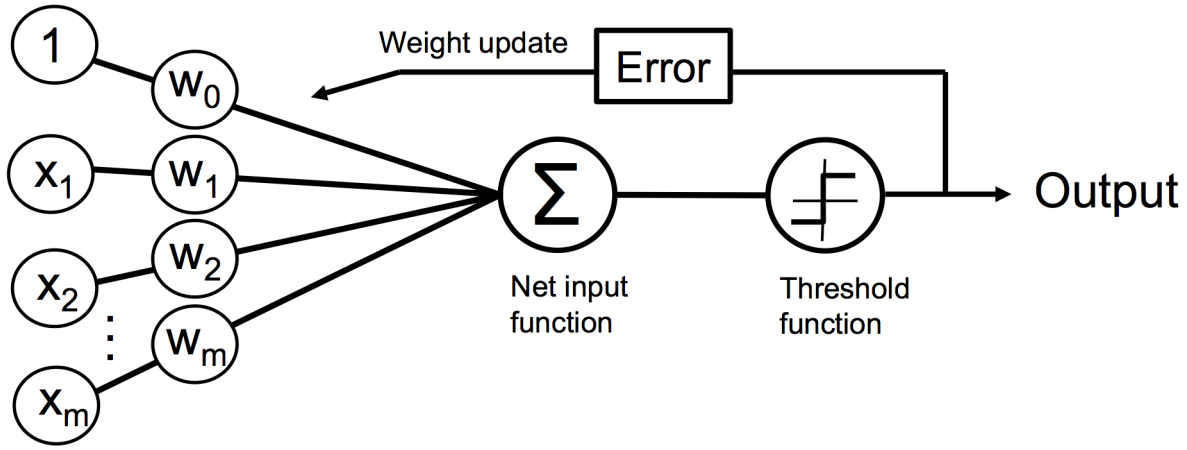


Figure 3.7: Diagram of the perceptron model [16]. The threshold function receives the net input, and generates a binary output. Then the error is calculated and the weights are updated.

## 3.5 Tree-Based Methods

Tree-based methods are conceptually simple and interpretable although powerful. The goal of these methods is to predict the value of a target variable based on based on several other input variables. They partition the feature space, $\{X_1, X_2, ..., X_p\}$ into a set of rectangles before fitting a simple model to each of them. Following the usual nomenclature, the observations of an item would be the branches, and the conclusions of the target would be the leaves. If the target variable takes discrete values, it is called a classification tree, where the leaves represent the class labels. In this study, we are going to employ regression trees since our response, $Y$, here the total cloud cover, is continuous. The data consists of $p$ inputs and a response for each of the $N$ observations, this is, $(x_i, y_i)$ for $i = 1, 2, ..., N$, with $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$.

A decision tree can be learned by splitting the input space into subsets based on an attribute value test. Although each partitioning line has a simple description, some of the resulting regions are complicated to describe. Therefore, we simplify the problem by focusing on binary recursive partitions. This consists of first splitting the space into two region regions and then model the response. In the next step, one or both sub-regions are split again, and the process is repeated until some stopping rule is applied, e.g. when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. Suppose that the first partition leads to $M$ regions $R_1, R_2, ..., R_M$ and that the response is modelled as a constant $c_m$ in

each region Hastie, Tibshirani, and Friedman (2001) [8]:

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m) \tag{3.19}$$

Adopting the sum of squares as minimization criterion, the best $\hat{c}_m$ is the average of $y_i$ in region $R_m$:

$$\hat{c}_m = ave(y_i | x_i \in R_m) \tag{3.20}$$

However, in most cases, it is computationally infeasible to find the best binary partition by applying the sum of squares, hence a greedy algorithm is preferred. In this kind of algorithm, the local optimum is chosen in each stage with the intention of finding the global optimum. The algorithm consists with splitting the variable $j$ at the point $s$, defining the half-planes:

$$R_1(j,s) = \{X | X_j \leq s\} \quad and \quad R_2(j,s) = \{X | X_j > s\} \tag{3.21}$$

Then, the following expression is solved for $j$ and $s$:

$$\min_{j,s}[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2] \tag{3.22}$$

For any choice of $j$ and $s$, the inner minimization is solved by

$$\hat{c}_1 = ave(y_i | x_i R_1(j,s) \quad and \hat{c}_2 = ave(y_i | x_i R_2(j,s) \tag{3.23}$$

The best split can be found from the equations above. After the partition, the process is repeated on each of the resulting regions.

An important tuning parameter is the tree size since a large tree size might overfit the data, whereas a small tree might not capture the important structure. The optimal size should be adaptively chosen from the data. One option would be to split the tree nodes only if the sum-of-squares exceeds some threshold, but this could oversee a potentially good split in the next step. Another popular strategy is the cost-complexity-pruning that allows a tree to grow large and stops the pruning only when some minimum node is reached.

Some techniques, often called ensemble methods, construct more than one decision tree, for instance, boosted trees or bootstrap aggregation. Random forest, used for in the analysis of our data, is a specific type of bootstrap aggregation, which consists of building multiple decision trees by repeatedly resampling training data with replacement, and voting the trees for a consensus prediction.

### 3.5.1 Sigmoid and other activation functions

The concept of sigmoid neurons is similar to the perceptron model represented in figure 3.5, and it is introduced to make small changes in bias and weights cause only small changes in the output (this is not always the case of the perceptron model). The difference between the perceptron and the sigmoid neurons is that for the latter the output is not 0 or 1 but $\sigma(W.X + b)$, where $\sigma$ is the sigmoid function defined by[15]:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{3.24}$$

The shape of this function is a smoothed out version of the step function (see figure 3.8). As a consequence, small changes in the output for small changes in the weights and bias. It is the shape of the function, and not its exact form, which is important. Thus, there are several activation functions that can be applied in neuronal networks (see figure 3.8). However, the algebra is simpler when the sigmoid function is used because of the properties of the exponential, and it is widely used in the literature.

The layers of deep neuronal networks may learn at different speeds. It might happen that the gradients get smaller for each iteration and that the training process does not converge to the solution (known as the vanishing gradients problem). But the opposite problem is also possible, that the gradients grow, and hence the updates of the weights are larger in each iteration (known as the exploding gradients problem). It has been shown by Glorot and Bengio (2010) that other activation functions than sigmoid behave much better in neuronal networks (see figure 3.8).

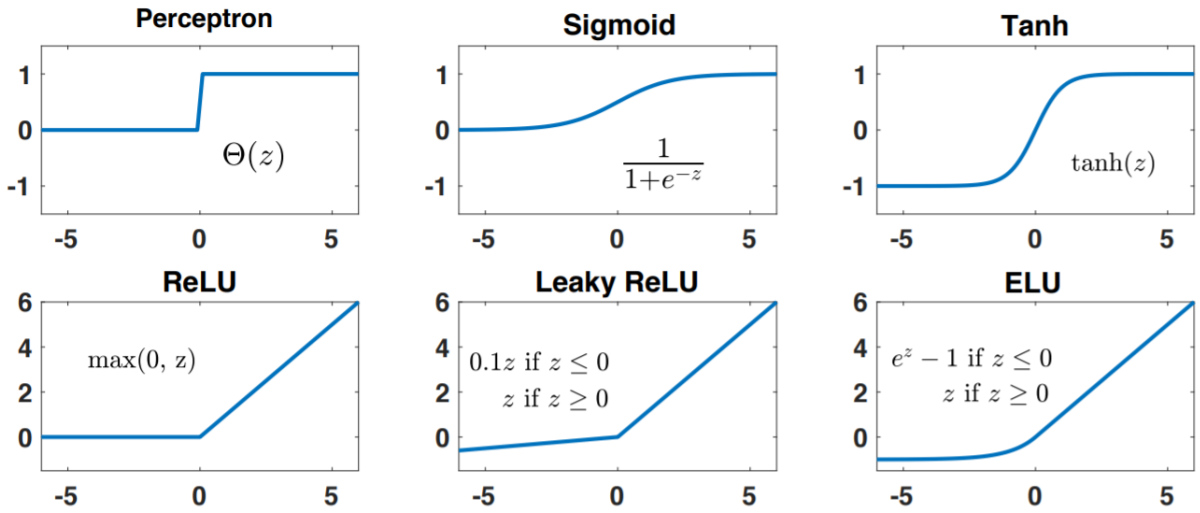


Figure 3.8: Different activation functions and their formulas [14].

### 3.5.2 Architecture

A neural network consisting of at least three layers, the input layer, the output layer, and the hidden layer in the middle is called a Multilayer Perceptron. The example of figure 3.9 has two hidden layers, the hidden layers consist of six neurons, and there is one single output.

In a feedforward neural network, the output from one layer is the input to the next one, so the information is always sent forward [15]. It is possible to have loops in other models, such as recurrent neural networks, but they have been less influential than feedforward neuronal networks.

We can also distinguish between hard and soft classifiers. A binary model, which only needs one neuron in the output layer, would be an example of a hard classifier because it outputs the class of the input directly. A classifier that outputs the probability of the class is a soft classifier. An example of this is the sigmoid function [10].

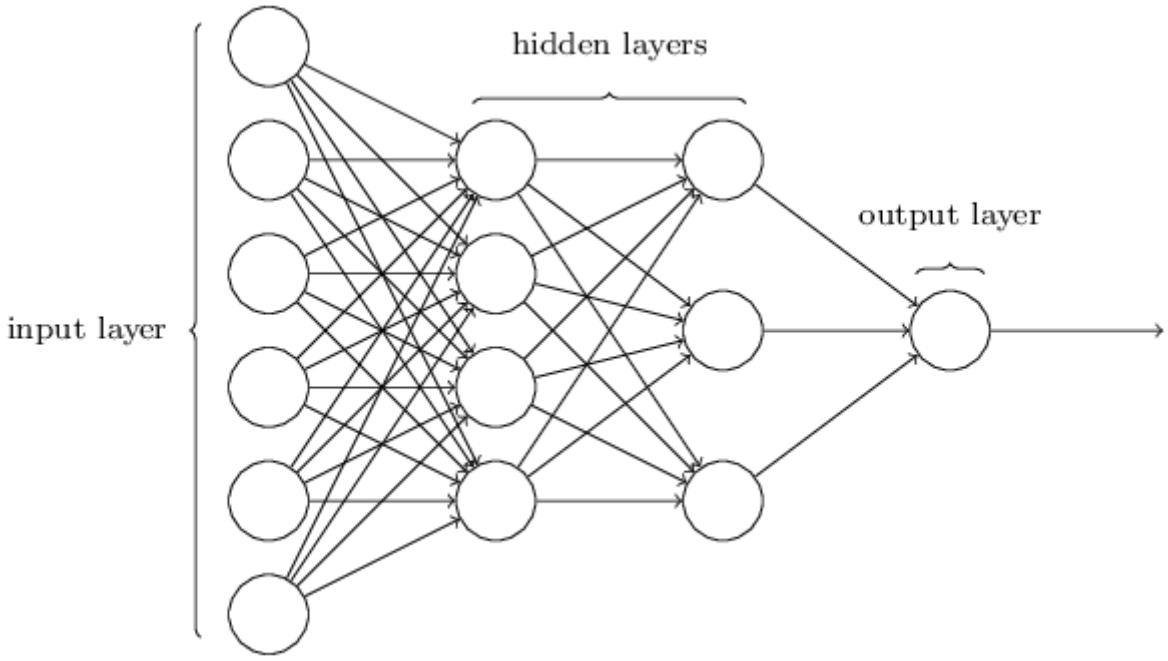


Figure 3.9: Illustration of a feedforward neural network with two hidden layers [15].

### 3.5.3 Forward and Backpropagation algorithm

Until this point, we have been looking at the forward propagation. It consists of calculating the weighted sum of the input features to each neuron in the hidden layer, then pass it through the activation function, and calculate the weighted sum of the activations in the hidden layer to each neuron in the output layer, to finally calculate the output. [10]

In the backpropagation algorithm, we study how altering the weights and biases affect the cost. The following calculations are based on two assumptions. The first one being that the cost function can be written as the average of the cost functions for all the individual training examples. The second one is that we assume it can be written as a function of the output from the neural network. These are both true for the cost function used in this article which is the sum of squares.

Let $L$ denote the output layer, $l$ the index of an arbitrary layer, $\delta^l$ be the error in layer $l$, $C$ be the cost function and b denote bias and w denote weights, $\sigma$ denote an arbitrary activation function.

$$\delta^L = \frac{\partial C}{\partial a_j^L}\sigma'\left(z_j^L\right) = \left(a_j^L - y\right)\sigma'\left(z_j^L\right) \tag{3.25}$$

$\frac{\partial C}{\partial a_j^L}$ describes how fast the cost changes as a function of the $j^{th}$ activation function. While $\sigma'\left(z_j^L\right)$ describes the change in the activation function at the output layer. In regression problems the activation in the output layer is always linear, $\sigma'\left(z_j^L\right) = 1$. The product of these terms expresses the error in the output layer.

$$\delta^l = \left(\left(w^L\right)^T \cdot \delta^L\right)\sigma'\left(z_j^l\right) \tag{3.26}$$

$\left(\left(w^L\right)^T \cdot \delta^L\right)$ backpropagates the error in the output layer, and $\sigma'\left(z_j^l\right)$ describes the change in the activation between these to layers. The product of the two terms describes the error in the hidden layer. This can be written more generally with $l+1$ instead of $L$.

$$\frac{\partial C}{\partial b_{jk}^l} = \delta^l \tag{3.27}$$

Equation 3.27 describes that the rate of change of the cost as a function of the bias is the error of a layer.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta^l \tag{3.28}$$

The rate of changes to the cost with respect to the weights of a certain layer, $l$, is the product of the activation in layer $l-1$ and the error in the layer $l$. The backpropagation algorithm was introduced in the late 1980s and has then been a central part of the learning process of neural networks. Nielsen (2015) [15].

## 3.6 Structure and implementations

Since we have gridded data it needs to be reshaped before we can use it in our neural network. Starting at the north-west corner, $(x_0, y_0)$ on the map shown in figure 3.1. We transpose each row and append it to the data, X.

The more detailed description of this is as follows. Let the power $(x_n, y_m)$ denote the coordinates of the (n,m)-th grid cell and let the index $t_j$ denote the jth timestamp. rh denotes relative humidity, qv denotes specific humidity, p stands for pressure and T is temperature. Let's assume the grid has the shape [N+1, M+1] so the last grid cell has coordinates $(x_N, y_M)$. Let there be k timestamps.

$$\mathbf{X} = \begin{pmatrix} rh^{t_0}_{(x_0,y_0)} & qv^{t_0}_{(x_0,y_0)} & p^{t_0}_{(x_0,y_0)} & T^{t_0}_{(x_0,y_0)} \\ rh^{t_0}_{(x_1,y_0)} & qv^{t_0}_{(x_1,y_0)} & p^{t_0}_{(x_1,y_0)} & T^{t_0}_{(x_1,y_0)} \\ rh^{t_0}_{(x_2,y_0)} & qv^{t_0}_{(x_2,y_0)} & p^{t_0}_{(x_2,y_0)} & T^{t_0}_{(x_2,y_0)} \\ \vdots & \vdots & \vdots & \vdots \\ rh^{t_0}_{(x_N,y_M)} & qv^{t_0}_{(x_N,y_M)} & p^{t_0}_{(x_N,y_M)} & T^{t_0}_{(x_N,y_M)} \\ rh^{t_1}_{(x_0,y_0)} & qv^{t_1}_{(x_0,y_0)} & p^{t_1}_{(x_0,y_0)} & T^{t_1}_{(x_0,y_0)} \\ rh^{t_1}_{(x_1,y_0)} & qv^{t_1}_{(x_1,y_0)} & p^{t_1}_{(x_1,y_0)} & T^{t_1}_{(x_1,y_0)} \\ rh^{t_1}_{(x_2,y_0)} & qv^{t_1}_{(x_2,y_0)} & p^{t_1}_{(x_2,y_0)} & T^{t_1}_{(x_2,y_0)} \\ \vdots & \vdots & \vdots & \vdots \\ rh^{t_k}_{(x_N,y_M)} & qv^{t_k}_{(x_N,y_M)} & p^{t_k}_{(x_N,y_M)} & T^{t_k}_{(x_N,y_M)} \end{pmatrix} \tag{3.29}$$

All the files are available in the GitHub repository. All the model are build and the result are plotted in the notebooks.

The structure of our self-implemented neural network is inspired by the structures found in the book by Raschka (2017). We have attempted to use both sigmoid and elu activation functions, but most of the focus has been on sigmoid since elu results in NaN for more than one hidden layer. We have kept the same batch size (= 10), optimizer (stochastic gradient descent) and initialization of the weights and bias, $w \in (-0.1, 0.1)$ and $b = 1$ thought the experiments. The main focus has been on tuning the learning rate, model depth and the number of nodes in each layer.

### 3.6.1 Implementation of stochastic gradient descent with minibatches in neural network

Code Listing 3.1: Python pseudo-code of Stochastic Gradient Descent with Minibatches

```
def _minibatch_sgd(self, X_train, y_train):
    """
    Performes the stochastic gradient descent with mini-batches for one epoch.

    X_train : array, shape = [n_samples, n_features]
        Input layer with original features.
    y_train : array, shape = [n_samples]
        Target class labels or data we want to fit.

    """
    n_samples, n_features = np.shape(X_train)

    indices = np.arange(n_samples)

    if self.shuffle:
        self.random.shuffle(indices)

    for idx in range(0, n_samples, self.batch_size):

        batch_idx = indices[idx:idx + self.batch_size]

        # Forwardpropagation.
        Z_hidden, A_hidden, Z_out, A_out = self._forwardprop(
            X_train[batch_idx, :]
        )

        # Backpropagation.
        self._backprop(
            y_train, X_train, A_hidden, Z_hidden, A_out, Z_out, batch_idx
```

```
            )

        return self
```

Employing a batch size larger than one has two advantages, it reduces the variance in the parameter update leading to more stable convergence, and it allows us to use highly optimized matrix operations in the computation of the cost and gradient. [19]. Bearing in mind that the matrix operations are more efficient when the matrix size is a power of 2, it is recommended to use a batch size which also is a power of 2. Another difference in the implementation for this variant is that we randomize the training sample before applying the method.

### 3.6.2 Implementation of Neural Networks

The perceptron model can be summarised by the following steps[16]:

1. Initialise the weights, w, to zero or small random numbers.
2. For each training sample $x^{(j)}$:
   (a) Compute the output value $\hat{y}$.
   (b) Update the weights.

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias weights $\hat{b}$ are often initialised to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

In this article we have improved the flexibility in our neural net MLP to include a arbitrary number of layers, with a arbitrary number of nodes in each layer.

Code Listing 3.2: Python implementation of forward and backward prop in neuronal networks

```
def _forwardprop(self, X):
    """Compute forward propagation step

    X : array, shape = [n_samples, n_features]
        Input layer with original features.
    """
    A_hidden = []
    Z_hidden = []

    for i in range(self.n_hidden_layers):
        if i == 0:
            z_temp =  np.dot(X, self.W_h[i]) + self.b_h[i]
            #print(z_temp)
        else:
            z_temp = np.dot(a_temp, self.W_h[i]) + self.b_h[i]
            #print(z_temp)

        Z_hidden.append(z_temp)
        a_temp = self.activate(z_temp, self.activation, deriv = False)
        A_hidden.append(a_temp)

    Z_out = np.dot(a_temp, self.W_out) + self.b_out
    # Linear activation in the output layer regression problems
    A_out = Z_out

    return Z_hidden, A_hidden, Z_out, A_out
```

```
def _backprop(self, y_train, X_train, A_hidden, Z_hidden, A_out, Z_out, batch_idx):
    """ Backpropagation algorithmn for MLP with a arbitrary number of hidden nodes and

    X_train : array, shape = [n_samples, n_features]
        Input layer with original features.
    y_train : array, shape = [n_samples]
        Target class labels or data we want to fit.
    Z_hidden : (array-like) shape = []
      Signal into the hidden layer.
    A_hidden : (array-like) shape = []
      The activated signal into the hidden layer.
    Z_out :
        Signal into the output layer.
    A_out :
        Activated signal function.
    batch_idx : int
        The index where you iterate from.
    """

    # This is the derivative assuming our cost function is 0.5*two_norm(A_out - y)**2

    error_out = A_out - y_train[batch_idx].reshape(len(y_train[batch_idx]), 1)

    # Since we are in the regression case with a linear ouput funct.
    act_derivative_out = 1

    delta_out = error_out*act_derivative_out

    grad_w_out = np.dot(A_hidden[-1].T, delta_out)
    grad_b_out = np.sum(delta_out, axis=0)

    # Updating the output weights
    self.W_out = self.W_out - self.eta * grad_w_out
    self.b_out = self.b_out - self.eta * grad_b_out


    # Looping over all the hidden layers except one
    # If the layer only have one layer it doesn't go into this while loop

    i = 0
    while (i < self.n_hidden_layers-1):
        # Index moving backward in the layers.
        layer_ind = self.n_hidden_layers - 1 - i
        act_derivative_h = self.activate(Z_hidden[layer_ind], self.activation, deriv=T

        if (i == 0):
            error_prev = np.dot(delta_out, self.W_out.T) * act_derivative_h
        else:
            error_prev = np.dot(error_prev, self.W_h[layer_ind+1].T) * act_derivative_h

        grad_w_h = np.dot(A_hidden[layer_ind - 1].T, error_prev)
        grad_b_h = np.sum(error_prev, axis=0)

        self.W_h[layer_ind] = self.W_h[layer_ind] - self.eta * grad_w_h
        self.b_h[layer_ind] = self.b_h[layer_ind] - self.eta * grad_b_h
        i += 1

    act_derivative_h = self.activate(Z_hidden[0], self.activation, deriv=True)

    # Case with one hidden layer doesn't enter the while loop.
```

```
        if( self.n_hidden_layers == 1):
            error_last = np.dot(delta_out, self.W_out.T) * act_derivative_h
        else:
            error_last = np.dot(error_prev, self.W_h[layer_ind].T) * act_derivative_h

        grad_w_h = np.dot(X_train[batch_idx].T, error_last)
        grad_b_h = np.sum(error_last, axis = 0)

        self.W_h[0] = self.W_h[0] - self.eta * grad_w_h
        self.b_h[0] = self.b_h[0] - self.eta * grad_b_h

        return None
```

For more details about methods and implementation of regression analysis, see the report and the code of project 3 in the GitHub repository.

### 3.6.3 Keras and Tensorflow

The python libraries provide easily readable code and are very powerful. We will use TenosorFlow and Keras (which is built inside tensorflow).

The basic computation unit in TensorFlow is a graph. A TensorFlow project is typically structured into 2 parts:a construction phase where you design the computational graph, and an analysis phase where you run the graph and perform calculations on it. The packages and versions used for the generation of the code are listed in the file tflow.txt in the folder envs in the GitHub. The following lines show the setup for the notebook files tensorflow_nn.ipynb and tensorflow_nn_different_results.ipynb (the last one provides different results for several combinations of layers and nodes, and activation functions).

Code Listing 3.3: Setup of TensorFlow

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
import tensorflow as tf

import matplotlib.pyplot as plt
import netCDF4 as n
%matplotlib inline
from utils import train_test_split

from tflow_reg import nn_model
```

The function nn_model from tflow_reg.py is self-implemented for this analysis, and is based on the TensorFlow module. The following lines define our model.

Code Listing 3.4: Implementation of a neural Network with TensorFlow

```
def nn_model(X, input_size, hidden_sizes, output_size, stddev=None):

    """
    :param X: features
    :param input_size: number of features
    :param hidden_sizes: number of nodes of the hidden layers, array-like
    :param output_size: size of the output: 1
    :param stddev: standard deviation for better elu and relu fits
    :return:
    """
```

```
    # Define weights and biases
    # (Using normal distribution)

    # 1) set up parameters
    w = []
    b = []
    layer = []

    # 2) APPEND ALL THE LAYERS

    # Input layer
    w.append(tf.Variable(tf.random.normal([input_size, hidden_sizes[0]],
                                          mean=0.0,
                                          stddev=1.0,
                                          dtype=tf.float32,
                                          seed=None,
                                          name=None
                                          )))

    b.append(tf.Variable(tf.zeros(hidden_sizes[0])))

    # add hidden layers (variable number)
    for i in range(1, len(hidden_sizes)):
        w.append(tf.Variable(tf.random_normal([hidden_sizes[i - 1],
        hidden_sizes[i]], stddev=1.0)))
        b.append(tf.Variable(tf.zeros([hidden_sizes[i]])))

    # Output layer
    w.append(tf.Variable(tf.random.normal([hidden_sizes[-1], output_size],
                                          mean=0.0,
                                          stddev=1.0,
                                          dtype=tf.float32,
                                          seed=None,
                                          name=None
                                          )))

    b.append(tf.Variable(tf.ones(output_size)))

    # 3) DEFINE MODEL

    layer.append(tf.nn.sigmoid(tf.matmul(X, w[0]) + b[0]))

    for i in range(1, len(hidden_sizes)):
        layer.append(tf.nn.sigmoid(tf.matmul(layer[i - 1], w[i]) + b[i]))

    output = tf.matmul(layer[-1], w[-1]) + b[-1]

    return output
```

After defining our model, train the neural network by iterating it through each sample in dataset, itarating over the data i times.

Code Listing 3.5: Setup of TensorFlow

```
# Initiate session and initialize all vaiables
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver = tf.train.Saver()
    c_t = []
    c_test = []
    r2_t = []
    r2_test = []
```

```
for i in range(10000):
    sess.run([cost,train],feed_dict=     {xs:Xn_train, ys:yn_train})
    c_t.append(sess.run(cost, feed_dict={xs:Xn_train,ys:yn_train}))
    c_test.append(sess.run(cost, feed_dict={xs:Xn_test,ys:yn_test}))
    print('Epoch :',i,'Cost train:',c_t[-1], 'Cost test:',c_test[-1])
    r2_t.append(sess.run(R_squared, feed_dict={xs:Xn_train,ys:yn_train}))
    r2_test.append(sess.run(R_squared, feed_dict={xs:Xn_test,ys:yn_test}))
    print('Epoch :',i,'R2 train:',r2_t[-1], 'R2 test:',r2_test[-1])

pred = sess.run(output, feed_dict={xs:Xn_test})

# Plot the data
```

tf.Session() initiate current session, then we use sess.run() to the run elements in the graph. tf.global$_v ariables_i nitializer()u$
$xs : X_train[j,:], ys : y_train[j])willacutallyrunthecostandtrainstep.Aftertrainingthemodel, wecantestitbyrunningpred =$
$sess.run(output, feed_dict = xs : X_test)onthetestdata.$

Although the functions nn_model learns from the data and the cost converges, the R-squared (also self-implemented because it is not available on the TensorFlow module) increases very slowly and does not reach positive value within the number of iterations chosen. This is a very flexible library, but it is very costly computationally to run, so we will further test it in the future. Each run takes several hours with our system, and the memory does not allow us to use more than 3 days of data. In this sense, Keras resulted in more convenient for our work. For this reason, the results using model_nn.py are left out of the report (but they can be found in the notebooks).

We compare the results obtained with our self-implemented code with the results accomplished using tf.keras, TensorFlow's high-level Python API for building and training deep learning models. Some of the benefits of using this API (stated in Keras' website, but also proved during this study) are that it is [20]:

- "User friendly Keras has a simple, consistent interface optimized for common use cases. It provides clear and actionable feedback for user errors.
- Modular and composable Keras models are made by connecting configurable building blocks together, with few restrictions.
- Easy to extend Write custom building blocks to express new ideas for research. Create new layers, loss functions, and develop state-of-the-art models."

To use this library, the program must be set up by:

Code Listing 3.6: Setup of keras

```
!pip install -q pyyaml  # pyyaml is an optional
import tensorflow as tf
from tensorflow.keras import layers

print(tf.VERSION)
print(tf.keras.__version__)
```

Then we build the network by stacking the layers with keras.Sequential. There are different activation functions, such as sigmoid, elu, relu, and optimizers, such as SGD, RMSProp, and Adam, that we tested. The configuration of the learning process is completed by calling the compile method, see the example:

Code Listing 3.7: Example of learning process with keras

```
model = tf.keras.Sequential([
# Adds a densely-connected layer with 64 units to the model:
layers.Dense(64, activation='relu'),
# Add another:
```

```
    layers.Dense(64, activation='relu')])

model.compile(optimizer=tf.train.AdamOptimizer(0.001),
              loss='mse',
              metrics=['mse'])
```

Although we set the number of epochs to 1000, we use a use a callback that tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then automatically stop the training. Unfortunately, the R-squared is not implemented as one of the metrics, but the MSE is available. Then we can use tf.keras.Model.fit, and later plot the results. All the code is available in the notebook "keras_nn.ipynb".

## 3.7 Choosing the network architecture

Since no one that we are aware of has attempted to use machine learning to predict cloud cover fractions we have no literature describing architectures that have been used before. This leaves us with the brute force approach. We start by building a one layer model with nodes in the interval {10,30,50,100,200,300,400,500}. For two, three and four layer models we choose the number of nodes in the interval {100,125,150,175,200}. We have created plots based on different amounts of data and different combinations and permutations of the number of nodes. In this article we will use the shape of the curve, performance metric to evaluate if our self implemented neural network. A smooth curved shape of a curve shows that you've used the correct learning rate. This shape is illustrated by the red line in figure 3.10.

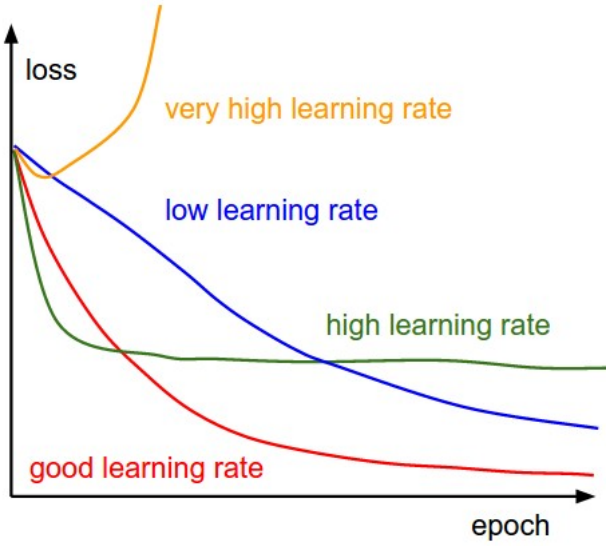


Figure 3.10: Illustrates the relation between the correct learning rate and the shape of the curve. [21]

### 3.7.1 Tree-based methods

We used the modules RandomForestRegressor and tree.DecisionTreeRegressor from the scikit-learn python library. Unfortunately, manual pruning is not implemented in sklearn [17]. However, we can limit the depth of a tree using the max-depth parameter. We tune the parameters max_depth, max_leaf_nodes, and n_estimators. The file of the environment used is called trees.txt and can be found in the GitHub.

# Chapter 4

# Results

## 4.1 Inspection of the Data

We predict the total cloud cover (tcc) by employing the factors relative humidity (rh) at 1000 hPa, specific humidity (q) at 1000 hPa, sea level pressure (sp) and temperature measured 2m meters over the ground (t2m). All the variables have the dimensions time, latitude, and longitude. The humidity variables have also the dimension level (1000 850 700 500 400 300 hPa); for these two variables, we have chosen the level 1000 hPa, which is near the surface (as the other variables).

The units for each dimension are:

- latitude: degrees north
- longitude: degrees east
- time: hours since 1900-01-01 00:00:0.0
- level: millibars

The shape of each file is shown in the next table. The time variables have 32 levels, 8 days times 4 observations per day. We use only the first time step when running our models, and 7 days for the best of them to show some longer runs because it is computationally expensive.

| | time | level | latitude | longitude |
|---|---|---|---|---|
| tcc | 32 | - | 61 | 77 |
| rh | 32 | 6 | 61 | 77 |
| q | 32 | 6 | 61 | 77 |
| sp | 32 | - | 61 | 77 |
| t2m | 32 | - | 61 | 77 |

Table 4.1: Dimensions of netcdf files.

The first 5 rows of the training set are shown in table 4.2.

| tcc | rh | q | sp | t2m |
|---|---|---|---|---|
| 0.9985 | 98.6066 | 0.0028 | 98941.0368 | 269.7680 |
| 1.0000 | 84.1849 | 0.0029 | 101904.9002 | 273.4086 |
| 0.8404 | 95.4674 | 0.0060 | 96063.5555 | 280.4226 |
| 0.0471 | 84.4467 | 0.0038 | 94983.2043 | 274.6886 |
| 0.2129 | 71.3095 | 0.0080 | 101414.2503 | 290.3192 |

Table 4.2: First rows of the train data.

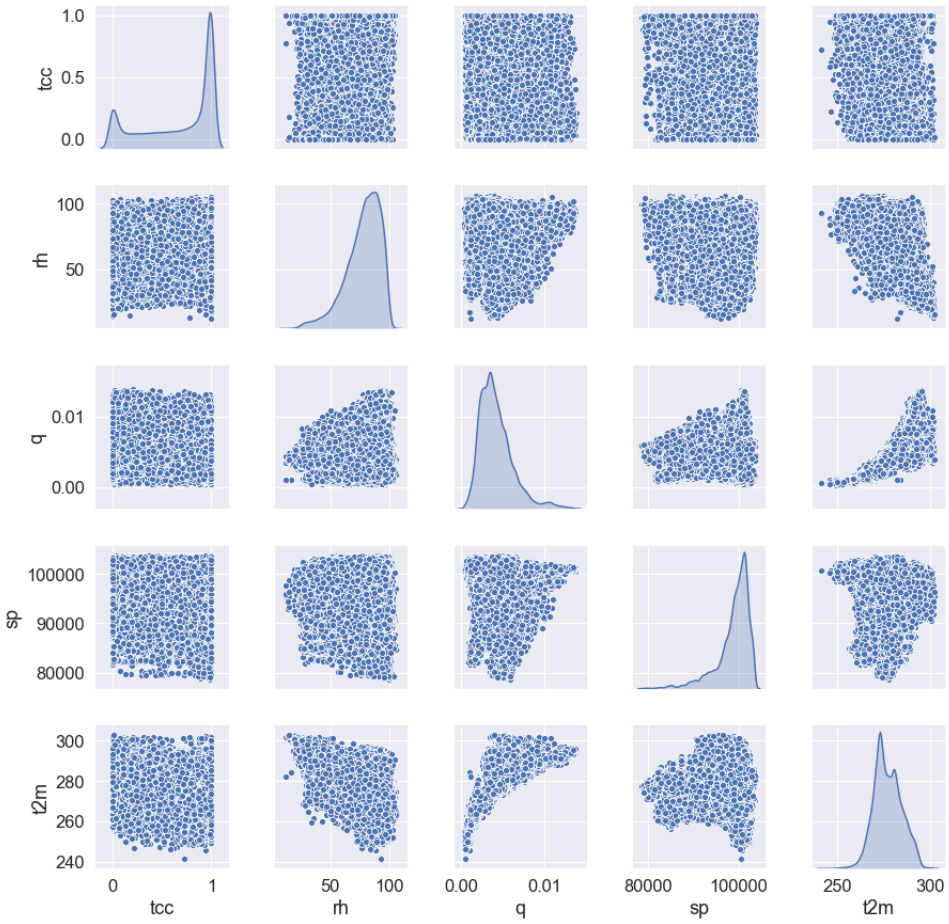


Figure 4.1: Scatterplot matrix and histograms of the data

The training data is displayed in figure 4.1. The empirical distribution of each variable is shown in the diagonal; the rest of the subplots are scatter plots of the data taking pairs of variables. Notice that the first row shows the response (tcc) against each of the predictors in turn. The data follows approximately the expected theoretic distributions. Since we have chosen a period where the NAO is very positive, we expect, in general, high values of total cloud cover and humidity (close to 1), and low pressure (a standard value for sea-level pressure is 1013 hPa). The response variable shows a peak close to 1 and a second maximum, but much lower, close to 0; between these to values the tcc seems to be uniformly distributed. The relative humidity distribution shows more frequent values around 90%, and its distribution is skewed to the right. The specific humidity does not vary with the temperature as the relative humidity does, its distribution is skewed to the left. Typically, the value of specific humidity varies by latitude. Higher values can be found at the equator with lower values at the poles. The pressure values are mostly below 1000 hPa, which is representative of a low-pressure system associated with upward movement of parcels that cools and eventually saturates (see Clausius-Clapeyron equation 3.3), hence increasing the cloudiness. The theoretical temperature distribution is a Gaussian, which in this case is not completely evident, and should be tested.

From the scatter plots in figure 4.1, we see a low correlation between the response and the predictors since we cannot distinguish any particular pattern in the data. As expected, there is a correlation between the other variables.

Table 4.3 contains the statistics of the normalized training data.

According to Ahrens (2007) [1], the average specific humidity ranges from 0.004 kg/kg at 60 degrees (north or south) to 0.018 kg/kg at the equator, so we can see that we have higher values than normal. The minimum pressure levels are too low, so we should consider correcting the data for outliers. The mean temperature values, $1^oC$ are typical of the wintertime, and all the other variable seems to have normal values.

| | tcc | rh | q | sp | t2m |
|---|---|---|---|---|---|
| count | 1.052130e+05 | 1.052130e+05 | 1.052130e+05 | 1.052130e+05 | 1.052130e+05 |
| mean | 6.6935e-01 | 78.0258 | 0.0042 | 98960.5897 | 277.3905 |
| std | 3.6362e-01 | 14.7660 | 0.0020 | 3590.9330 | 7.4874 |
| min | 9.9987e-13 | 11.8839 | 0.0003 | 78702.6875 | 241.2994 |
| 25% | 3.5977e-01 | 69.9459 | 0.0028 | 97688.3055 | 272.1735 |
| 50% | 8.4075e-01 | 80.6460 | 0.0038 | 99918.4975 | 276.7315 |
| 75% | 9.9686e-01 | 89.158527 | 0.0052 | 101376.6261 | 282.5273 |
| max | 1.0000e+00 | 106.3440 | 0.0138 | 103829.8757 | 302.7093 |

Table 4.3: Statistics of the training set.

## 4.2 Linear regression

The linear regression data analyses are all performed on data for seven days (28 timesteps). We have normalized the result in order to make all the models comparable. As expected none of the regression models performed well on this complicated data set, none of the models showed no improvement by introducing penalties. The best model had a MSE  0.9 and this was OLS without using the logit transformation on the predictor space and with no feature scaling. This is shown in figure 4.2. We attempted to use linear regression on the normal data and with logit transformation of the predictor space, this gave a worse result. The plots are available in the GitHub repository.

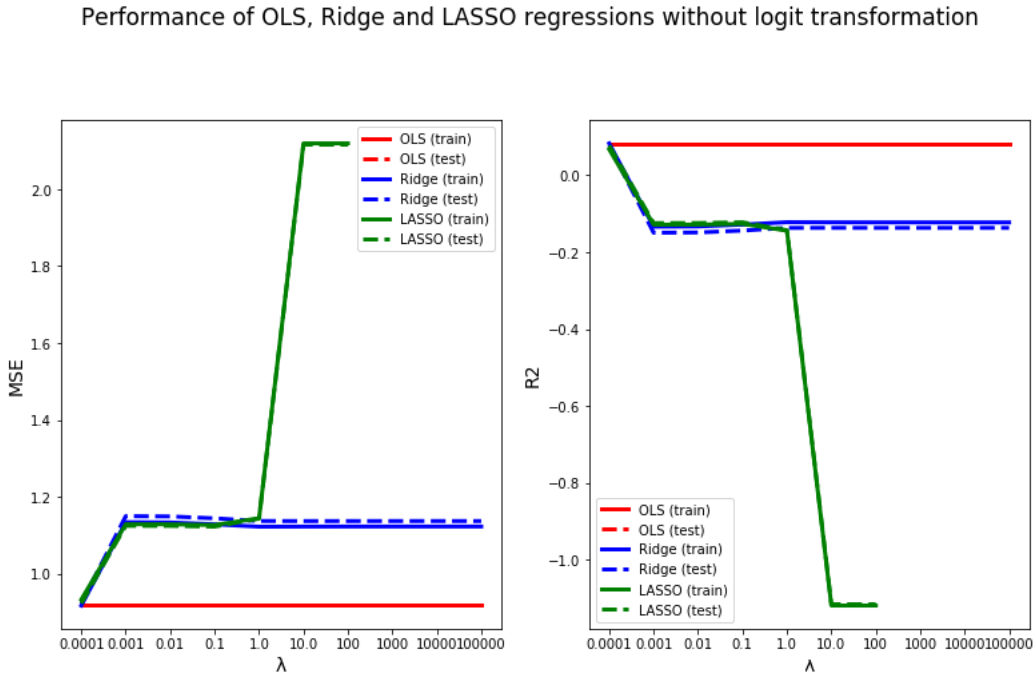


Figure 4.2: Linear regression on 7 days of data.

### 4.2.1 Bias variance analysis

Table 4.4 shows that the MSE of the test sample for the OLS and Ridge optimal lambda is 0.0947. All the schemes present a higher MSE than the sum of the variance and the bias, even though the difference is small ($< 10^{-9}$). These results are not normalized and thus the MSE is not on the scale of the other plots.

| | Variance | $\text{Bias}^2$ | MSE |
|---|---|---|---|
| Ridge, $\lambda = 0.001$ | $3.62 \cdot 10^{-6}$ | 0.0947 | 0.0947 |
| Lasso, $\lambda = 0.001$ | $3.94 \cdot 10^{-6}$ | 0.0957 | 0.0957 |
| OLS | $3.47 \cdot 10^{-6}$ | 0.0947 | 0.0947 |

Table 4.4: Bias variance tradeoff.

## 4.3 Neural networks

Applying the brute force approach in order to find the best combination results in a lot of plots. We have chosen to show the best result here and the rest will be discussed and the plots are available in the GitHub repository Examining the training MSE and training R2 always converged rapidly to 0 and 1. For all numbers hidden layer we calculated the result for one time step, one time step using the logit transformation and for 7 days (28 time steps).

### 4.3.1 One hidden layer

#### Self-implemented

Applying the brute force approach in order to find the best combination results in a lot of plots. We have chosen to show the best result here and the rest will be discussed and the plots are available in the GitHub repository For one hidden layer, we tested two activation functions; sigmoid and elu. On one timestep both elu and sigmoid activations result in $MSE$ 1.00, this gets slightly improved by combining sigmoid activation function and logit transformation on the data. The best performance comes by combining the elu activation function and the logit transformation $MSE \approx 0.80$. This figure showed some irregular curves this indicates that using the $\eta = 0.0001$ is a to high learning rate. The overall best performance for a one layer model both low $MSE \approx 0.70$ and relatively smooth curves was obtained using simoid activation and 7 days of data. Using elu activation we got approximately the same MSE but the curves oscillate. This indicates that larger amounts of data and the elu activation function need a slower learning rate.

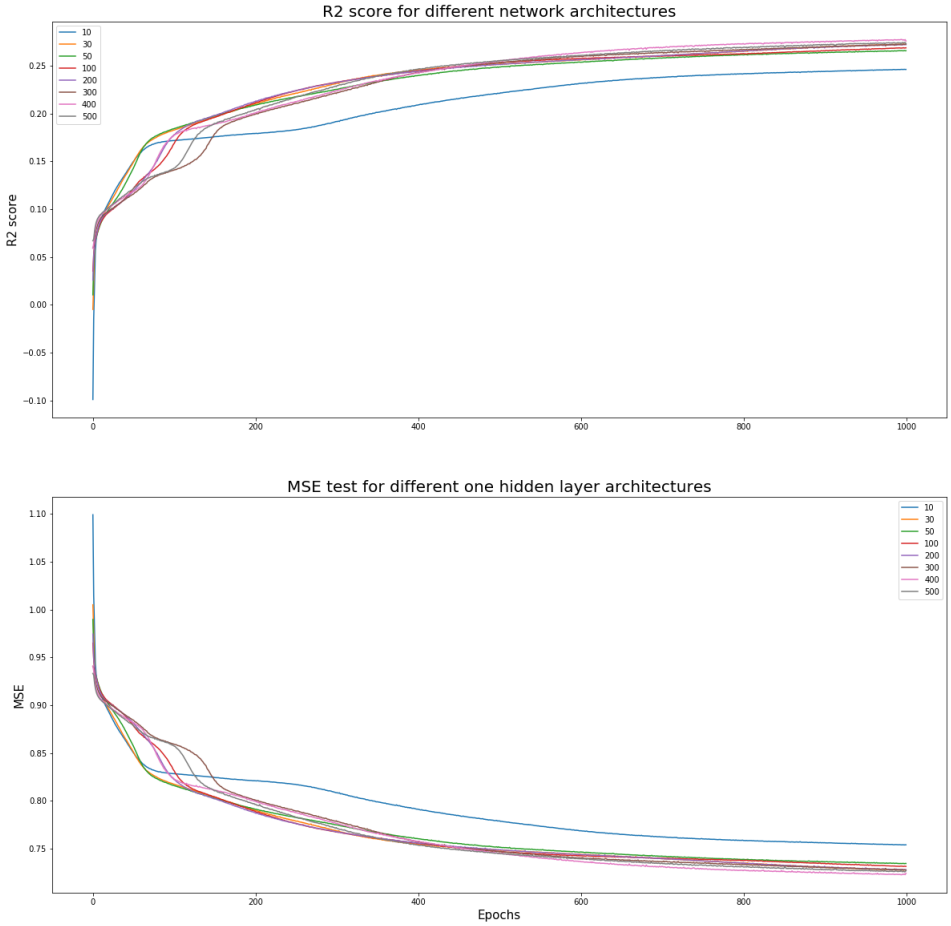


Figure 4.3: Illustrates performance using seven days of data, one hidden layers, sigmoid activation function, $eta = 0.0001$ and 1000 epochs.

**Keras**

Figure 4.4 shows the MSE of the model implemented with the Keras API, consisting of one hidden layer of 64 nodes, where the activation function is relu and the optimizer is RMSProp algorithm. The cost decreases rapidly within the first 100 epochs, after 200 epochs it remains stable. We set the maximum epochs to 1000 but used a callback that tests a training condition for every epoch. If a set amount of epochs elapses without showing improvement, then the training automatically stops.

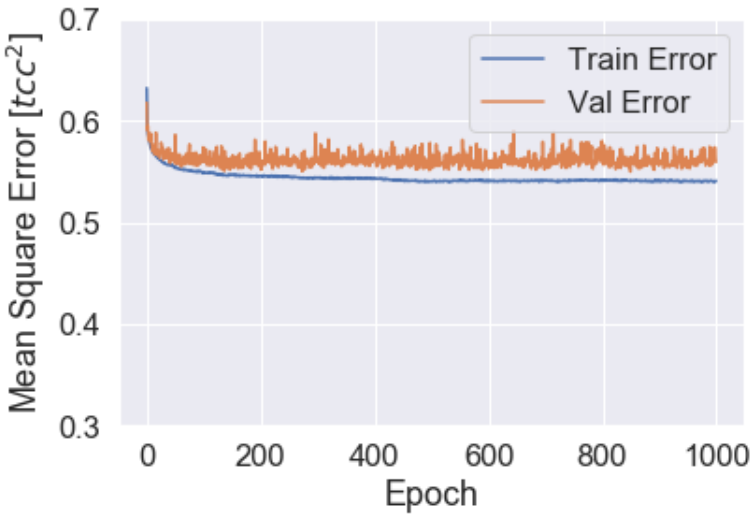


Figure 4.4: MSE of test and train data using keras with 1 hidden layer of 64 nodes, RMSPropOptimizer and relu for only one time step.

For one model, the training stops at 300 epochs, and the train and test MSE of the last 5 epochs is presented in table 4.5. The R-squared of the test data for 300 epochs is 0.41.

| Epoch | MSE test | MSE train |
|---|---|---|
| 296 | 0.574795 | 0.571273 |
| 297 | 0.573959 | 0.571304 |
| 298 | 0.582787 | 0.571089 |
| 299 | 0.585952 | 0.572036 |
| 300 | 0.580905 | 0.571667 |

Table 4.5: MSE of last 5 epochs for 1 hidden layer and 64 nodes using keras, RMSPropOptimizer and relu.

### 4.3.2 Two hidden layer

Using elu as activation function for self-implemented neural networks with more than one layer only resulted in NaN's. From this point on all plots of self-implemented neural networks use sigmoid as activation function. Using the logit transformation resulted in the smoothest curves, but not surprisingly the model with the largest amounts of data gave the best result. A $MSE \approx 0.85$ when we used all of data. This graph is shown in figure 4.5.

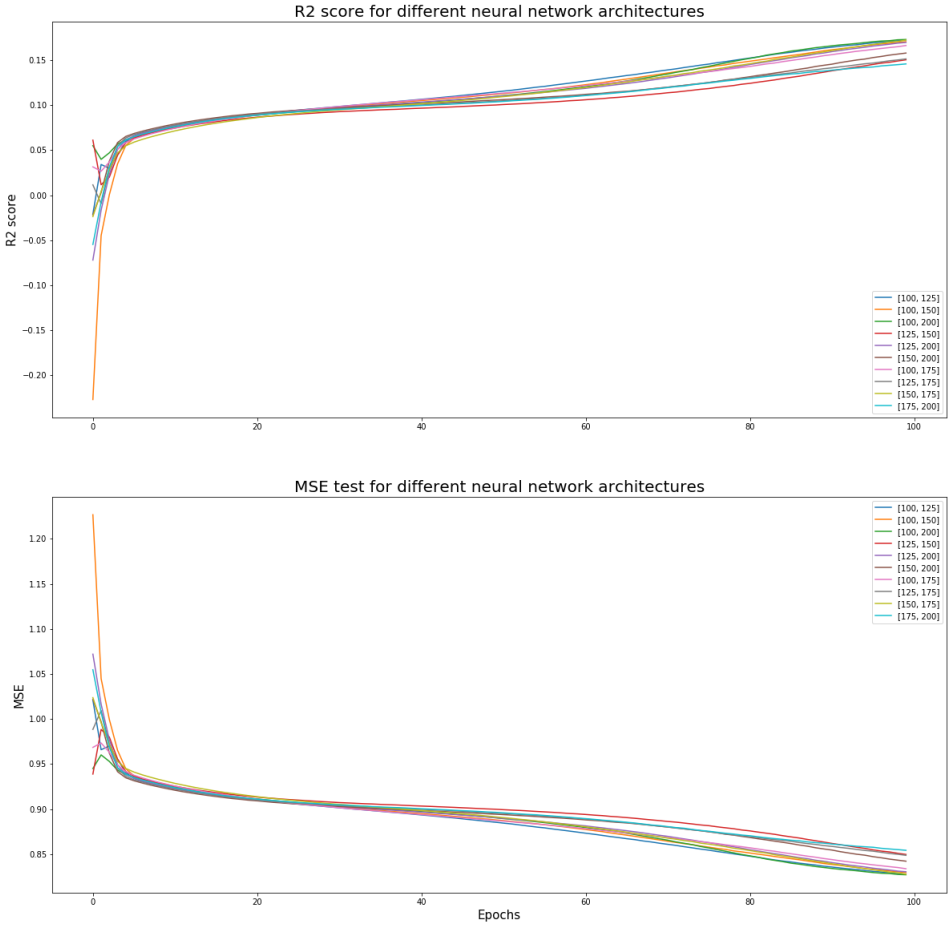


Figure 4.5: Illustrates performance using 7 days of data, two hidden layers, sigmoid activation function, $\eta = 0.0001$ and 100 epochs.

**Keras**

The cost, MSE, is calculated and plotted for the test (orange line) and train data (blue line) using Keras with 2 layers, RMSPropOptimizer and relu for only one time step is shown in figure 4.6. We see that the cost decreases rapidly during the first hundred epochs, then it continues decreasing but at a slower rate, and then stabilizes. The MSE lies between 0.40 for the train data and 0.48 for the test data, and the R-squared of the test data for 1000 epochs is 0.44. As we can see, the metrics are better than those of the 1-hidden layer models.

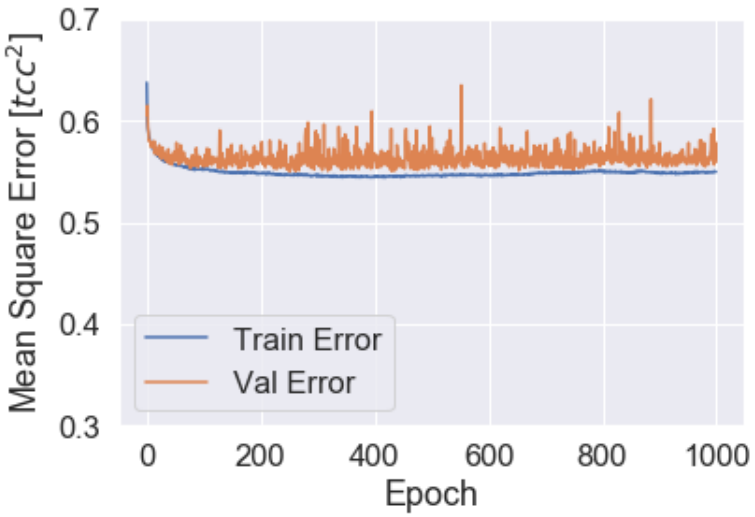


Figure 4.6: MSE of test and train data using Keras with 2 hidden layers of 64 nodes, RMSPropOptimizer and relu for only one time step.

The scatter plot in figure 4.7 shows that the predictions tend to overestimate the data on clear days and underestimate on overcast days. when using a neural network consisting of 2 layers with 64 nodes, relu and RMSPropOptimizer. This behavior is also observed when running other models, with different activation functions, optimizers, and number of hidden layers and nodes.

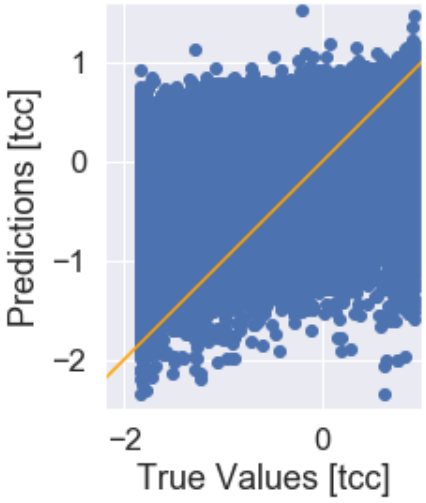


Figure 4.7: Scatterplot of the predicted data versus the true data using Keras with 2 hidden layers of 64 nodes, RMSPropOptimizer and relu for only one time step.

The histogram of the errors using the same model is displayed in figure 4.8. At a simple glance, it looks like a normal distribution; but we observe that, even though most of the errors are close to zero, some values are almost as large as the maximum and minimum tcc values in the test set. These points contribute to a higher MSE. In the future, it would be interesting to investigate it closer and try to find out whether it is a physical condition that leads to a poor prediction. If that is the case, the points could be left out of the analysis, improving the performance of the model.

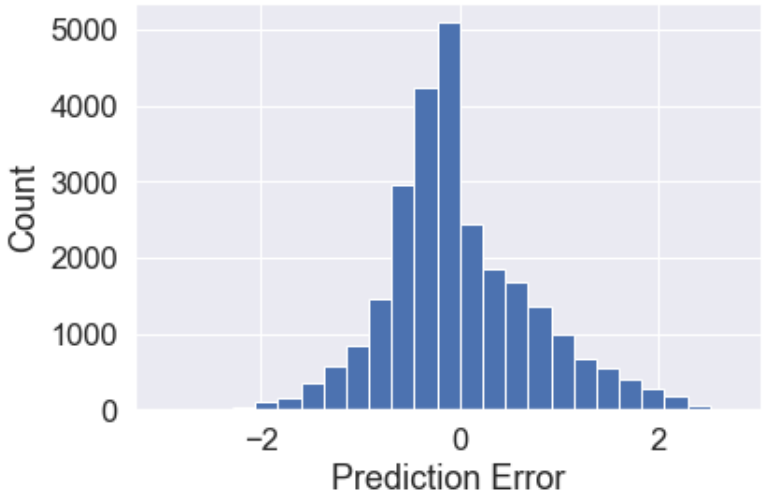


Figure 4.8: histogram of the error using Keras with 2 hidden layers of 64 nodes, RMSPropOptimizer and relu for only one time step.

Other activation functions like sigmoid and elu, and optimizers like SGD and Adam were tested as well, without improving notably the results. Also 32 nodes instead of 64 were considered. The corresponding figures can be found in the notebook keras_nn.ipynb.

### 4.3.3 Three hidden layer

#### Self-implemented

Figure 4.9 shows the model performance for several combinations of the layers, all the number of nodes in one layer are in the range [100, 200]. The model uses data for a period of seven days. Both the plot using sigmoid and the one with a logit transformation of the data and sigmoid activation function had an MSE of approximately 1.0. This can be found in the GitHub repository. This result in an MSE of approximately 0.85. The curves do not show a perfect shape, this can be explained by a too high learning rate.

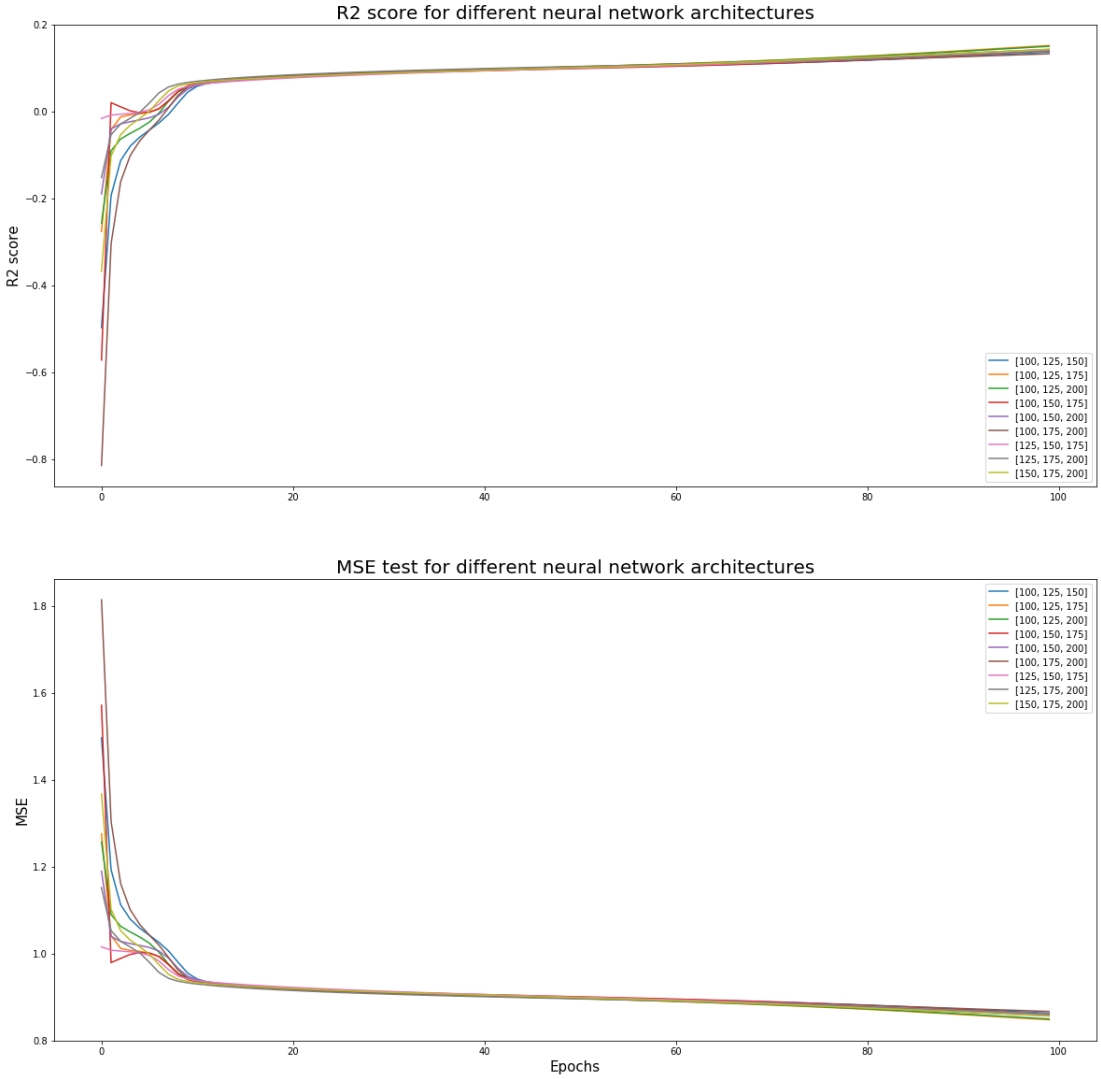


Figure 4.9: Illustrates performance using 7 days of data, tree hidden layers, sigmoid activation function, $\eta = 0.0001$ and 100 epochs.

#### Keras

As we can see in figure 4.10, adding a third hidden layer to the model does not improve the results. The model converges after after 265 epochs (more than with two layers) to an $MSE = 0.54$, and an $R^2 = 0.44$.

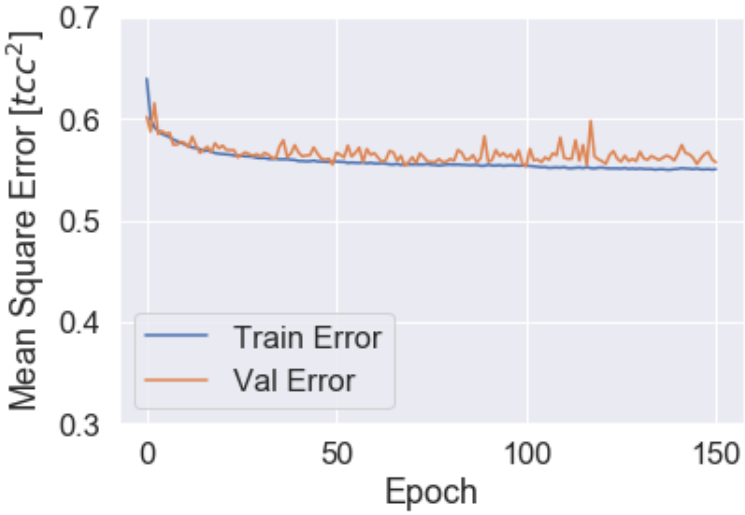


Figure 4.10: MSE of test and train data using Keras with 3 hidden layers of 64 nodes, RMSPropOptimizer and relu for only one time step.

### 4.3.4 Four hidden layer

The model with four layers performs worse than the models with fewer layers. Either this is the wrong depth, or the optimal number of nodes when we use four layers doesn't lie in the range [100,200]. Among the three lines in 4.11 the green curve has the best performance it has both a great shape and it results in the lowest $MSE \approx 1$. This plot shows the desired shape indicating that the model is trained with the correct learning rate. The curves generated after using a sigmoid transformation has a higher MSE and a worse shape.

Comparing the results in figure 4.12 and 4.11 it is evident that the plot which was trained on the least amount data performs slightly better. These were both trained for their optimal learning rate, this was $\eta = 0.0001$ for the shortest timestep and $\eta = 0.00001$ for the longer timestep.

#### Self-implemented

In order to examine the importance of the ordering of the nodes we build one model for each permutation of the best combo of nodes in figure 4.11 {125,150,175,200}. The result is displayed in figure 4.13, the same color appear several times which makes it impossible to distinguish which of the model perform best. The figure still serves it purpose by illustrating the large span in performance as a function of epochs. You can find a similar plot using logit function on the same amount of data in the appendix see figure 6.1.

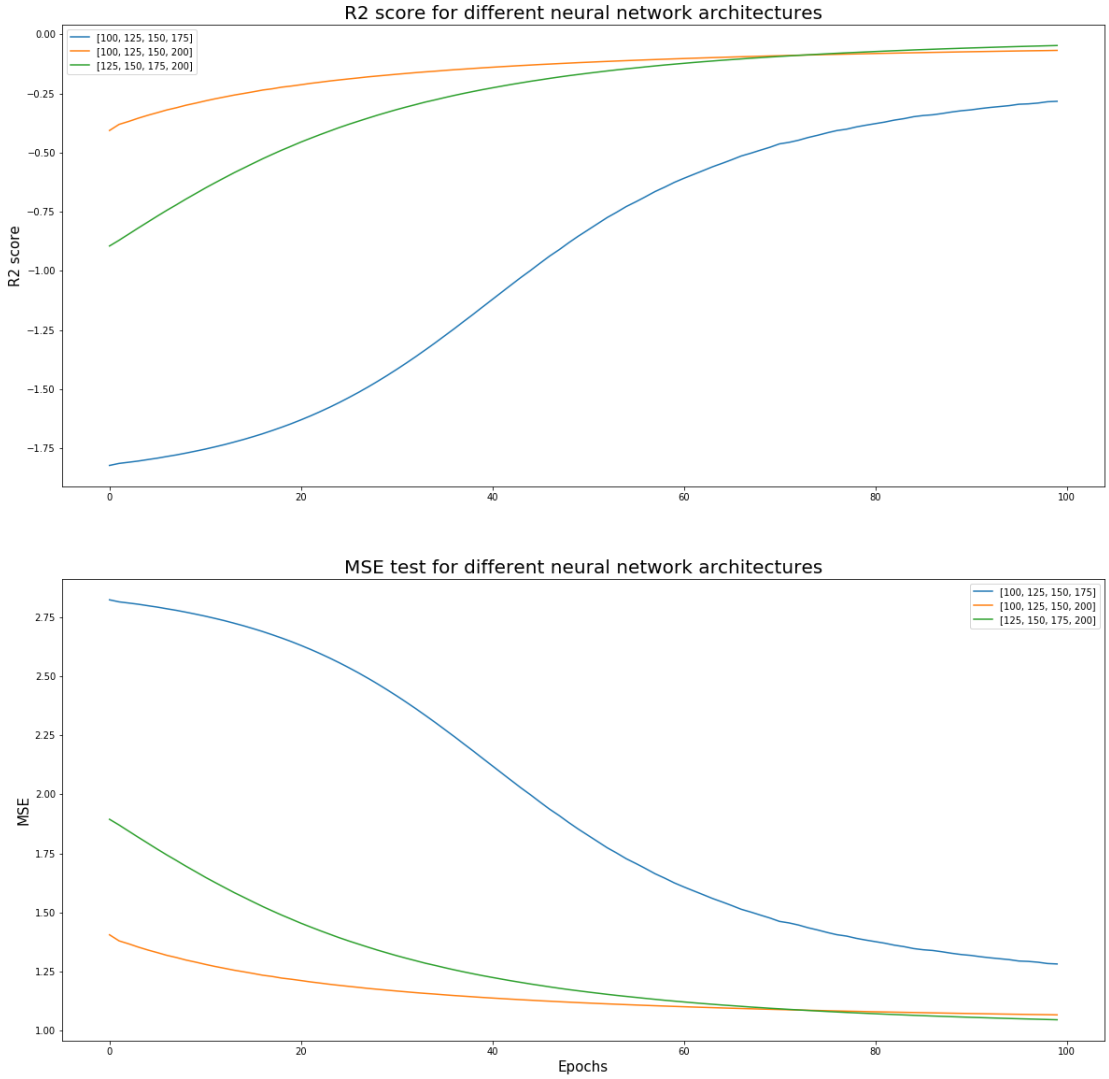


Figure 4.11: Illustrates performance using 1 timestep of data, four hidden layers, sigmoid activation function, $eta = 0.0001$ and 100 epochs.

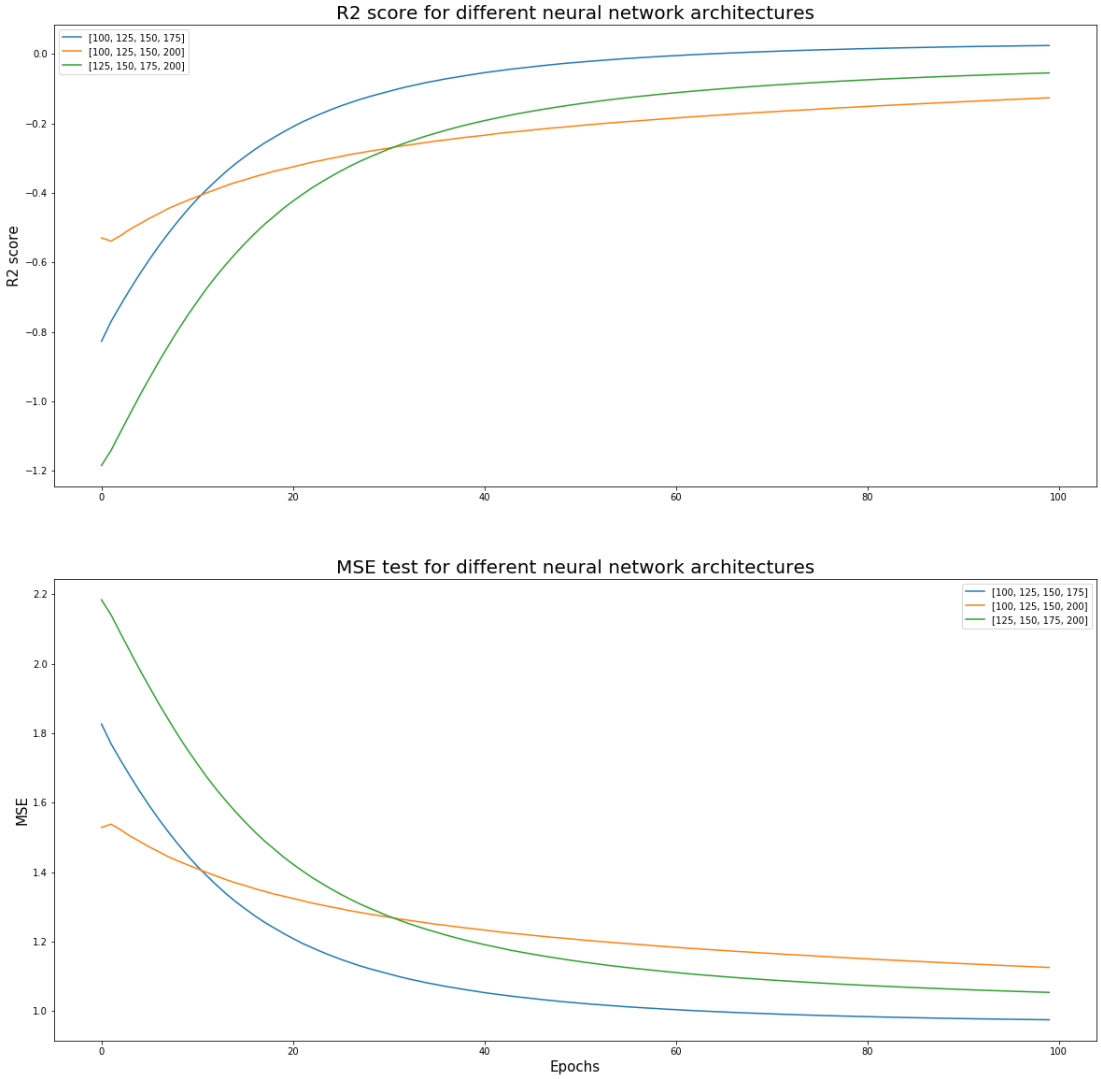


Figure 4.12: Illustrates performance using 7 days of data, four hidden layers, sigmoid activation function, $eta = 0.00001$ and 100 epochs.

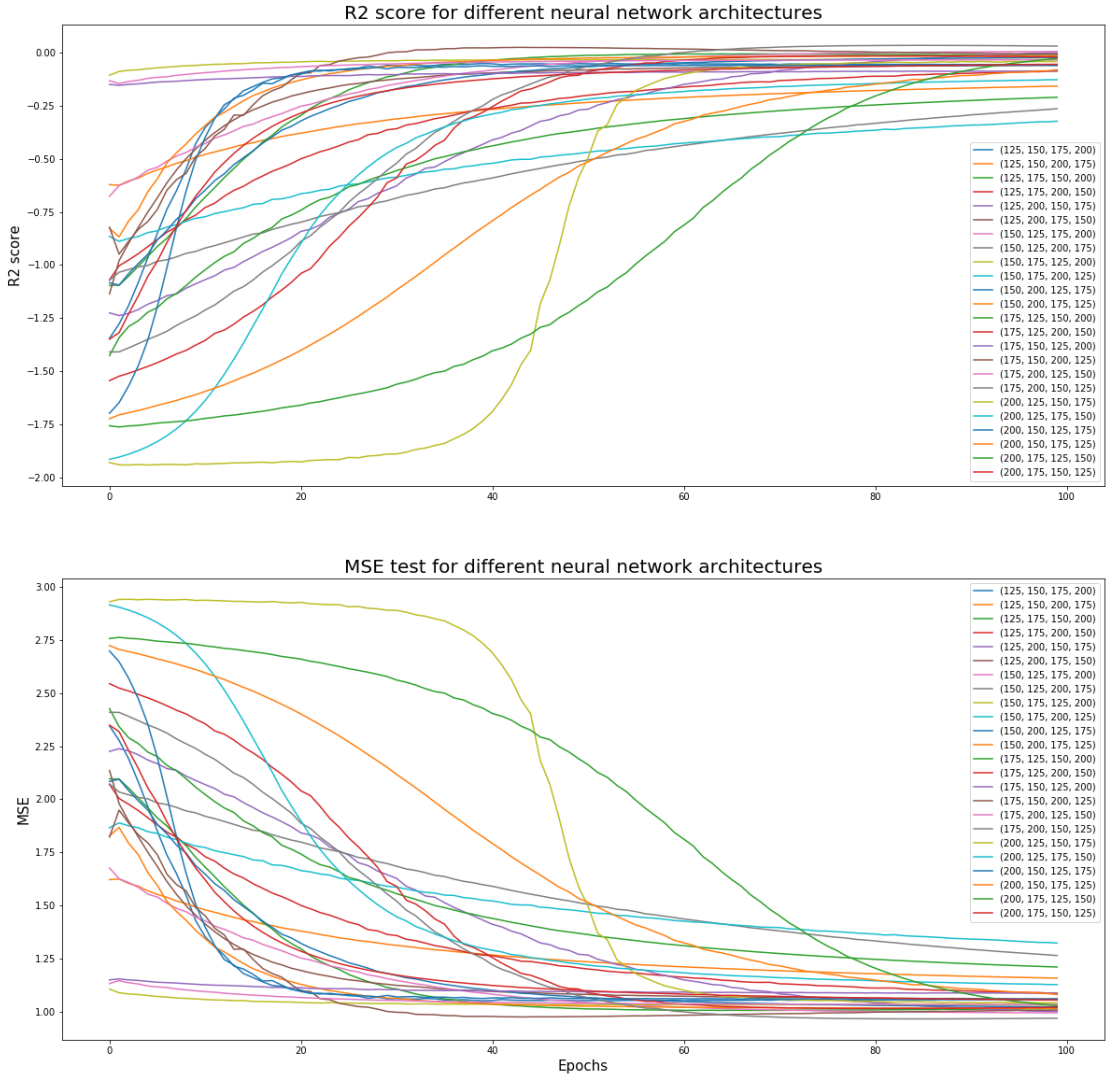


Figure 4.13: Illustrates the performance using one timestep of data, four hidden layers, sigmoid activation function, $eta = 0.00001$ and 100 epoch of all permutations of {125,150,175,200}.

**Keras**

Again, adding another hidden layer to the model does not result in better predictions, as can be observed in figure 4.14. With four hidden layers, the model converges after 117 epochs to an $MSE = 0.55$ and an $R^2 = 0.43$.

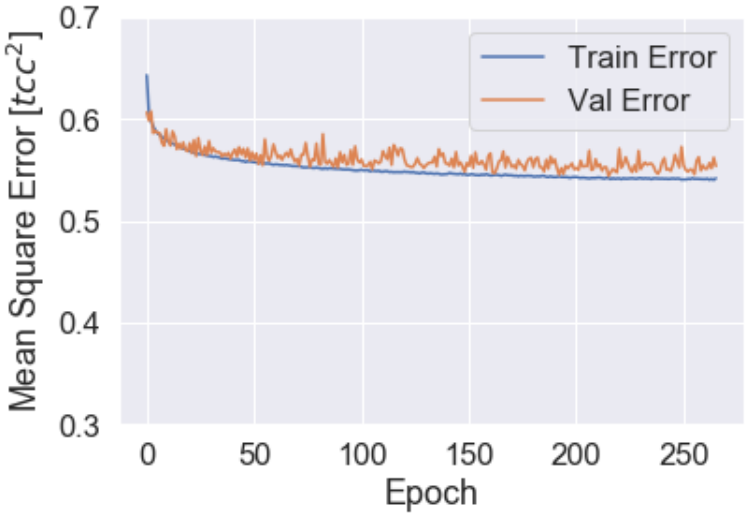


Figure 4.14: MSE of test and train data using Keras with 4 hidden layers of 64 nodes, RMSPropOptimizer and relu for only one time step.

In conclusion, the best models we have tested with different libraries consists always of 2 hidden layers.

## 4.4 Trees-based algorithms

The figures presented in this section correspond to only one time step. Some figures for 7 days can be found in the Appendix, but running for longer periods does not improve the prediction.

### 4.4.1 Decision trees

For finding the best model that fit the data, two parameters were varied: the maximum depth of the tree and the number of leaves. Ideally, we would grow a large tree and then prune it, but this is not implemented in scikit-learn yet, although it is planned to be done. The method used in this paper to select the parameters was to plot the MSE and the R-squared trying to optimize these and avoid over- and underfitting, as well as look at the error and try to find a combination of parameters that lead to a Gaussian-like distribution.

Figure 4.15 is plotted for a maximum depth equal to 4 (the corresponding figure can be found in the notebook "trees_t0"). The model was run several times, showing slight changes in the curves, especially for more than 20 maximum leaves. In all cases tested, the train and test are steep and close to each other up to approximately 7 maximum leaves, and then separate and continue increasing in the case of the R-squared, or decreasing in the case of MSE, but with a slower rate. This is a sign of underfitting, that the model cannot capture all the features. In this particular case, we have chosen a maximum number of leaves equal to 25, but it might be different for other random states. Most of the runs for this model shows, in the best case, R-squared values around 0.40 and MSE around 0.65.

The scatter plot of predicted data versus true test data is shown in figure 4.16; the 1:1 line is plotted as a reference. We can see that there is little correlation between the predictions and the true data, the points are dispersed, and no pattern can be identified. However, the R-squared shows that 40% of the variance is explained by this model, which is a very good result in a meteorological context.

The histogram of the error (prediction - true data) in figure 4.17 shows higher frequency for negative er-

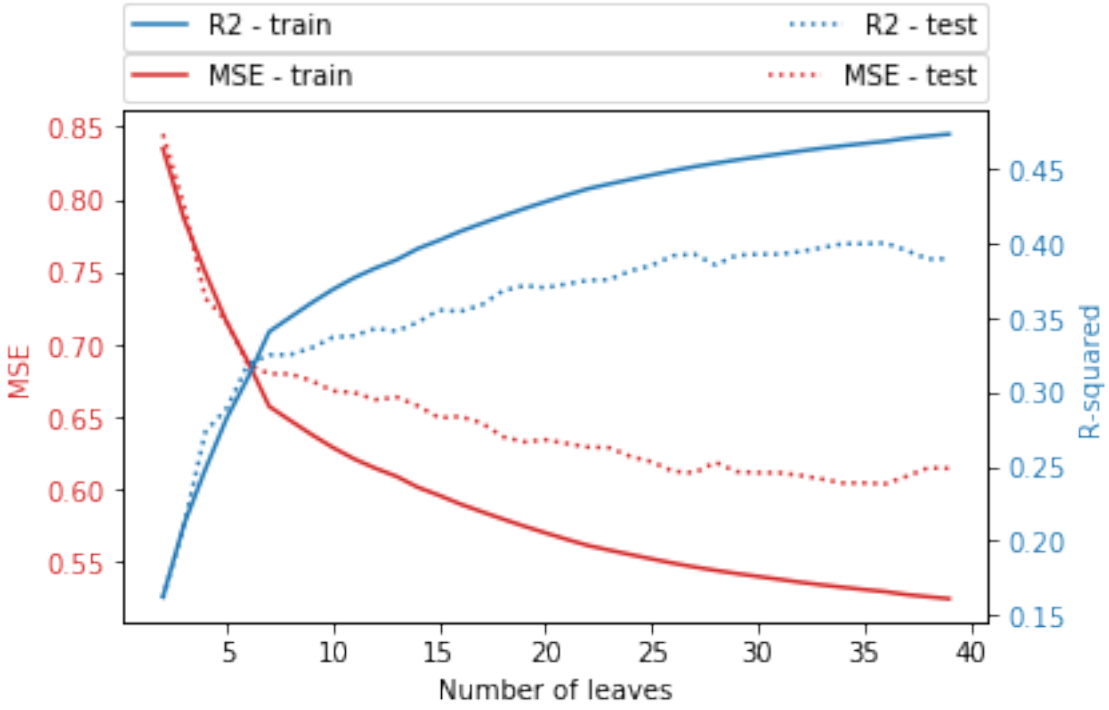


Figure 4.15: MSE and R-squared of train and test data using decision trees for max. depth = 4 in terms of max. number of leaves.

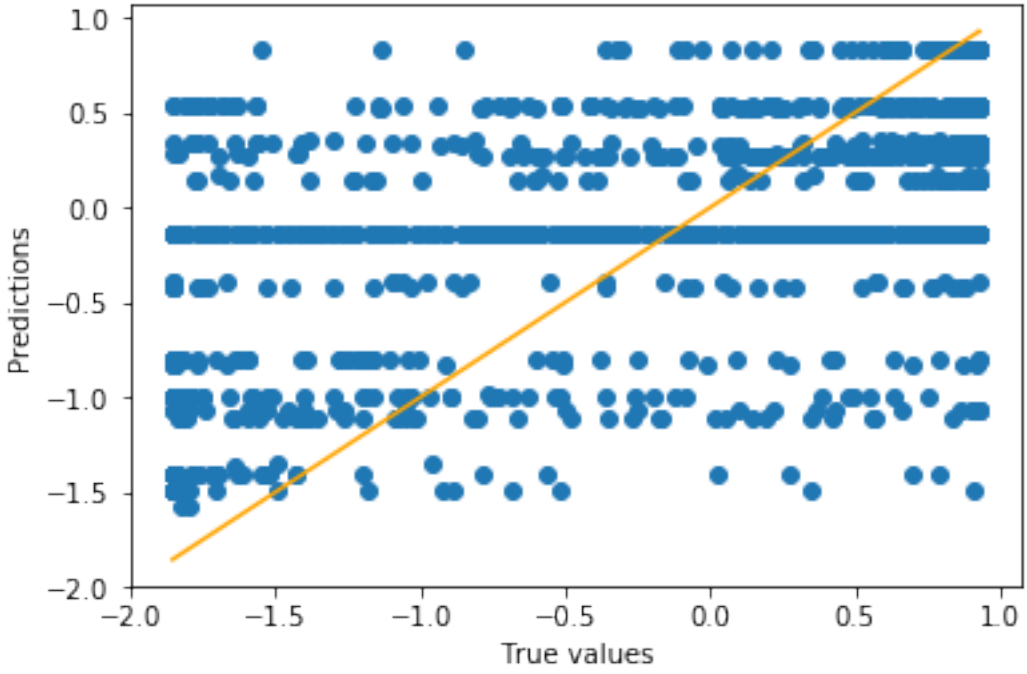


Figure 4.16: Scatter plot of true data and predicted data using decision trees with max. depth = 4 and max. number of leaves = 25.

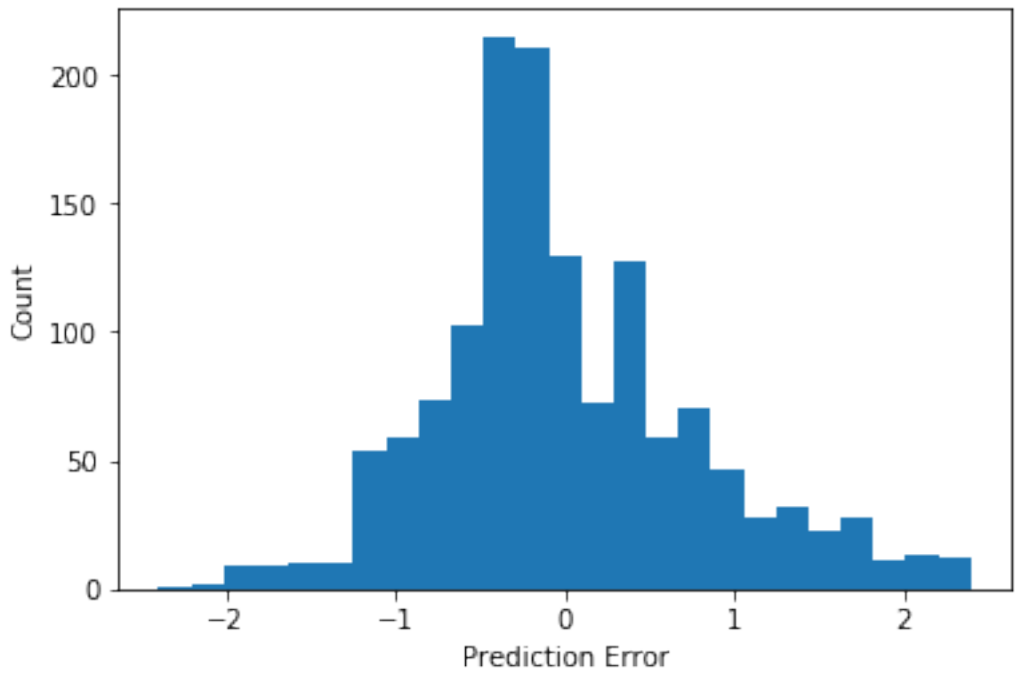


Figure 4.17: Histogram of the errors using decision trees with max. depth = 4 and max. number of leaves = 25.

rors close to zero, which means that the prediction is lower than the true values, and thus our model tends to underestimate. However, this combination of parameters is the one that gives a more Gaussian-like distribution of the errors.

### 4.4.2 Random forest

Random forest conduct to better results than decision trees, but appears to be more sensitive to changes in random states and have a larger tendency to underestimate. The parameters tuned are maximum depth, the maximum number of leaves, and the number of estimators. The procedure is analogous to that of decision trees, and all the figures can be found in the notebook.

figure 6.3 was obtained for a maximum depth equal to 9, a maximum number of leaves equal to 25 and 3 estimators. This figure is very sensitive to changes in random states, so the optimal parameters might change. Furthermore, after running the same code for different states, we conclude that the maximum R-square lies around 0.47, and the minimum MSE is approximately 0.45, which are better than the scores for decision trees.

Again, it is hard to find any particular pattern in the scatter plot of predicted data versus true values. The histograms of the errors show also that when employing random forest the model tends to underestimate the data.

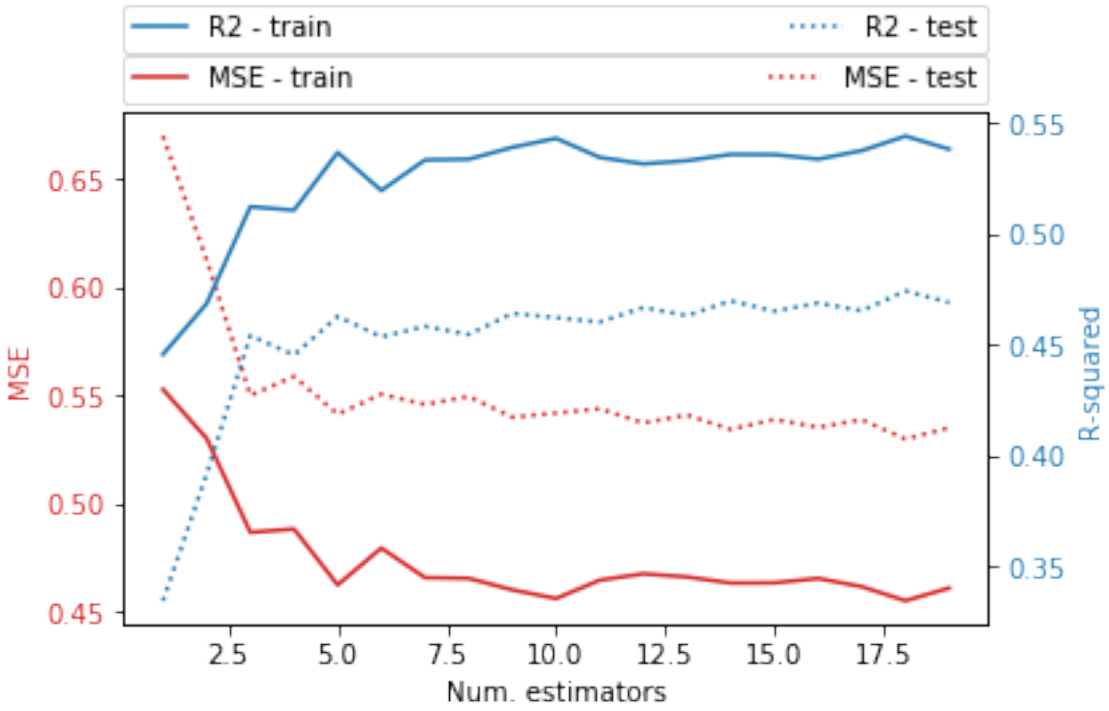


Figure 4.18: MSE and R-squared of train and test data using random forest for max. depth = 9, max. number of leaves = 25, in terms of number of estimators.

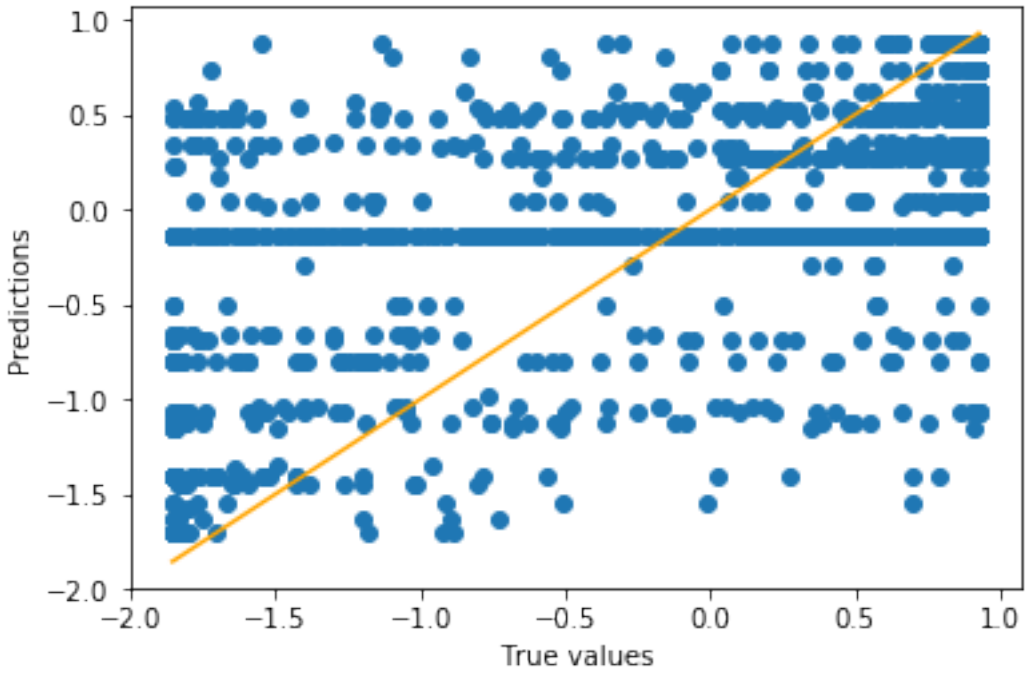


Figure 4.19: Scatter plot of true data and predicted data using random forest with max. depth = 9, max. number of leaves = 25, and number of estimators = 3.

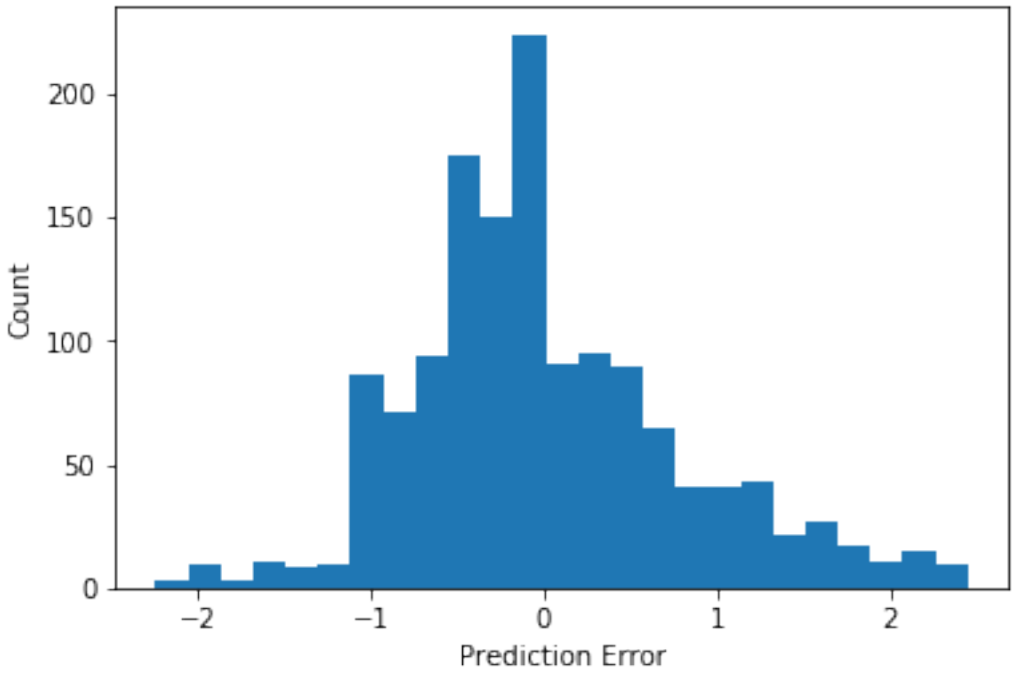


Figure 4.20: Histogram of the errors using random forest with max. depth = 9, max. number of leaves = 25, and number of estimators = 3.

# Chapter 5

# Conclusion

The models used in this project explain a large part of the variance of total cloud cover compared to standard climate models, despite that they do not provide a perfect fit, and invites to continue the research on this field.The self-implemented version of a neural network could not compete with the Python libraries for neural networks. Keras and Tensorflow both did best with a depth of two layers. Sadly the performance is still not good enough to accurately predict cloud cover, but in a next stage we will restrict our data to specific regions with similar geographic characteristics, analize carefully the outliers and study the temporal autocorrealation in the data.

We will try other networks which can include knowledge on coordinates and time steps i.e. convolutional neural networks (CNN). Other improvements to the models in this article is rewriting the code for the neural network more object-oriented and allow for different activation functions in different layers. Random forest trees outer perform the other models. This has the lowest MSE around 0.45. In the future we intend to study the effect of pruning the trees, by implementing it ourselves and combine it with the scikit-learn functionalities. We expect this to reduce the chances of overfitting.

For the future we intend to compute a common baseline. This will make it easier to access if the model performs well. A common baseline is the *random guess* of a data set. It the model predictions are as good as the common baseline it means that the model hasn't learned anything. This requires some precise meteorological assumptions on which we need to get a expert opinion.

We also intend to increase our computational capacity. In this article, we have used a relatively small data set to avoid memory errors and extremely slow programs. All our data contains cloudy weather, we need to increase the data amount if we wish to find a model which works for all types of weather phenomenon.

# Bibliography


[1] C. Donald. Ahrens. *Meteorology Today : an Introduction to Weather, Climate, and the Environment.* Springer Series in Statistics. New York, NY, USA: Belmont, CA :Thomson/Brooks/Cole, 2007.

[2] *Area examples ECMWF Post-processing keywords.* https://confluence.ecmwf.int/display/UDOC/Post-processing+keywords#Post-processingkeywords-area. Accessed: 2018-12-15.

[3] L. Bengtsson. "Four dimentional data assimilation". In: *European Centre for Medium Range Weather Forecasts, ECWMF* (1982).

[4] P. Berrisford et al. "The ERA-Interim archive. Version 2.0". In: (2011).

[5] Dee et al. "The ERA-Interim reanalysis: configuration and performance of the data assimilation system". In: *Royal Meteorological Society* (2011).

[6] *Feature scaling Wikipedia article.* https://en.wikipedia.org/wiki/Feature_scaling. Accessed: 2018-12-16.

[7] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics.* Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[9] Morten Hjort-Jensen. *Lecture notes Data Analysis and Machine Learning: Getting started, our first data and Machine Learning encounters.* https://compphysics.github.io/MachineLearning/doc/pub/How2ReadData/html/How2ReadData.html. Accessed: 2018-11-08.

[10] Morten Hjort-Jensen. *Lecture notes Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning and convolutional networks.* https://compphysics.github.io/MachineLearning/doc/pub/NeuralNet/pdf/NeuralNet-minted.pdf. Accessed: 2018-11-08.

[11] *Illustrations ch. 8: Anthropogenic and Natural Radiative Forcing AR5.* https://www.ipcc.ch/report/ar5/wg1/anthropogenic-and-natural-radiative-forcing/. Accessed: 2018-12-15.

[12] *Logit transformation Wikipedia.* https://en.wikipedia.org/wiki/Logit. Accessed: 2018-11-20.

[13] Lohmann, Lüönd, and Mahrt. *An Introduction to Clouds: From the Microscale to Climate.* Cambridge University Press, 2016.

[14] P. Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *ArXiv e-prints* (Mar. 2018). arXiv: 1803.08823 [physics.comp-ph].

[15] Micheal Nielsen. *Neural Networks and Deep Learning.* Determintation press, 2015.

[16] Sebastian Raschka. *Python Machine Learning.* Packt Publishing, 2017. ISBN: 1787125939, 9781787125933.

[17] *scikit-learn Module trees.* https://http://scikit-learn.org/stable/modules/tree.html. Accessed: 2018-12-16.

[18] *Ship tracks NASA earth observatory.* https://earthobservatory.nasa.gov/images/3275/ship-tracks-off-europes-atlantic-coast. Accessed: 2018-12-15.

[19] *Standford tutorial Optimization of SGD.* http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/. Accessed: 2018-11-08.

[20] *TensorFlow Keras Guide.* https://www.tensorflow.org/guide/keras. Accessed: 2018-12-16.

[21] *Toward datascience Understainding learining rate and how it improves performance in deep learning.* https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10. Accessed: 2018-12-16.

[22] S. M. Uppala et al. "The ERA-40 re-analysis". In: *Quarterly Journal of the Royal Meteorological Society* 131.612 (2006), pp. 2961–3012. DOI: 10.1256/qj.04.176. URL: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1256/qj.04.176.

[23] Paul Voosen. "The earth machine". In: (2018). URL: http://www.sciencemag.org/news/2018/07/science-insurgents-plot-climate-model-driven-artificial-intelligence.

# Chapter 6

# Appendix

## 6.1 Versions of Python packages

There are three available environments in the folder envs in the GitHub repository, use env_nn.txt when running the notebooks; LinearRegression.ipynb, NeuralNetworks_1timestep.ipynb, NeuralNetworks_7days.ipynb and NeuralNetworks_1timestep_logit.ipynb. The enviornments tflow.txt is for the notebooks; tensorflow_nn.ipynb and tensorflow_nn_different_results.ipynb. The notebooks trees_7days.ipynb and trees_t0.ipynb. We underline that these environment's are different and that they are used on different operatingsystems. The information on how to create an enviornment and which platform it runs on can be found in the first lines in the txt-file.

## 6.2 Figures

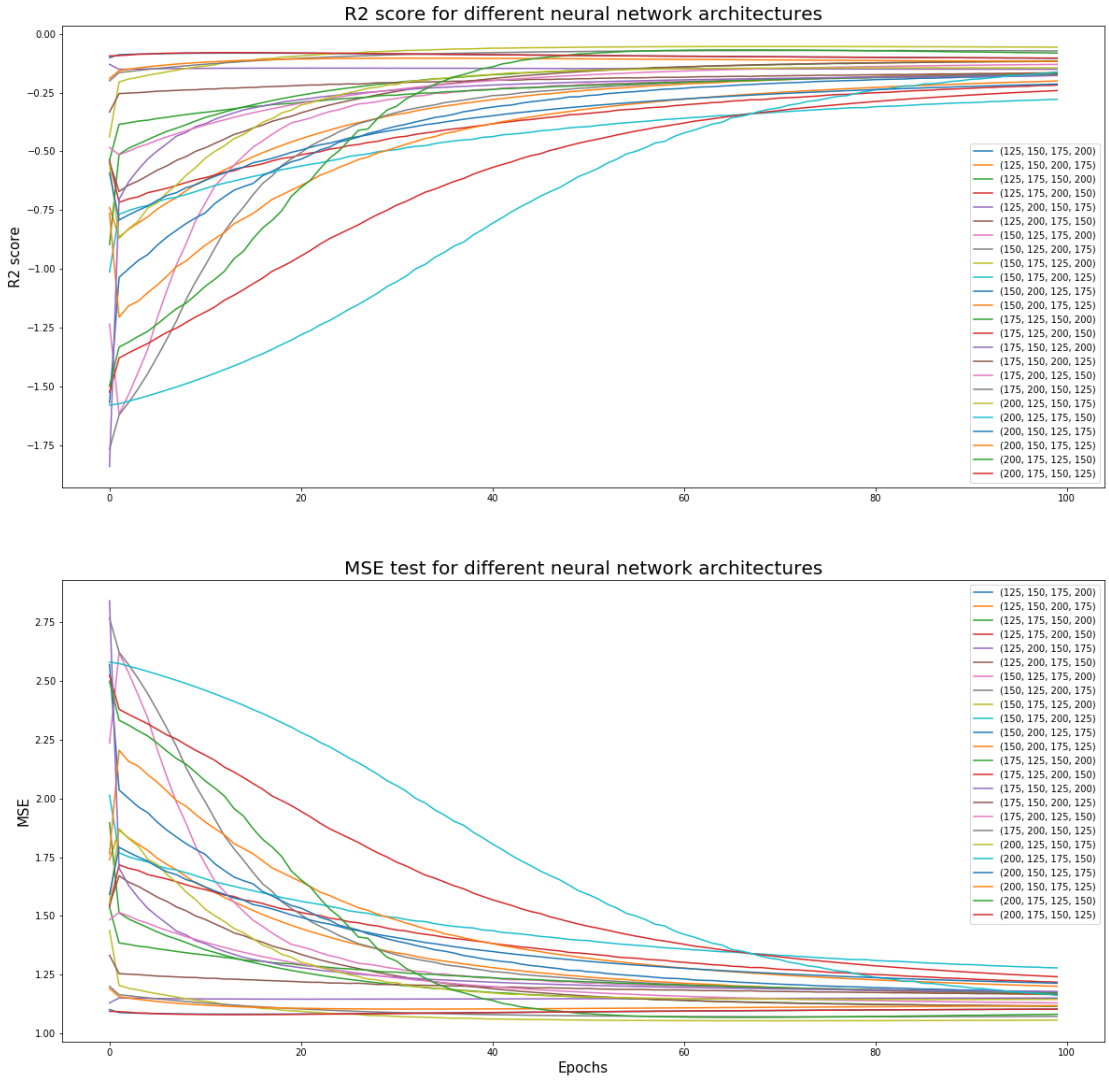


Figure 6.1: IIllustrates the performance using one timestep of data, four hidden layers, sigmoid activation function, logit transformation function, $eta = 0.00001$ and 100 epoch of all permutations of {125,150,175,200}.

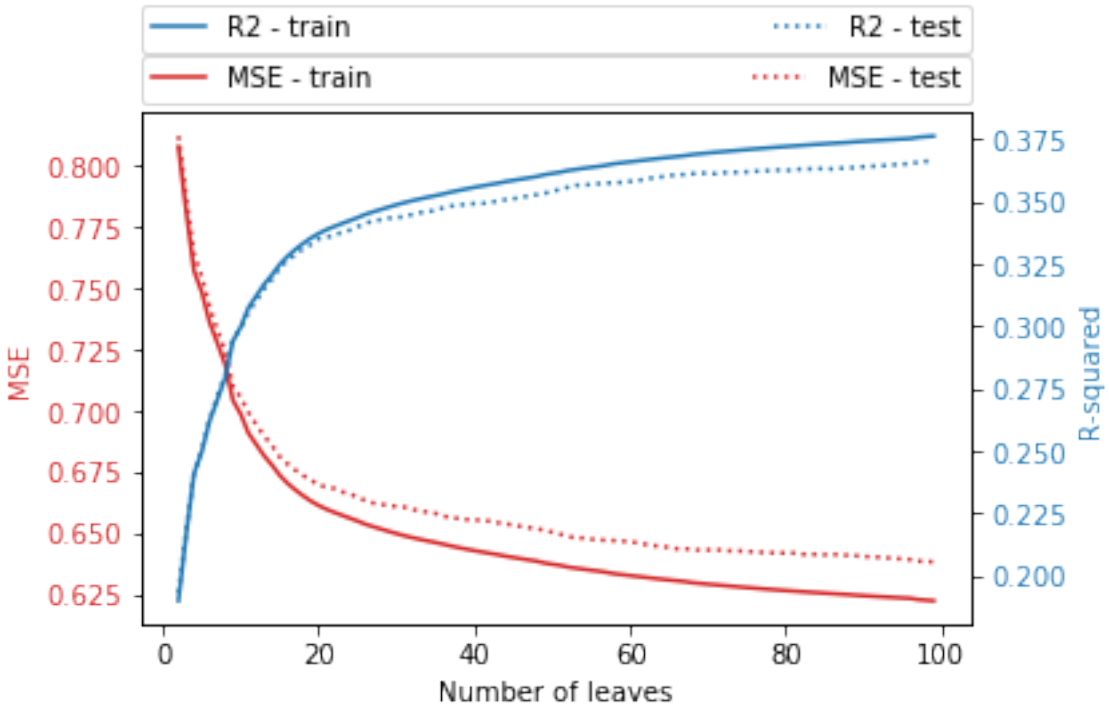


Figure 6.2: MSE and R-squared of train and test data using decision trees for max. depth = 10, max. in terms of the number of leaves.

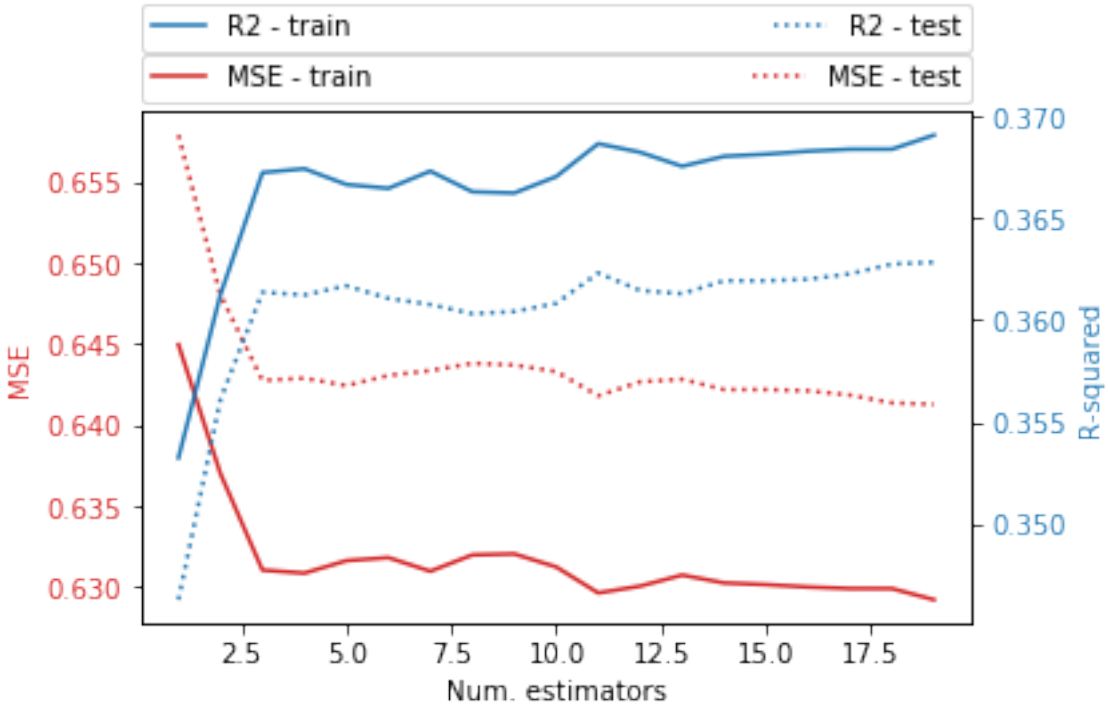


Figure 6.3: MSE and R-squared of train and test data using random forest for max. depth = 7, max. number of leaves = 40, in terms of number of estimators.