Norwegian School of Economics
Bergen, Spring 2019

Deep Learning With Application To Finance

FIE458 - Final Project

Candidate no:

2

3

15

43

Professor: Walt Pohl

Department of Finance

NORWEGIAN SCHOOL OF ECONOMICS

Abstract

In this paper we tried to address two objectives through the application of machine learning techniques. The first part concerned predicting the Bitcoin close price on a 24-hours basis, while the second part concerned predicting the direction of its movement on a 24-hours basis. Accordingly, the study experimented with three deep learning techniques on one hand and model ensembling of deep learning with two tree-based techniques on the other hand. Our findings in regards to the first objective showed that the models were not able to generalize well. Slight improvements were noted when comparing the different machine learning techniques to each other. However, a naive benchmark predicting today's price to be tomorrow's price was established for comparison, and none of the models were able to beat it. We therefore concluded that we were not able to predict the close price of Bitcoin, and that applying machine learning techniques was not worthwhile, in our particular case, with respect to the first objective. In regards to the second objective, the underlying method was model ensembling of different machine learning techniques. Our findings indicated some improvements in performance through model ensembling in comparison to the respective models' individual performances. However, the models' performances did not suggest predictability of the directions of the Bitcoin movement. Applying machine learning techniques to answer these two objectives, was therefore not as successful as one would hope, and thereby this project, provides further arguments towards the unpredictabililty of the cryptocurrency market.

Contents

Abstract

List of Figures **ii**

List of Tables **iii**

1 Introduction **1**

2 Problem Description **2**

3 Introduction to Neural Networks and Deep Learning Techniques **4**
3.1 History . . . 4
3.2 Neural Networks . . . 4
3.3 Dense Neural Networks . . . 6
3.4 Recurrent Neural Network . . . 7

4 Tuning the Neural Networks **11**
4.1 Consistent parameters . . . 11
4.2 Tuning strategy . . . 12

5 Data sets **16**
5.1 Original data set . . . 16
5.2 New data set . . . 16
5.3 Principal Component Analysis . . . 22
5.4 Autoencoder . . . 23
5.5 Data Split . . . 27

6 Analysis **28**
6.1 Simple Dense Neural Network . . . 28
6.2 Deep Dense Neural Network . . . 30
6.3 Recurrent Neural Network . . . 35
6.4 Concluding remarks - DNN and RNN . . . 40

7 Model Ensembling **43**
7.1 Introduction to Model Ensembling . . . 43
7.2 Introduction to Random Forest and Gradient Boosting Machine . . . 43
7.3 Approach . . . 44
7.4 Implementation . . . 44
7.5 Results – Random Forest, Gradient Boosting Machine and Neural Network . . . 48
7.6 Ensembling the Models . . . 50
7.7 Results – Ensembled Models . . . 51
7.8 Concluding Remarks - Model Ensembles . . . 53

8 Conclusion **55**

9 Limitations **56**

10 References **57**

11 Appendices **61**
11.1 Appendix A: Correlation Matrix 61
11.2 Appendix B: Training loss, simple dense 61

List of Figures

1 Benchmark: Movement of actual and predicted values . . . 3
2 Depiction of a node . . . 5
3 Simple dense neural network . . . 6
4 Deep neural network . . . 6
5 Recurrent Neural Network feedback loop . . . 7
6 Single GRU unit . . . 9
7 Price of Bitcoin from new data set . . . 17
8 Principal Components Analysis . . . 23
9 Deep Learning Autoencoder . . . 24
10 Deep Learning Autoencoder . . . 26
11 Predicted results vs. actual . . . 29
12 Training comparison . . . 32
13 Predicted results vs. actual . . . 33
14 Predicted results vs. actual . . . 37
15 Predicted results vs. actual . . . 39
16 Training loss, simple dense . . . 62

List of Tables

1 Descriptive statistics . . . 22
2 Simple Dense Neural Network Structure . . . 28
3 Test Results . . . 29
4 Overview of Deep Dense Neural Networks . . . 30
5 Overview of Parameters Chosen at Random . . . 31
6 Validation and Test Results . . . 31
7 Dense Neural Network Structure . . . 32
8 Validation and Test Results . . . 33
9 Tuning parameters, single layer GRU . . . 36
10 Best model, single layer GRU . . . 36
11 Tuning parameters, stacked GRU . . . 38
12 Best model, stacked GRU . . . 39
13 Descriptive statistics, stacked GRU . . . 40
14 Model comparison - Deep learning . . . 41
15 Tuning Strategy – Random Forest . . . 45
16 Tuning Strategy – Gradient Boosting Machine . . . 47
17 Tuning Strategy – Neural Network . . . 48
18 Confusion Matrix – Random Forest, Gradient Boosting Machine and Neural Network 49
19 Validation Results – Random Forest, Gradient Boosting Machine and Neural Network 49
20 Confusion Matrix – Random Forest, Gradient Boosting Machine and Neural Network 50
21 Test Results – Random Forest, Gradient Boosting Machine and Neural Network . . 50
22 Summary - Weights . . . 51
23 Confusion Matrix – Model Ensembles . . . 51
24 Validation Results – Model Ensembles . . . 52
25 Confusion Matrix – Model Ensembles . . . 52
26 Test Results – Model Ensembles . . . 52

1 Introduction

Cryptocurrency has raised a lot of attention and concerns among investors and scholars. While one side debates on the market being a speculative bubble, the other side advocates for its legitimacy. As of March 31st 2019, the aggregated market capitalization of the two major cryptocurrencies, **Bitcoin** and **Ethereum**, was $87.24 billion (Barchart, 2019). **Bitcoin**, the market leader, made up for 82.84% of this sum. Although, these numbers illustrate the prominent position of **Bitcoin**, they do not capture the volatile nature of the measure. Throughout 2017 and up until 21st of December 2017, **Bitcoin**'s value exponentially increased from around $900 to approximately $20.000, translating into an increase of about 2122% (Higgins, 2017). By the 8th of February 2018, the value had fallen again by over 50%, which accentuates the uncertainty associated with the crypto market.

Peetz & Mall (2017) identify cryptocurrencies as high-risk speculative assets, i.e. they behave similarly to high-risk stocks rather than fiat currencies. This makes it difficult to predict the value fluctuations of cryptocurrencies. Moreover, Bartos (2015) performed regression analysis on cryptocurrencies and concluded that the crypto market follows the Efficient Market Hypothesis (EMH). The EMH implies that no one can generate more than the market return over time since financial markets are informationally efficient, and investors are assumed to be rational (Bartos, 2015). Therefore, the prices change randomly as investors act on new information rather than present/past prices. Nevertheless, findings from standard statistical models such as regression models, could potentially be challenged and outperformed by machine learning models. With the transparent constellation of the crypto market, there is an abundance of data available to perform more complex analysis using these progressive techniques.

In this report, we try to address the problem of predictability of cryptocurrencies. Our aim is to construct models based on machine learning techniques to predict the close price of **Bitcoin**. In addition to this, we experiment with machine learning techniques to predict the direction of movements in the **Bitcoin** price. The following section of this report will provide a detailed problem description.

2 Problem Description

As aforementioned, this report intends to predict the close price of **Bitcoin** and its movement. With the first research question we intend to address how well we can predict the close price of **Bitcoin** on a 24-hour basis using machine learning techniques. The second question is how well we can predict the direction of the movements in the **Bitcoin** close price on a 24-hour basis using machine learning techniques. Our analysis features in total three deep learning techniques and two tree-based models. Specifically we applied, a **Simple Dense Neural Network**, **Deep Dense Neural Network**, and **Gated Recurrent Unit Network**, in addition to **Gradient Boosting Machine** and **Random Forest**. The respective techniques and the construction of the models will be detailed in the subsequent sections. We try to answer our first question by using a continuous predictor, and we therefore face a regression problem. In addition, we will address a binary classification problem and thereby predict bitcoin price movements instead of the exact spot value. The first part of this paper will solely address the regression problem, while the binary problem will be addressed separately towards the end of this paper. The following section will discuss the construction of a benchmark for the regression problem.

Developing a benchmark can be beneficial when evaluating the efficiency and effectiveness of the techniques applied in this paper. The benchmark could as such provide a baseline that we will have to be beat in order to prove that more advanced machine learning techniques are useful. By doing so, you are better able to evaluate whether or not a potential improvement in results, compared to your other techniques, are worth the extra time and energy spent. We have created a benchmark using a guessing approach, where we assume that the current close price will be equal to the price in 24 hours. Subsequently, we compared our guessed predictions to the actual values in time.

Our benchmark resulted in a Mean Absolute Error (mae) of 159,54, a Mean Absolute Percentage Error (mape) of 2.50% and the values depicted in figure 1. We note that the predicted values that followed the actual prices with a delay of 1440 days.

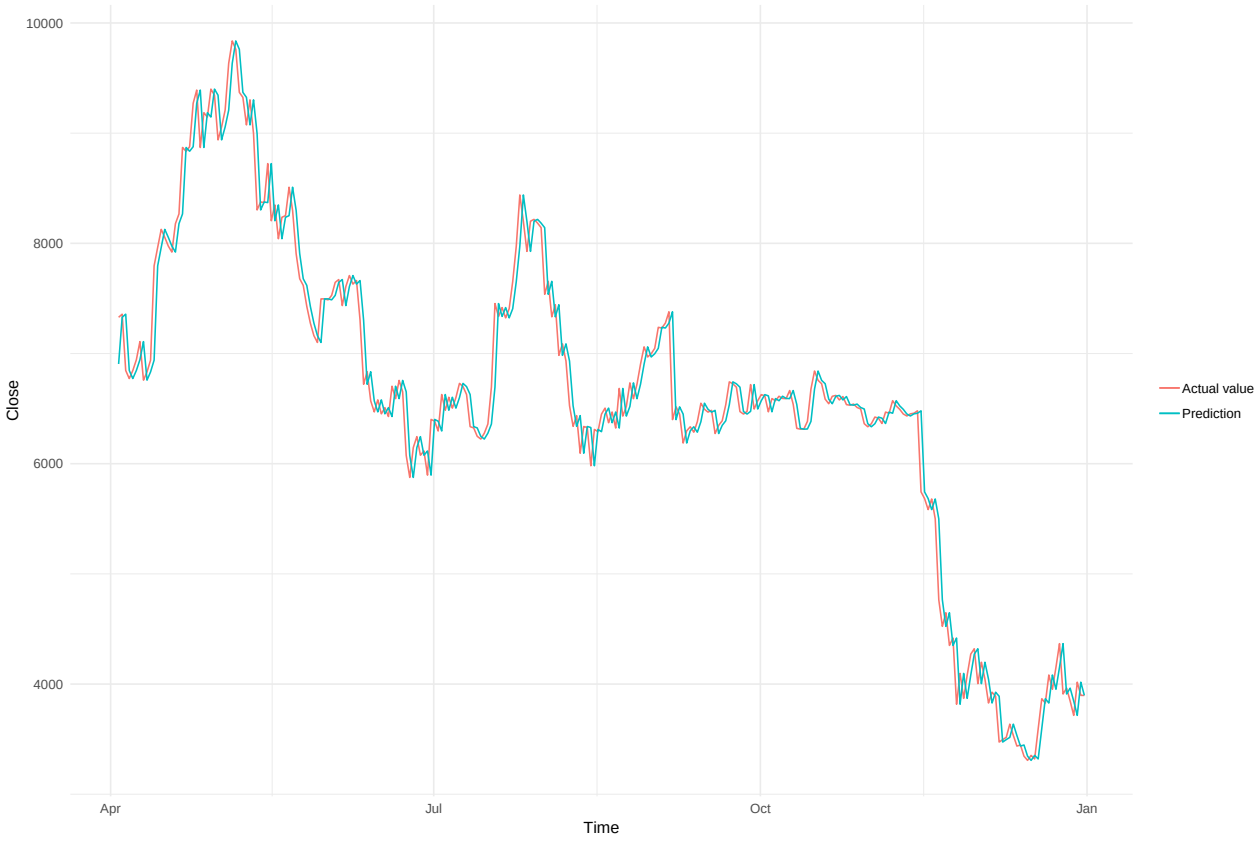


Figure 1: Benchmark: Movement of actual and predicted values

By graphically comparing our best deep learning models' predictions to the actual **Bitcoin** price, we can evaluate the performance of these models in following movements. The mae and mape comparison, will serve as further quality measures for our predictions. The closer to 0 the mae and mape is, the better. Finding a model that follows the **Bitcoin** movement better than our "naive benchmark" and whose mae and mape are lower, would suggest that the use of deep learning techniques to predict **Bitcoin** was worthwhile and improved predictability.

3 Introduction to Neural Networks and Deep Learning Techniques

In this project we have made the assumption that the reader has previous knowledge of what deep learning and neural network are. As a result, we will only give a brief overview on deep learning, how it works, and its history, before we take a closer look at the specific types of deep learning techniques we have used for this assignment.

3.1 History

Deep learning can be described as an intelligent algorithm that is capable of solving very complex problems, which just a decade ago were considered infeasible. It is a subset of Artificial Intelligence (AI), a general term that covers any technique which performs intellectual tasks, normally performed by humans (Chollet & Allaire, 2018). In the early days of AI, researchers believed that human level AI only could be achieved by having programmers "hard code" rules and constraints that could manipulate knowledge (Chollet & Allaire, 2018). However, this opinion has now changed quite drastically. With increasing availability of "big data" and faster hardware, Machine Learning (ML), as a subset of AI, grew rapidly in the 1990s. ML holds the ability to learn representations directly from data without explicitly being programmed to do so (Amini, 2018). With the growth of computing power, ML has become more and more prominent. By using ML-techniques, such as Deep Learning, we can now handle large complex data sets, with millions of data points.

3.2 Neural Networks

A neural network is built by stacking layers of neurons, or nodes, on top of each other. These neurons can be described as computational units that combine a set of input signals, and give it a weight that either amplifies or dampens the importance of the information, based on what the algorithm is trying to solve. These weights are adjusted with respect to how helpful the input can be to make predictions with the lowest possible error. After weighting the input, the information is summed, and passed through an activation function which decides how this input should be processed through the rest of the network, and with that, how much it should affect the final outcome (Skymind, n.d.).

How these nodes are connected, and how information flows through the network can vary quite a

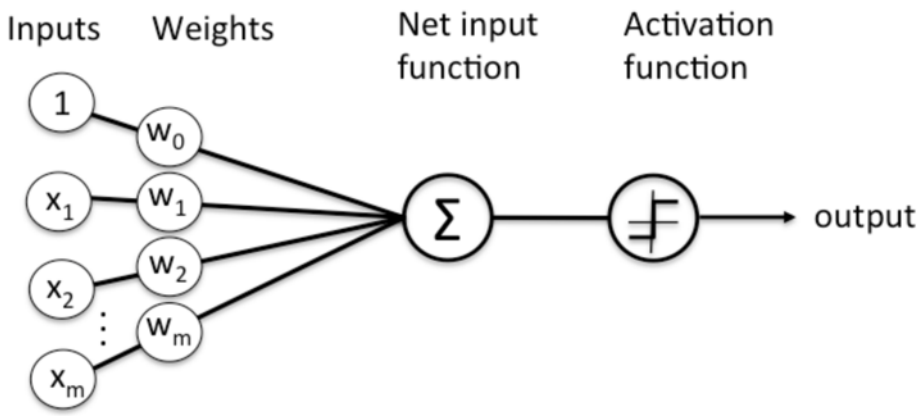


Figure 2: Depiction of a node. Reprinted from Skymind (n.d.)

lot depending on what type of neural network you apply. However, Deep Feedforward Networks, also called Multilayer Perceptrons, are considered the foundation of most deep learning models used today (Upadhyay, 2019). It is called feedforward network because the information flows through the network in one direction. Input **x** is fed through the network in a forward direction, where calculations, are done in each node, for every hidden layer, to calculate an output **y**. The model then applies an optimization algorithm to calculate the error between our prediction and the actual target value. It uses a gradient learning approach through a process called backpropagation, where the model tries to minimize the error rate by updating the weights associated with each node in the network. While, the foundational structure can be found in most deep learning models, it would be important at this point to get more insight into the applied models and their structural differences.

3.2.1 Simple Dense Neural Networks

The first deep learning technique that this paper applies is a Simple Dense Neural Network. As can be seen in Figure 3, the network is essentially composed of one single hidden layer, where the input and output are connected through a layer of weights.

Single layer networks can only handle simple problems that show linearly separable patterns. A simple classification problem for instance, should work well with a single layer network (Brownlee, 2018b). Thus, by building a simple dense neural network to make a complex prediction, such as the closing price of Bitcoin, we expect to see poor results. However, it is useful to create another baseline using a cheap machine-learning method, before going into more computationally expensive techniques, such as **Dense Neural Networks** and **Gated Recurrent Units**. This way we can

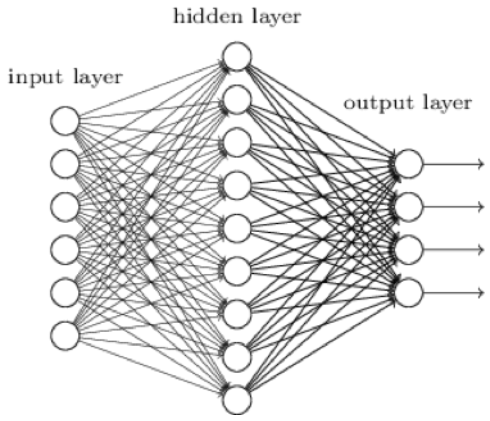


Figure 3: Simple dense neural network. Reprinted from Nielson 2015

see if the more complex model is worth the extra effort and find out if they actually provide useful results.

3.3 Dense Neural Networks

Illustrated in Figure 4, we have a dense neural network which consists of one input-layer, a given number of hidden-layers and an output-layer, where all nodes are densely connected to each other. Each node in the network receives an input from all nodes in the previous layer. This way, all the "neurons" are directly or indirectly connected to the output by a weight (Rampurawala, 2019).

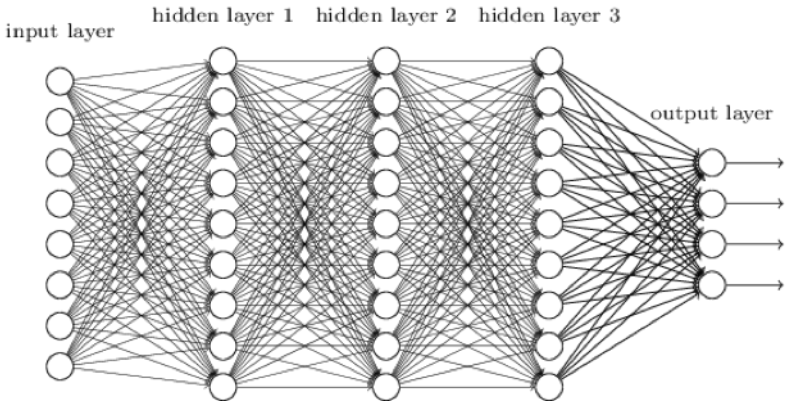


Figure 4: Deep neural network. Reprinted from Nielson 2015

Unlike a **Simple Dense Neural Network**, these networks can handle complex problem sets. Multiple layers increase the possibility of tapping into non-linearly separable patterns in the data (Brownlee, 2018b; Upadhyay, 2019). Moreover, these kind of networks are not too computationally expensive, which makes it possible to try various parameters within a reasonable amount of

time. As aforementionned, a feedforward model, feeds information only forward, meaning that it processes every information once and thereby, past weights are not revised based on new information (Pererveńko, 2016). This is potentially not ideal for time series data, such as our data sets. Therefore, the next section will discuss recurrent neural networks, its ability to handle timeseries data and its drawbacks.

3.4 Recurrent Neural Network

As can be seen in Figure 5, the Recurrent Neural Network (RNN) is different from the previously described feedforward network, in the sense that it iterates through the sequence elements while simultaneously maintaining a state that contains information relative to the previous information (Chollet & Allaire, 2018). In other words, it takes into consideration the earlier inputs, meaning it has two inputs, both the present and the past (Donges, 2018). The fact that RNNs are able to incorporate previous inputs also means that they have a high potential in time series forecasting. Since RNNs are able to input and output sequences of vectors and keep track of previous inputs, they can detect patterns in the input sequence, which makes them ultimately very good at showing a trend (Weller, 2018).

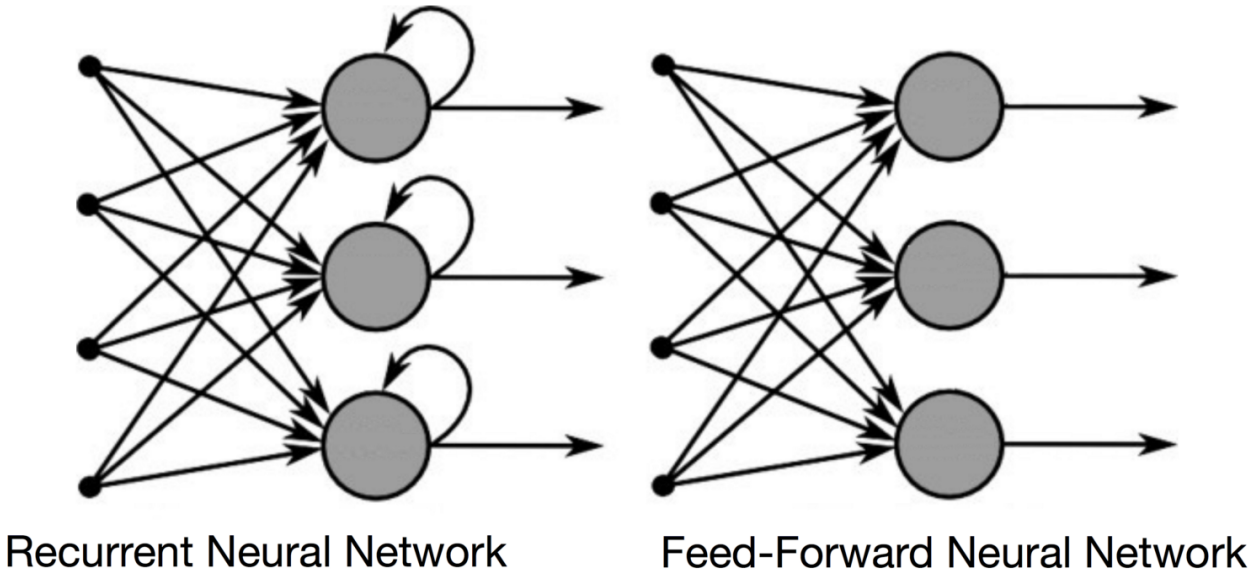


Figure 5: Recurrent Neural Network feedback loop

Due to the fact that both past and present inputs are taken into consideration, RNNs require a lot

more processing power than a feedforward network. Another problem that is more prevalent when working with RNNs is the problem of vanishing/exploding gradients. The problem occurs when the gradient network ends up with many similar numbers, hence the gradient value becomes extremely small or extremely large, therefore either not contributing to- or resulting in very large updates to model's weights (Brownlee, 2017). If this happens, RNNs may not be able to carry information from the earlier time steps to the later ones, which may result in important information from the earlier steps being lost. Since this is a large problem when working with RNNs, RNNs are characterized as suffering from short-term memory.

To resolve the issue of short-term memory, Long Short-Term Memory (LSTM) and Gated Recurrent Unit GRU were created. By using mechanisms called gates, a neural network that regulates the flow of information through the sequence chain and thereby allowing past information to be reinjected at a later time, LSTM and GRU are able to mitigate the problem of short-term memory (Nguyen, 2018). In our project, we have used the GRU network, since this is somewhat more streamlined, making it less computationally expensive than the LSTM network. Ideally, we would run both, LSTM and GRU, and thereby decide. However, since this would require a lot of processing power, it was not possible, given our time- and computational constraint.

3.4.1 GRU

A GRU network works in a lot ways similarly to LSTM but has a smaller amount of operations. Whereas LSTM has three gates; **forget gate**, **input gate** and **output gate**, GRU only has two; a **reset gate** and an **update gate**. The **update gate** decides what and how much past information should be passed on to the future and what new information should be added. The **reset gate** on the other hand, decides how much of the past information should be forgotten (Kostadinov, 2017).

Figure 6 shows a detailed explanation of a single unit in a GRU network. The first sigmoid function from the left illustrates the **update gate**. The update gate is calculated using the following equation:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{1}$$

The calculation is specifying that x_t is multiplied by its own weight $W(z)$, while the information

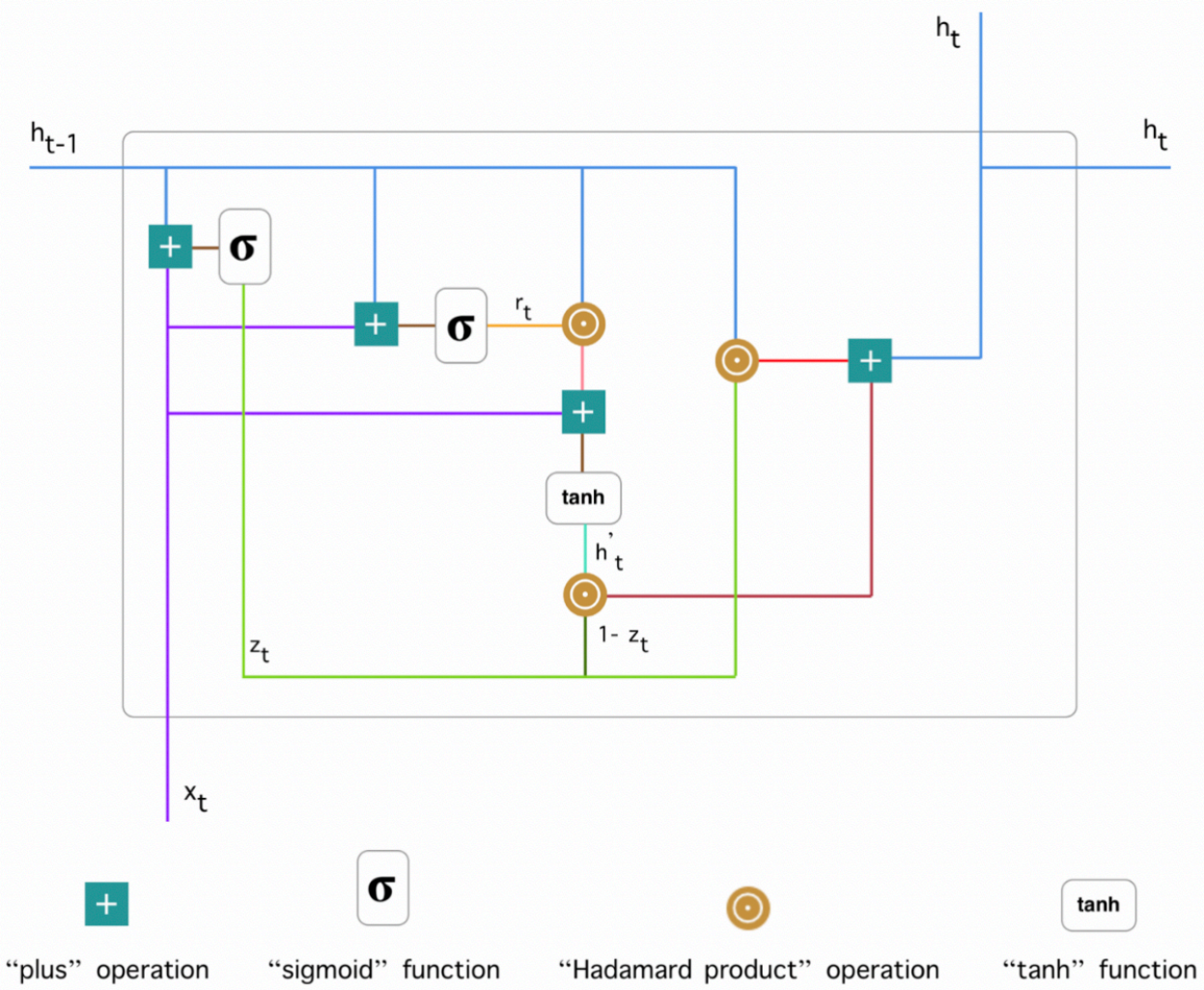


Figure 6: Single GRU unit

from the past (h_{t-1}) is multiplied by the weight U(z). The results of these multiplications are then added together before a sigmoid activation function is applied. This compresses the results between 0 and 1 (Kostadinov, 2017). The next sigmoid function from the left in Figure 6 illustrates the **reset gate**. The calculation of the reset gate is similar to the calculation of the update gate; however, the weights are different, leading to a different usage (Kostadinov, 2017).

$$r_t = \sigma(W^{(r)} + U^{(r)} h_{t-1}) \tag{2}$$

Next, the GRU unit calculates the content of the current memory. Here, the input (x_t) is multiplied by its weight, before it is added to the Hadamard product of the **reset gate** (r_t), the multiplication

of the past information (h_{t-1}) and its weight U, before a nonlinear activation function, $tanh$, is applied (Kostadinov, 2017). This can be seen in the following function:

$$h_t^{'} = tanh(Wx_t + r_t \circ Uh_{t-1}) \tag{3}$$

Lastly, the final memory is calculated by applying an element-wise multiplication of the update gate z_t, the information from the previous unit h_{t-1}, the same element-wise multiplication to $(1 - z_t)$ and the current memory content ($h_t^{'}$). The sum of this is the final memory at the current time step (Kostadinov, 2017). This process can be seen in the function:

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ h_t^{'} \tag{4}$$

As can be seen in Figure 6, the calculations of a single unit in a GRU network is far more extensive than the previously discussed feedforward dense networks, making the process of running a GRU model far more computationally extensive. This notion, as will be discussed in more detail later, meant that the implementation of RNNs, and specially a GRU network, had some limitations in our project.

In summary, GRUs are able to filter the information, and store the important parts, using their update and reset gates. Due to this, the vanishing/exploding gradient problem is eliminated since the model is keeping relevant information and passing it to the next time step, rather than washing out the new input every single time (Kostadinov, 2017).

4 Tuning the Neural Networks

For machine learning in general, finding the optimal hyperparameters can be a very difficult and time-consuming process. When applying deep learning techniques to predict the price of Bitcoin, on a 24-hour basis, we need to make a lot of parameter decisions that will be essential for the model outcome. Examples of such decisions can be: architecture of the network, should dropout be introduced, what learning rate to pick and so on. If we give the neural network wrong or misleading parameters, the training may take forever to complete, or it will never reach its local minimum at all (Maladkar, 2018). We can divide the hyperparameters we need to pick when constructing a neural network into two categories: **consistent parameters** and **tuning parameters**.

4.1 Consistent parameters

Some of the parameters we need to pick, will stay consistent throughout the whole training process and are selected based on the specific problem we are trying to solve. To build a functioning model, there are two key parameters that need to reflect our regression problem, namely the activation function and the loss function. These two hyperparameters will be outlined in the following sections.

4.1.1 Activation funciton

The Activation function is applied to the sum of the weighted inputs, such that the inputs are translated into a non-linearly shaped output, that is then fed to the subsequent layer. Without the activation function, the output would be a linear polynomial in the form of a sum of weighted inputs. In other words, we would treat the assignment as a linear regression problem. This simplicity, would not allow the models to learn complex non-linear relationships from our data. With our large data sets and the commonly known challenge of predictability of the underlying asset, we need to enable our models to extract possible non-linear patterns from the data. Therefore, for the given regression problem, we have chosen **ReLu** (Rectified Linear units) as activation function and applied it to our hidden layers. One reason for choosing ReLu is that, it evades and rectifies the formerly elaborated vanishing and exploding gradient problem (Walia, 2017). A drawback of ReLu is that, due to its gradients being either 0 or 1, it might lead to dead neurons. When it returns a gradient of 0, then that node will never activate again, i.e. be “dead” (Walia, 2017). With regards to its advantages

and disadvantages, it is still the most popular activation function and will therefore be used for the hidden layers in our models. The last-layer activation function will control how the model output should look. For our regression problem we would not like to force the model-output to a specific range but rather let it predict freely. Therefore, the last layer is left without an activation function (Chollet & Allaire, 2018).

4.1.2 Loss function

The loss function has to be chosen in accordance to the nature of the underlying problem. We chose "mse", with added metrics "mae" and "mape" for comparison. **Mean Squared Error**, or mse is the default *loss function* for regression problems. This function squares the predictive errors and thereby penalizes the large errors more (Brownlee, 2019b). **Mean Absolute Error** (mae), gives less weight to outliers which deviate strongly from the mean, by calculating the average absolute difference between actual and predicted values (Brownlee, 2019b). **Mean Absolute Percentage Error** (mape) is the mae expressed in percentages. *Mape* is usually used to increase interpretability of results (Pascoal, 2019).

4.2 Tuning strategy

Tuning the model to minimize the error rate is a challenging and time-consuming task, even for professional machine learning engineers. Even though it is possible to gain some tuning-skills through experience, the result is very likely to be suboptimal, if the chosen parameters are based solely on intuition (Chollet & Allaire, 2018). Ideally, the optimal procedure would be to perform a grid search, where every combination of several hyperparameters is evaluated, in order to find the best model (Gozzoli, 2018). Considering the limited time- and computing power available for this project, we used the random search approach through **automatic hyperparameter optimization** as our main tuning strategy. Here, combinations of hyperparameters are not picked systematically but randomly. We herefore used the package **tfruns**. This package automatically creates a unique folder containing all information from each run. It also provides many useful functions for comparison of previous training-runs and its results. In addition, it makes it easier to fine tune the model's hyperparameters using something called flags. By specifying flags for each of the tuning parameters we would like to tune, we could easily change and introduce lists of different combinations of values

for each of the parameters without changing our code (J. Allaire, n.d.). It should be mentioned that we did about 10-15 tuning runs for each of the data sets (regular, PCA and autoencoder). This is likely to be an insufficient amount of tuning runs and our results will possibly not represent the local- nor global-optimum. The parameters we tuned using tfrun package were:

1) Layers
2) Units
3) Dropout rate
4) Regularizers and lambda values
5) Optimizers and Learning Rate
6) Batchsize
7) Epochs

4.2.1 Layers

As aforementioned, the number of layers is crucial depending on the complexity of the problem. As elaborated with *Simple Dense Neural Network*, one hidden layer would not pick up non-linear patterns (Brownlee, 2018b). Hence, depending on the data at hand, the model structure should comprise more or less hidden layers. The deeper the model is structered, i.e. the more additional hidden layers are added to the model structure, the more complex non-linear patterns could be picked up (Brownlee, 2018b; Upadhyay, 2019). As for every hyperparameter, the goal is to optimize the model's performance. Therefore, the depth of each model will be decided based on model performance.

4.2.2 Units

As formerly elaborated, each neuron in the network receives a weighted input from all neurons in the previous layer. This number of neurons is defined with the number of units in each layer. A wider layer, i.e. a layer with more units than the input layer, can potentially unveil complex relationships from the input. On the contrary, a narrower layer than the input layer, would try to compress the input by eliminating unnecessary information (Hubens, 2018). Therefore, the number of units is important when trying to adress the underlying problem.

4.2.3 Dropout rate

With a deep and wide structure, a model can potentially train "too well" on the input data by introducing random variability, i.e. learning meaningless patterns (Goldacre, 2011). This is referred to as overfitting, which leads to a poor model performance on unseen data. The dropout rate is a hyperparameter that tries to prevent overfitting by regulating the width of a layer, and thereby controlling statistical noise. Here, the model randomly "drops out" nodes, which changes the number of inputs that each node in the next layer is fed. As subsequent nodes learn from the input of previous nodes, dropping out nodes possibly interrupts the introduction of noisy patterns (Brownlee, 2018a).

4.2.4 Regularizers and lamba values

As aforementionned, complex models are at risk of overfitting if not regulated. Besides defining dropouts, the models can generalize better when regularizers are introduced. Regularizers add a "regularization term" to the defined loss function, thereby penalizing the assigned weights. The logic is that smaller weights lead to simpler models and decrease overfitting (Jain, 2018). The strength of the regularization effect can be adjusted by assigning a specific lambda value. If lambda is high then the model can underfit, and when its value is too low, it can overfit (Developers, 2019).

4.2.5 Optimizers and Learning Rate

Optimizers define in which way the model will be updated, i.e. it links the loss function to the chosen parameters. Due to limitations in regards to computing power and time, we were not able use an extensive experimental approach to find the best performing optimizer. Therefore, this paper will try two optimizers, namely Adam and RMSProp. Adam, or Adoptive Moment Estimation, is considered the state-of-the-art by many machine learning professionals (Ruder, 2016). Further, RMSProp, or root mean square error, was selected as it appears to be a versatile optimizer that can be applied to various problems (Chollet & Allaire, 2018). For the optimizer a learning rate has to be defined. The learning rate sets the learning speed of the model. When the learning rate is high, then the model will update less frequently, and vice-versa. Finding a balance is important, given the underlying problem to solve, the time and computational limitations (Brownlee, 2019c). Throughout the training process, the chosen optimizers adapt these learning rates for each weight

individually. In this regard, Adam adjusts the learning rate according to variance and magnitude of changes in weight, while RMSProp considers only the latter (Brownlee, 2019a).

4.2.6 Batchsize

The batchsize defines how many samples a model should go over until it updates its weights. After every batch, the model will evaluate how well it did in predicting the actual values, and according to the resulting error, adjust the weights. Consequently, the higher or lower the batchsize is defined, the less or more often the model weights will be updated (Brownlee, 2018c).

4.2.7 Epochs

The number of epochs will define how many times the whole data set will be run through. Ideally, it should be set such that the model can decrease the predictive error with every additional epoch. A high number of epochs can thereby potentially allow the error rate to decrease substantially, however it is quite time-consuming with a large data set and may lead to overfitting (Brownlee, 2018c). With the given time limit, we have to experiment with a number of epochs that is high enough to capture model improvements but low enough to save computational expenses. Additionally, the actual number of runs can be reduced through early stopping. Early stopping is usually set to a fraction of the predefined epochs. When running the models, it prevents the execution of all epochs, if the error could not be improved over the predefined fraction of runs.

5 Data sets

5.1 Original data set

We started off by looking at the data set provided in class, consisting of different metrics for 2071 unique cryptocurrencies. The metrics for the different currencies were logged once a day, but due to differences in the lifespan of the currencies, the number of observations varied significantly. The most observations available for a single cryptocurrency was for **Bitcoin**, with observations dating as far back as 28.04.2013. Other variables in the data set, besides the date, included an observed value of the opening price, closing price, the highest price and the lowest price for each available day.

As we investigated the data set further, we saw that there were some issues with the data set. By looking at the correlation matrix in Appendix A, a correlation matrix of all coins with more than 2000 observations, in addition to the cryptocurrency **Ethereum**, we saw that for the most part, the different currencies were highly correlated. This is natural, considering the fact that a lot of the smaller coins in the data set can only be exchanged with a few big coins, meaning they are all priced relative to the larger coins. Further, we noted that a large number of coins have a limited number of observations, making the probability of making a good model, less likely. Even if we were to exclude the coins with fewer observations, we would at the most be left with a data set consisting of 2.042 observations.

5.2 New data set

Since no model can overcome a severe lack of data (Chollet & Allaire, 2018), we chose to introduce a new data set with minute data for **Bitcoin** in the time period 2016 to 2018. The data set was downloaded from GitHub and contained the time/open/close/high/low/volume from Bitfinex, one of the largest cryptocurrency exchanges available. The new data set was significantly larger than the original data set. Further, we noticed that there were some minutes in the data set that were not included and we therefore extrapolated these minutes with the last known observation, in order to better be able to compare the observed values in the different days. This might lead to somewhat distorted predictions, but since only a limited number of values were being extrapolated, we argue that it most likely did not affect the models predicting power significantly.

5.2.1 Further discussion

There is a fine line between too much- and too little data. We are aware of the fact that a data set with minute data might include noise that could overwhelm the signal and thereby making it difficult for the model to make predictions. However, since deep learning models in general make better predictions with more data included and the alternative is making predictions with less data, we argue that proceeding with the minute data is the best alternative. We also note that this data is downloaded from a less secure source and therefore may not be as reliable as if we were to download the data set directly from the source. Since the Bitfinex API limits the number of requests per minute and the number of records per request, downloading the data directly from the source would take hours and thousands of request, making it highly impractical. Due to this, we proceeded with the data set from GitHub. To mitigate the risk of severe mistakes in the data set, we have compared the observations in our data set to the daily price of **Bitcoin**, found on Bitfinex. As can be seen in Figure 7, the two plots follow the same trend and we therefore conclude that the data set from GitHub is fairly accurate.

Figure 7: Price of Bitcoin from new data set

5.2.2 Variables

Since past performance generally is a bad predictor of future prices (Chollet & Allaire, 2018), we have further included variables that encompass more information about the underlying asset. The respective variables are explained in the following sections.

5.2.2.1 Feature Engineering

Feature engineering is the art of transforming available historical data into new features, which could improve the performance of a model (Datarobot, 2019). Finding patterns in, for example, volatile data might be diffult for a model. With the current minute to minute prices, our models might react extremely to changes. Instead, this volatility could be reduced by computing new variables from the available data. If well executed, creating new variables from existing features, can provide different insight into the data and thereby support the model in finding patterns. However, feature engineering is quite complex and requires expert knowledge for the analyzed domain (Datarobot, 2019). Unless, the investigated problem is the same, it will not be possible to apply it to different problems. Hence, the process is quite time consuming, as it required each feature to be manually constructed. Moreover, manual coding of features is likely to entail mistakes (Koehrsen, 2018).

Given these limitations and risks, we have decided to create features that are the "status quo" for the type of problem at hand. Accordingly, as cryptocurrencies act similarly to stock (Peetz & Mall, 2017), we decided to include features that are commonly used to measure and observe trends for stock data. The approach was applied to the open, high and low prices of Bitcoin. Specifically, we created *returns*, *moving average*, *moving minimum* and *moving maximum*, as suggested by Gray (2018). These are all features that express the trend of price movements rather than price spot values in time. Given Bitcoin's volatile nature, these features could provide smoothed data that represents trends rather than minute to minute fluctations. In the bigger picture, exceptionally high fluctuations would therefore not get as much weight unless they occur frequently. With this motivation in mind, we proceeded with engineering the new variables.

The *return* was expressed as a 24-hours percentage change in prices. For example, the open return in observation 1441, was expressed as the percentage change of the open price from minute 2 until minute 1441. The *moving average*, *minimum* and *maximum*, were calculated with the arithmetic average of a number of observations contained in a specified time-span. Given Bitcoin volatile

nature, we created 24 hours moving average, minimum and maximum and added a 8 weeks moving average. The motivation behind adding a moving average for a longer timespan, was to tackle the previously mentioned extreme volatility of Bitcoin in 2017, where the price dropped by over 50% in approximately eight weeks. An eight weeks long moving average would smooth these extreme increases and decreases, and potentially allow the model to react to "long-term" trends and not temporary extreme peaks.

5.2.2.2 Additional Variables

To potentialy add further valuable information to our datset, we turned towards external market specific information, using the **Quandl** package. Quandl is a company that is specialized in providing analyzable data sets. The **Quandl R** package allows to tap into the Quandl API and extract data. Considering, that the theories about the drivers of Bitcoin are as speculative as the cryptocurrency itself, we have decided to consider some of these theories when choosing our variables. After extensively following various online discussions and researching the functioning of Bitcoin, we retrieved 9 variables from the BCHAIN Data set, containing information on the Bitcoin Blockchain. We extracted:

1. Average daily block size
2. Cost per transaction
3. Transaction cost as a percentage of the transaction volume
4. Difficulty
5. Hashrate
6. Bitcoin Total Transaction Fees in USD
7. Miners revenue per transaction
8. Bitcoin Number of Transactions Excluding Popular Addresses
9. Bitcoin Number of Unique Bitcoin Addresses Used

A problem with the data set is that our bitcoin prices are recorded by minute while the added features are recorded daily, meaning that they will be reprinted 1440 times when merging the data sets. It is also important to note that since the daily data is retyped for each minute of the day we are introduced future bias, as each observation represents the value for the whole day. We are aware that this bias might affect our predictions and we thereby lagged the observations in order to address this issue. In the following sections, we will make sense out of the chosen variables whilst

explaining the Bitcoin Blockchain.

The Bitcoin Blockchain is a digital protocol of all transactions between parties within the network. The Blockchain got its name from the way its digital records are registered. Here, transaction are not added individually but in blocks. Hence, a desired transaction will not be immediately completed on the Blockcain, but miners have to incorporate these transactions into a block and then add it to the Blockchain (Crosby, Pattanayak, Verma, & Kalyanaraman, 2016). The size of the blocks is limited, which leads us to the first chosen variables, the *average daily block size*, *cost per transaction* and *transaction cost as a percentage of the transaction volume*.

The limited *average daily size of the blocks* could have an indirect impact on Bitcoin Prices through *cost per transaction* and *transaction volume*. Limiting the *block size*, creates a competitive transaction environment, where the highest offer goes first. This leads to increased *cost per transaction* and *higher transaction cost share of transaction volume*. In this sense, it makes competitors coins cheaper and more attractive, thereby potentially decreasing *transaction volume* for Bitcoin (Wolframm, 2019). Recently, the *average blocksize* reached a high (Tamuly, 2019) through the implementation of Segwit, which upgraded the storing format (CoinDesk, 2018). As suggested Tamuly (2019), this implementation was followed by lowered *cost per transaction* and overall more transactions, which can potentially influence the price.

Transaction fees are not only influenced by *block size*, but possiblity also indirectly by the *hashrate*, which is the speed at which a computer is able to complete a block (Tuwiner, 2019). Miners need to find a hash to add a block to the blockchain. The *difficulty* of finding a hash, is modified by the network itself. A higher bitcoin *hashrate* would mean that more Bitcoin miners were mining and so that more transactions have been completed (Taylor, 2018). The block production rate is supposed to be at 10 minutes per block, but when miners collectively complete blocks faster, in other words, when the *hash rate* is increasing, then a block could be completed even faster than 10 minutes. The *difficulty* level is then adjusted automatically, to ensure that the block completion time remains at 10 minutes ("Blockchain," 2018). In other words, when the *difficulty* level is high, the *hashrate* is high as well, meaning that more transactions are completed. This could potentially influence the price.

Building on the previous connections, the *Bitcoin Total Transaction Fees in USD* and the *miners revenue per transaction* were added to the data set. The miners revenue per transaction is the sum of daily earned bitcoins and the associated transaction fees, multiplied by the market price

of bitcoin. Naturally, miners will mine more if it is profitable for them, and as the *block size* is limited, they have an incentive to maximize the *transaction fees* per block. Every completed block provides them with a fixed reward of 12.5 BTC and the associated transaction fees. Deducting the encountered electricity cost gives the net profit or loss for the miners (Tuwiner, 2019). Taylor (2018) raised an interesting point, stating that is difficult to stay profitable for miners, considering the current system. However, they continue to mine, which Taylor (2018) relates back to speculative reasons. Miners get 12.5 BTC for each completed block, and the more transactions are completed, the more potential there is for the Bitcoin price to go up, thereby increasing the value of the 12.5 BTC they get per block.

Adding to the aformentined theories, the *Bitcoin Number of Transactions Excluding Popular Addresses* and the *Bitcoin Number of Unique Bitcoin Addresses Used*, were added to the data set. A report from Chainalysis argues that 30% of the total bitcoin supply is held by long-term investors that only trade when the market price is high. Indeed, from the rise in April 2017 to the fall of Bitcoin in April 2018, the share of coins held by short-term investors increased from 14% to 35%, due to long-term investor cashing out (Chainalysis, 2018). This report thereby links the exponential Bitcoin increase and crash, to the activity of long-term investors (Abeslamidze, 2018). Therefore, we argue that the two last variable could potentially reflect long-term investors trading their coins, as the *number of unique adresses used* reflects increases or decreases in trading activity and the *number of transactions excluding popular addresses* could quantitize that increase or decrease in trading activity excluding most frequently used adresses.

5.2.3 Complete data set

The complete data set has a total of 1.576.801 rows and 30 variables. The response variable was **Close** and we used a combination of variables derived from the currency exchange, including trading volume and prices, and variables that explain factors like mining difficulty, etc. In Table 1 we present the different categories with the corresponding metrics mean, median, minimum value and maximum value for some of the variables in the data set. We can see from Table 1 that there was a lot of variation in the data set. For instance, the values in the **Close** variable were ranging from 353,04 to 19.891.

Table 1: Descriptive statistics

	Close	Volume	avg_block_size	cost_trans	cost_vs_trans	hash	mine_rev	trans_vol
Mean	4.018,32	20,90	0,87	35,80	1,22	14.705.168,68	8.450.213,05	228.486,37
Median	2.535,40	5,06	0,89	21,61	1,08	5.302.595,61	5.521.458,33	228.091,56
Min	353,04	0,00	0,43	3,87	0,28	693.089,36	883.763,26	37.558,21
Max	19.891,00	6.717,52	1,22	161,69	4,76	61.866.255,51	53.191.582,14	715.367,35

5.3 Principal Component Analysis

Running the new data set with 28 features (excluding date and close price) and about 1.6 million observations each, requires a lot of computing power. Naturally, by doing feature engineering and adding additional features to the data set, our goal was to introduce useful information as input to the model to improve its performance. On the other hand, by introducing new information to our model we also risk increasing the noise in our data set. Noise can be described as irrelevant or weak data items which don't help to explain the relationship between the feature and the target variable. Reducing noise can enable the model to train faster, make the input easier to interpret and reduce its complexity, reduce overfitting and improve the accuracy of the model given that the right subset is chosen (Rathi, 2018). With respect to both computing power and potential noise in our data set we wanted to identify these irrelevant and weak data items, and reduce dimensionality.

First, we decided to use *Principal Components Analysis* (PCA) to remove redundant information. By running a PCA-analysis on our data set, we are able to summarize the data in a lower-dimension set of features (Lopes, 2017). Put simply, PCA constructs new data-points using a combination of the old ones. The algorithm tries to fit the data set to a new coordinate system using linear transformation, and places the data items with the most significant variance in the first coordinate, the item with the second most significant variance in the second coordinate, and so on. This way, the first variables become strongly correlated with each other, which makes us able to explain most of the complexity of the data set using a small subset of the data (Hayden, 2018).

As figure 8 shows, we can explain close to 100% of the variance in the data set by using a subset of the 10 first PCA components.

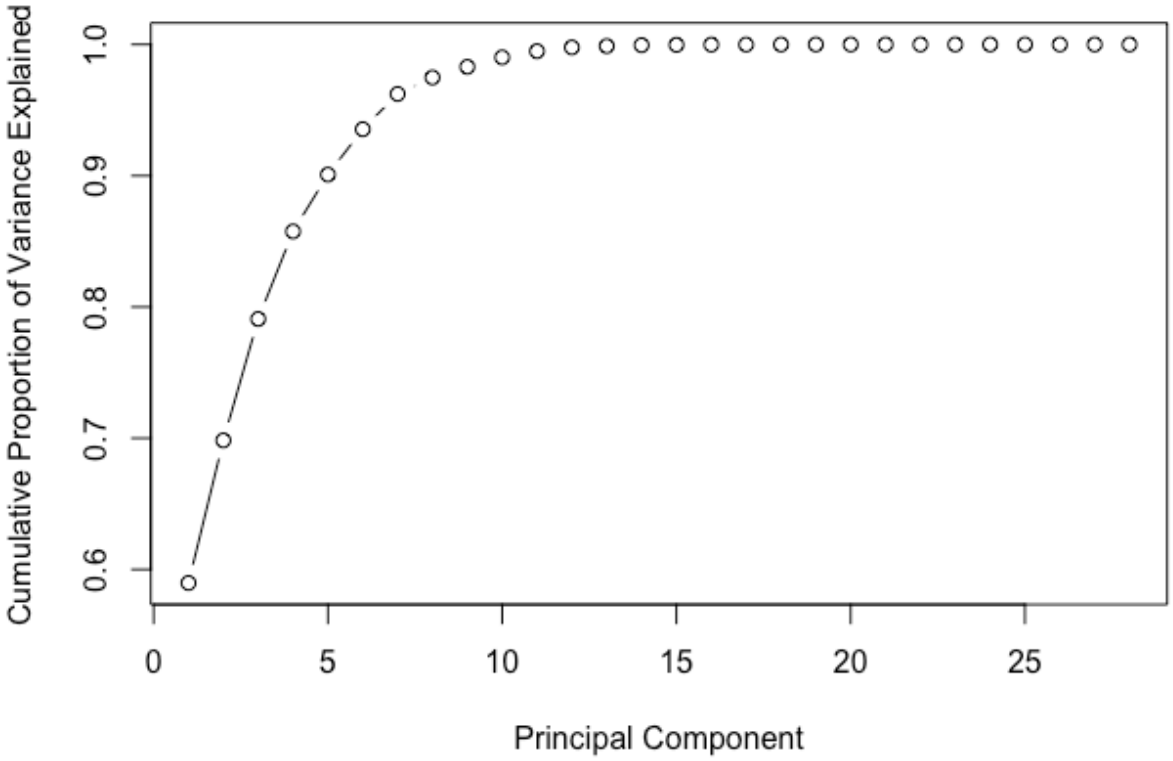


Figure 8: Principal Components Analysis

5.4 Autoencoder

Besides the previously introduced PCA, deep learning methods can be used to perform dimentionality reduction and de-noising. A study added random noise to a data set (in this case, zeros) and tested the deep learning models' capabilities of filtering the noise against other techniques, including PCA. They concluded that the deep learning models outperformed the other techniques when it comes to handling the introduced noise (Fan, Sun, Zhao, Song, & Wang, 2019). Moreover, auto-encoding with deep learning techniques compared to PCA and manual feature engineering, can capture hidden non-linear relationships. Considering the size of our data set deep learning autoencoders seem to be a reasonable choice, as they can handle large data sets very well (Zamparo & Zhang, 2015).

5.4.1 Process of auto-encoding

Auto-encoders, or AE, try to construct an output that is essentially a reconstruction of the input.

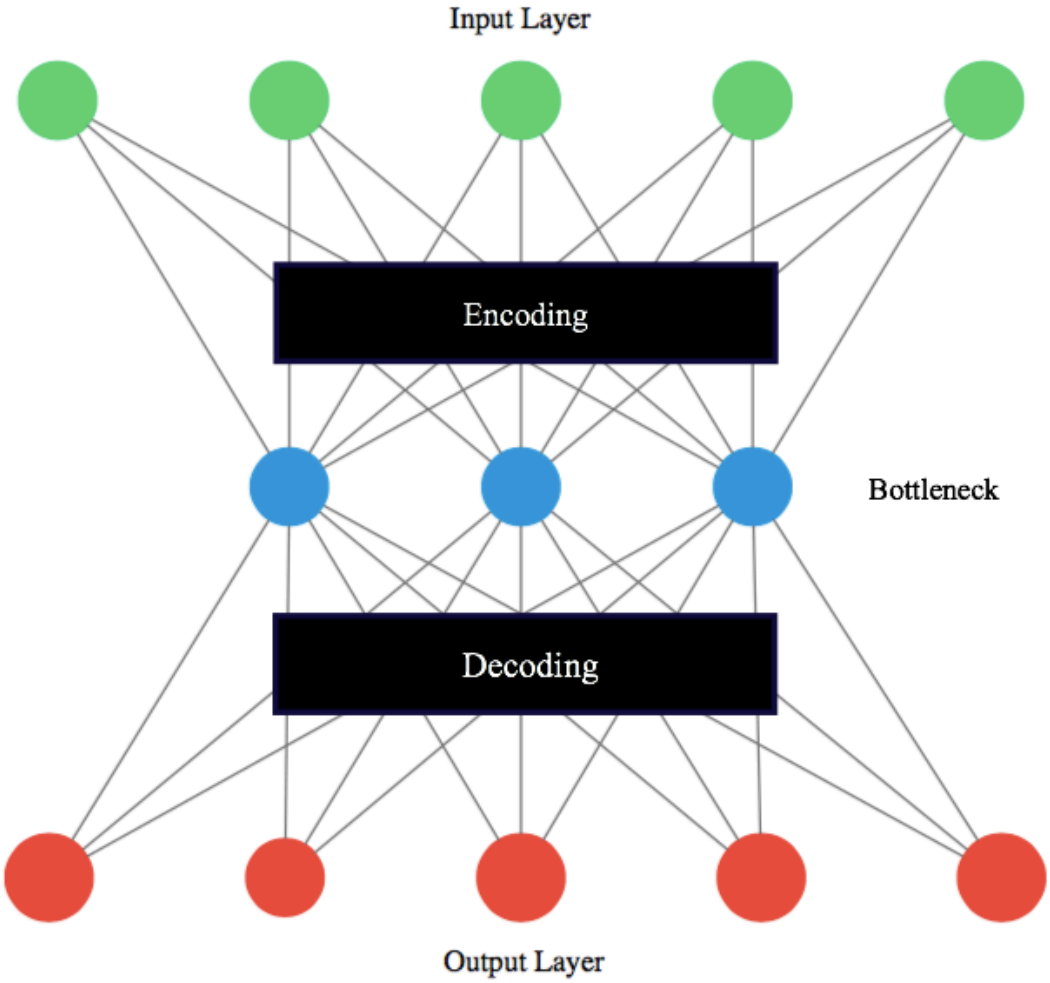


Figure 9: Deep Learning Autoencoder. Adapted from "Applied Deep Learning - Part 3: Autoencoders" by Hubens (2018)

Auto-encoding is initiated with an encoding process. Here, input features are fed into a deep learning model and compressed into a smaller dimension, the "latent representation", also often referred to as the "bottleneck" (Hubens, 2018). This bottleneck is then decoded to reconstruct the output. The goal of this auto-encoding process is not to plainly reproduce the input, but to make use of the bottleneck. By forcing the model to compress information into a smaller dimension, it

will learn the most relevant information. Hence, if the dimension of the bottleneck is set to equal or higher to the input's dimension, then the model will simply copy the input to the output and not learn anything (Hubens, 2018).

5.4.2 Model structure and tuning

Just like for the regular deep learning models, it is important to choose the right models and tune the hyperparameters to the problem at hand. Autoencoders can be used with any deep learning technique, but we chose to use Deep Neural Network and a GRU neural network, to remain consistent throughout the paper. Considering hyperparameters, four choices had to be taken before running the models: the bottleneck size, model depth, unit number for each layer and the loss function (Dertat, 2017). As usual, the tuning of these parameters should ideally be conducted with a grid search. However, due to time limitations and the focus of this paper not being autoencoders, we decided to follow an example by Oehm (2018). Here, the data scientist, Daniel Oehm, first performed a PCA and then used the length of the resulting data set to define the width of the first hidden layer. Then he set the width of the bottleneck to half the length of the PCA data set. In his test runs, the encoded variables were able to pick up on some patterns that PCA did not revealed.

We proceeded with creating the basic structure for both deep learning models, as represented in Figure 10.

In total, the models had three hidden layers. The number of nodes was set to 10 for the first hidden layer, according to the length of the PCA data set. The number of units for the second layer, the bottleneck, was set to 5. Following Daniel Oehm's example of a symmetric autoencoder (Dertat, 2017; Oehm, 2018), the decoding layers were symmetric to the encoding layers. The third layer thereby had a unit number of 10 and the final output layer had the width of the original data (28). In other words, the output was set to be a reconstruction of the input variables.

5.4.3 Results

The first run was conducted on the Deep Neural Network Autoencoder. The model was able to encode the variables and decode them to predict the output with a loss of **0.07865**. This number in itself does not represent a quality measure, if it is not compared to another run where the loss

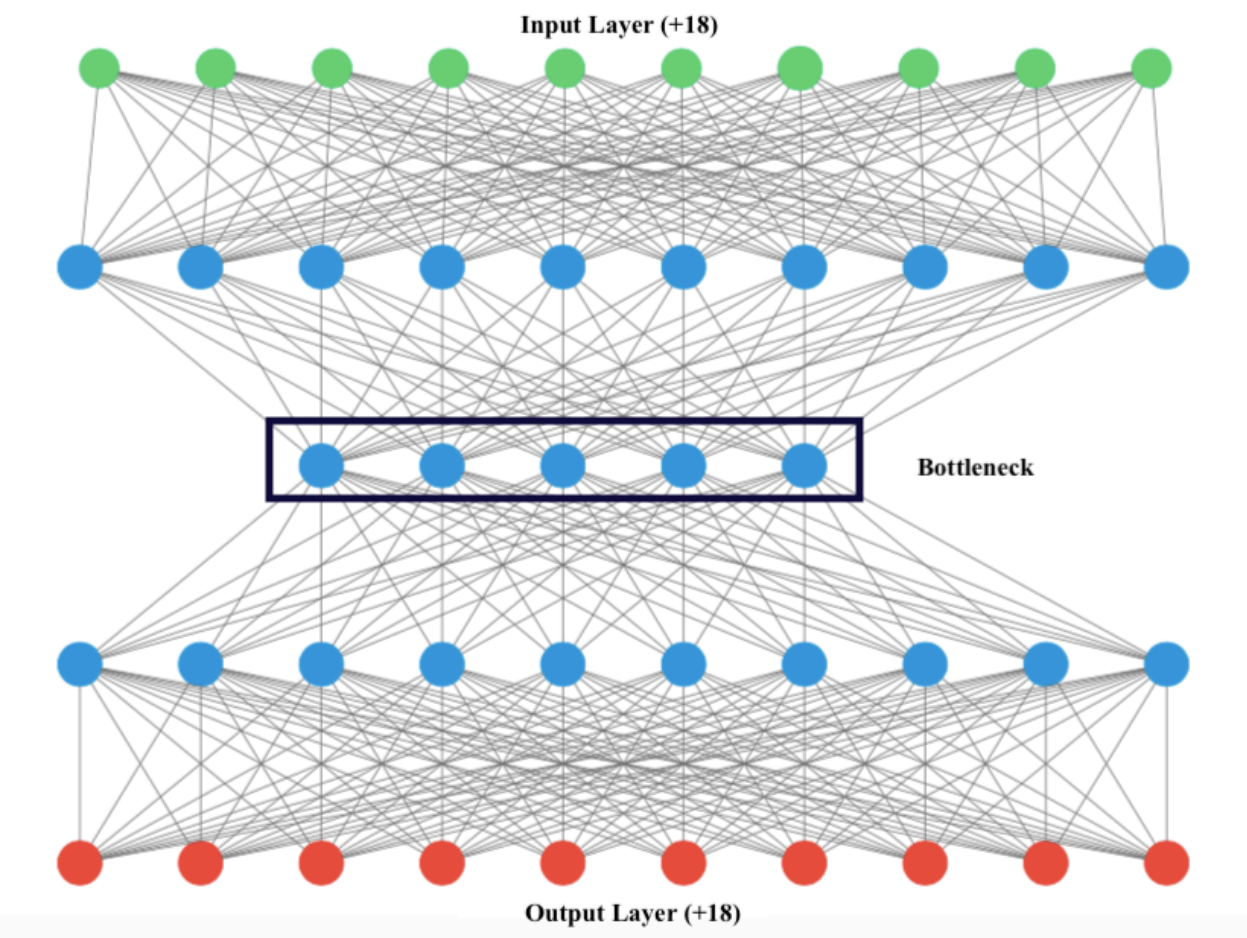


Figure 10: Deep Learning Autoencoder

is higher or lower. As aforementionned, the focus of this paper is not to find the best autoencoder data set for each model, but rather to reduce the dimensions, and noise in the data set, ideally in a better way than PCA. As pre-defined in the second hidden layer, we ended up with a new data set containing five variables. Following the same structural approach, we proceeded with building and running the GRU autoencoder. Although, this turned out to be a rather time-consuming and challenging task, it did lead us to a result. In the following section, we will elaborate on the work process and underlying ideas.

First, we had to think of the shape of the autoencoded data set, which is important for RNN models such as GRU. The model should be able to memorize previous observations. We therefore created a function that would feed the model sequences of input data, here with the difference of predicting the input data again. Hence, the model was supposed to train on the whole data set,

with a validation split of 20%. Moreover, all the variables had to be defined as target variables with the exception of Close Price and Date (excluded at this point). Our function helped us construct a 3D Array in the desired shape, with samples of lagged observations (7 days) for 28 variables.

Running the GRU autoencoder, provided a loss of **0.245** (mae: 0.228), when predicting the output. Hence, the latent representation created with the Deep Neural Network Autoencoder was performing better than the GRU Network Autoencoder. To investigate, whether a GRU autoencoder could still outperform a Deep Neural Network, we tried another run building on the deep dense approach. Here, we reshaped the input data for the Deep Neural Network Autoencoder to a 3D array. Technically, it is a 2D data set with a third dimension of 1. After running it for 24 hours, the model was still at epochs 60 out of 500. Given the time limitation and the focus of this paper, we decided to stop this run, and use the data set resulting from the Deep Neural Network Autoencoder.

5.5 Data Split

All the data sets above were split into two parts consisting of 75% for train and validation, in addition to 25% for the test set. This was done in order to have a sufficiently large data set to train on, while still being able to test the models on a representative portion of the data. Given that the data set consists of observations from 2016 to 2018, where the market for Bitcoin proved to be extremely volatile, this split is likely to affect our model's ability to generalize. Finally, as we are dealing with timeseries data, we did not shuffle the observations. The next chapter will detail the analytical process and try to answer our first research question.

6 Analysis

6.1 Simple Dense Neural Network

6.1.1 Implementation of tuning strategy

To start off, we wanted to build a simple dense neural network. In doing so, we applied R and the package **Keras**. By following the architecture of a simple dense neural network, as explained in the theory section, our baseline network ended up with the structure shown below:

Table 2: Simple Dense Neural Network Structure

Parameter	Values
Input layer - units	28
Hidden layer - units	100
Output layer - units	1
Activation	relu (hidden layer)
Loss	mse
Optimizer	Adam (l = 0.001)
Epochs	200
Batch	200

The model was given scaled inputs from the variables in our training set, including all features from the complete data set, including the added- and feature engineered variables. Briefly explained, our simple dense network consists of one hidden layer with a *ReLu* activiation function, and was compiled using *mse* as the loss function and *Adam* as the optimizer. Further, the model was fitted applying 200 epochs and a batch size of 200.

6.1.2 Results

Subsequent to training and validating our simple dense neural network, we ran our model on so far unseen test data. The test results are illustrated in Table 3, together with test results from the previously discussed benchmark.

As we can see, the simple dense network did not perform very well. The model is on average missing with 48%, compared to 2% for our naive-benchmark. To look further into this, we have plotted the actual closing price against the predicted closing price of Bitcoin. As we can see in In Figure

Table 3: Test Results

	Simple NN	Benchmark
	Test	Test
mae	2.885	159,54
mape	48.00%	2.00%

11, we are constantly overpredicting for all observations in the test set. It seems like we are able to follow the patterns to some extent, but we are always a bit late. This might be because the model is fed delayed information on a continous basis, due to the laggin of the data. We note that the initial training runs indicated that the simple dense model appears to be struggling with picking up patterns from the data. An illustration of this can be found in Appendix B.

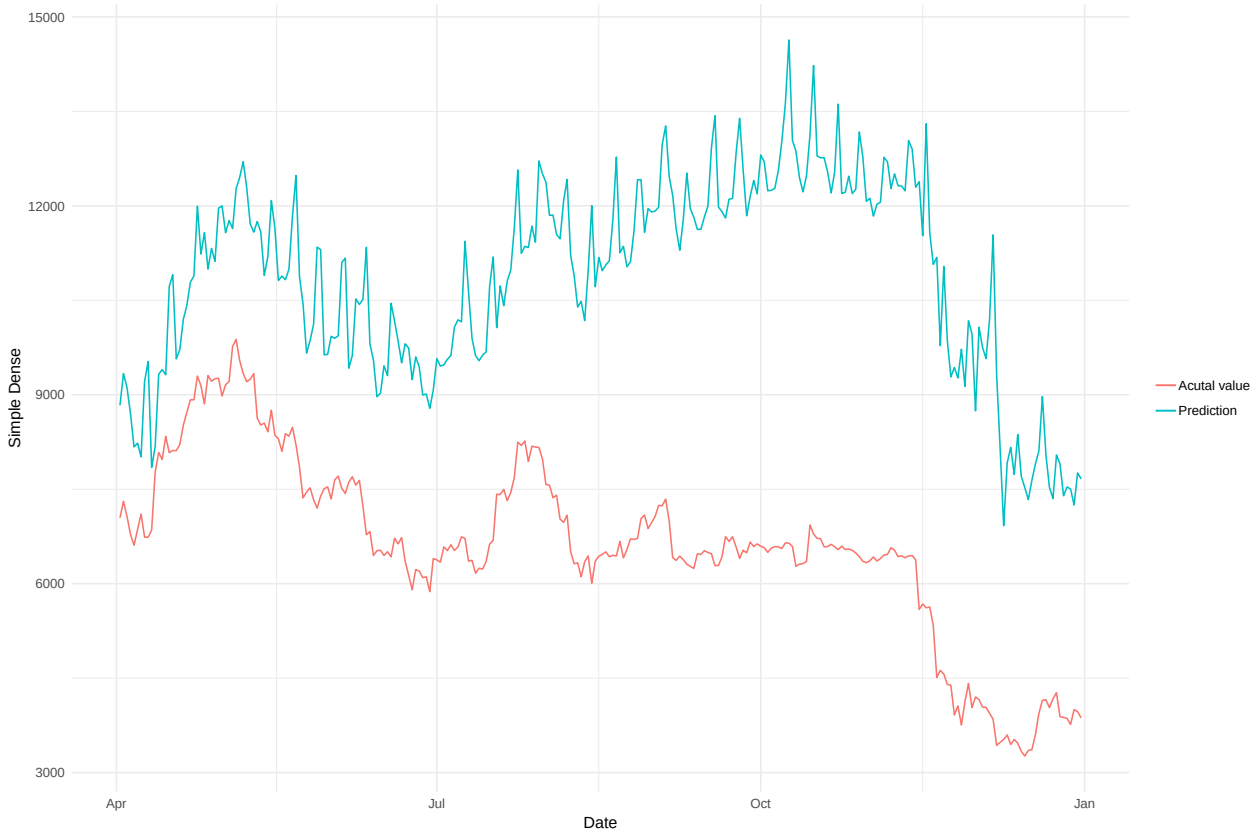


Figure 11: Predicted results vs. actual

6.2 Deep Dense Neural Network

6.2.1 Implementation of tuning strategy

To take the complexity of the neural network a step further, we decided to construct a deep dense neural network to see if this could help us make better predictions for the closing price of Bitcoin. Our initial approach was to construct and tune a deep dense network using our complete data set as input (DNN 1 in Table 4), which included all added- and feature engineered variables. Subsequently, we wanted to see if we could make better predictions by using our PCA and Autoencoder data sets. By using these data sets instead of the complete data set, we consider the most relevant information from our independent variables, gathered in a smaller set of features. Consequently, we tuned, trained and validated two additional and separate deep dense networks; one network taking the 10 variables explaining most of the variance in our *PCA* data set as input (DNN 2 in Table 4), and one network taking the compressed variables from our *Autoencoder* model as input (DNN 3 in Table 4). As such, three separate deep dense neural networks were constructed in total.

Table 4: Overview of Deep Dense Neural Networks

Network	Dataset
DNN 1	All features
DNN 2	PCA
DNN 3	Autoencoder

Similarly to the simple dense neural network, we implemented our models applying the **Keras** package in R. Additionaly, the package **tfruns** was used. With respect to tuning, the tuning strategy described in the section 4 was applied. To identify the best structure of the networks, we did some initial runs trying different optimizers and regularizes. Here, the best optimizer ended up being *Adam*. The most effective regularizer did on the other hand turn to out to be *ridge*. When fitting the models, the number of epochs was set to 500 for most of the models. A callback early stopping was also included, whereof a patience of 150 epochs was set to reduce time spent on models that did not improve our results. However, as model tuning is a very time-consuming and computationally expensive process, we found it necessary to limit the number of tuning runs to a reasonable amount. In total, we ended up doing about 40 tuning runs, where the set of tuning parameters was chosen at random from lists of possible value-combinations. These parameters are illustrated in Table 5. In practice, each run took about 3 hours to complete, depending on whether

the early stopping was realised or not.

Table 5: Overview of Parameters Chosen at Random

Parameters randomly tuned
Different numbers of hidden layers
Units for each hidden layer
Regularizer lambdas
Optimizer learning-rate
Dropout rate
Batch size

6.2.2 Results

The runs on the different data sets yielded the following results, as illustrated in Table 6.

Table 6: Validation and Test Results

	Simple NN		DNN 1		DNN 2		DNN 3	
	Validation	Test	Validation	Test	Validation	Test	Validation	Test
mae	1.743	2.885	5.465	1.978	5.117	1.732	5.033	1.676
mape	16.00%	48.00%	47.80%	28.98%	44.11%	26.10%	43.28%	25.41%

Comparing the validation results for our deep dense models to the less complex simple dense model, we can see that with an *mape* ranging from about 43-48% for our deep dense networks, we are on average missing by a lot more than the simple dense network. On the other hand, when building machine learning models, our goal is typically to build a generalized model that will perform well on new unseen data. Thus, when evaluating the models' performance, we are interested in testing how it performs on the test set. Considering the gradually decreasing *mape* on the test set, we can say that we are moving in the right direction. Further, as we can observe, the best performing model was trained on the *Autoecoder* data set (DNN 3). The model structure and the hyperparameter value combination of the best performing model is shown in Table 7.

Even though applying our *PCA* and *Autoencoder* data sets help us reduce the loss, we are still far from reaching the loss of our benchmark with a *mape* of 2%. By analyzing the loss metric that our best deep dense network gave us, we identified a slight decreasing trend. This indicated that we could achieve a lower loss by training the model using a higher number of epochs. As a result, we

Table 7: Dense Neural Network Structure

Parameter	Value
Hidden layer 1	100
Hidden layer 2	50
Hidden layer 3	100
Hidden layer 4	50
Output layer	1
Batch Normalization	All layers
Drop out rate	0.1
Optimizer	Adam (l = 0.001)
Epochs	300
Batch	128

recreated the structure and hyperparameter value-combination of our best performing model, and ran it for 2000 epochs using a call back patience of 500. In figure 12 below, we can see a comparison of the best model fitted on both 300 and 2000 epochs. After not improving for 500 epochs, the fitting process was stopped at epoch 1174 with an *mape* of 41.32%, this is almost 2pp better than before.

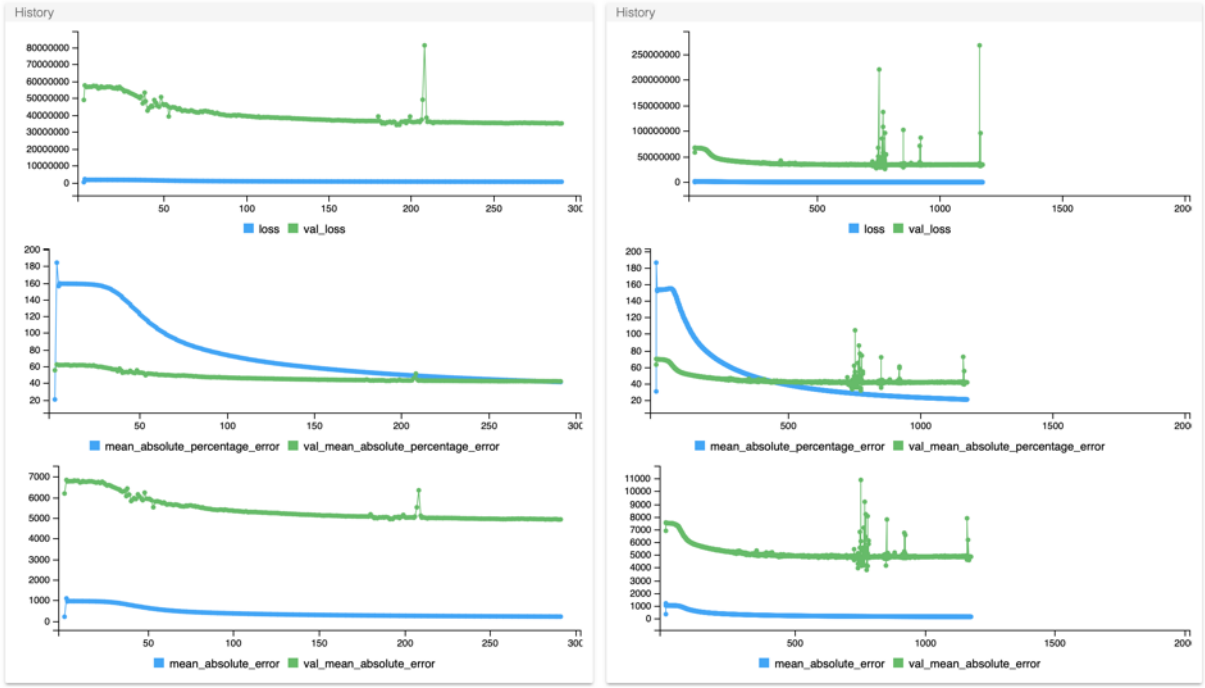


Figure 12: Comparison of the two best deep dense runs

By implementing our new model on the *Autoencoder* test set, we ended up with the results shown

in Table 8.

Table 8: Validation and Test Results

	DNN 3 - 300 epochs		DNN 3 - 2000 epochs	
	Validation	Test	Validation	Test
mae	5.033	1.676	4.851	1.548
mape	43.28%	25.41%	41.32%	23.91%

Further, to get a better overview of the predictions done for the test period, we plotted the actual Bitcoin price against the predictions, as shown in Figure 13.

Figure 13: Predicted results vs. actual

As we can see in Figure 13, it seems like our model is trying to make predictions close to an "average" price that lowers the overall loss as much as possible. By using *mse* as our loss function, large errors are penalized heavily. This might have reduced the models ability to capture the volatility in the actual *Bitcoin* price, forcing it to predict something that on average gives a lower loss compared to trying to follow the highs and lows of the volatile price.

6.2.3 Observations

With respect to the results above, we could say that none of our deep dense models are able to learn effectively. Hence, the networks are not able to pick up patterns, which can indicate that the independent variables in our data sets do not explain our dependent variable properly. On the other hand, by compressing information so that we only keep the information that explains most of the variance in our data sets, together with noise-reduction, we are able to reduce the error-rate. Yet, because the data we feed to the model appears to not explain the dependent variable properly, the loss can only be reduced by so much; even with an increase in the number of epochs. Therefore, with respect to our naive benchmark, we cannot say that our efforts towards applying different deep learning techniques was worthwhile. Not only is it a time-consuming process, but it also gave worse results.

6.2.4 Potential Improvements

As we can see by looking at the results, our best model is performing poorly compared to our benchmark. There could be many reasons for why our models struggles to pick up patterns in the data. First of all, if the input that the model trains on do not explain the fluctuations and the value of the Bitcoin price, the model will never be able to learn generalized patterns that will work well for future predictions on new unseen data. It would be interesting to see if we could have made better predictions by training and feeding the models smaller subsets of features from our data set. Because we had limited time available for this project, we could only manage to complete a limited amount of runs. Also, by increasing the number of runs with different hyperparameter-value combinations we would likely have been able to further reduce the average error rate for our predictions.

6.3 Recurrent Neural Network

Input

Since the RNN models require so much extra computing power, compared to the previously discussed models, in addition to the fact that the dense model performed best with the Autoencoder data set, we chose to train the RNN models using this data set. By doing so, we were able to somewhat reduce the computing time and thereby we were able to focus more on the model tuning. Further, we chose to use the past 10080 observations (one week) to predict the closing price 24 hours ahead. The choice of using only the past 10080 observations was also done in order to reduce the processing time.

Generator function

Contrary to the feedforward networks, RNNs take sequences of vectors, rather than a vector of inputs. The inputs then have to be 3 dimensional with rows, columns and timesteps. In order to feed the data correctly to the RNN, we needed to convert our 2D data into a 3D tensor. To resolve this, while freeing up as much memory as possible when running the computational expensive models, we decided to create a generator function. With a generator function, the samples and targets are being created while the model is training, rather than loading the full data set into the computer's memory. The implementation of the generator function proved to be more challenging than what we initially though. When feeding a model with data, using a generator function, there are a few extra parameters that need to be accounted for. For instance, the **Keras** data generator is meant to loop indefinitely, meaning it does not have the ability to determine when one epoch has been completed. To resolve this, we had to calculate and specify the *steps per epoch* by dividing the number of observations in the given set by the batch size. The fact that the generator function has to be created so that it feeds data indefinitely also means that, if not accounted for, discrepancies between the number of predicted values and de actual values might occur. We solved this problem by setting the length of the validation- and test set equal to $steps\_per\_epoch * batch\_size$ that is closest to the number of ovservations divided by the batch size. By doing so, we make sure that the length of the predicted values and the actual values are the same, since the model stops predicting where $nrow(test\_set)\%\%(steps\_per\_epoch * batch\_size) = 0$. Lastly, we had to implement a solution to our generator function so that it only generates x values, when this is required. This was done so that we could use the generator function when predicting, as the $predict\_generator()$

function only takes x-inputs.

6.3.1 Single layer GRU

6.3.1.1 Implementation of tuning strategy

We started off by implementing a single layer GRU model. The single layer GRU model was used as a first approach to RNNs and can therefore be seen as a baseline for our RNNs. The implementation of the model followed the same approach as described in previous sections, with **Keras** and the package **tfruns** being the main focal points of the implementation. The specific hyperparameters that were tuned in our single layer GRU model can be seen in Table 9. We further tried different runs with both the Adam optimizer and the RMSProp in order to see which optimizer that provided the best results.

Table 9: Tuning parameters, single layer GRU

Tfruns
Units
Steps per epoch
Number of epochs
Learning rate

6.3.1.2 Results

By running different models with different values for the parameters in Table 9, and by comparing the results for each run, we found that the model in Table 10 yielded the best results on the validation set.

Table 10: Best model, single layer GRU

Parameter	Value
Layer 1 - GRU	100 units
Output layer	1 unit
Optimizer	RMSProp (l = 0,001)
Epochs	200
Batch size	128
Steps per epoch	2

The model in Table 10 yielded an mae of 9.774,21 on the validation set, which was slightly better than the second best single layer GRU, which yielded an mae of 9.937,23. We then used the model that performed best on the validation set to make predictions on the test set. This gave us a test set mae of 5.744,91 and an mape of 90.60%. The plot of predictions against the actual values for the test set can be seen in Figure 14. These results are significantly worse than the results from the simple dense model, the deep dense model and far from the benchmark we introduced earlier.

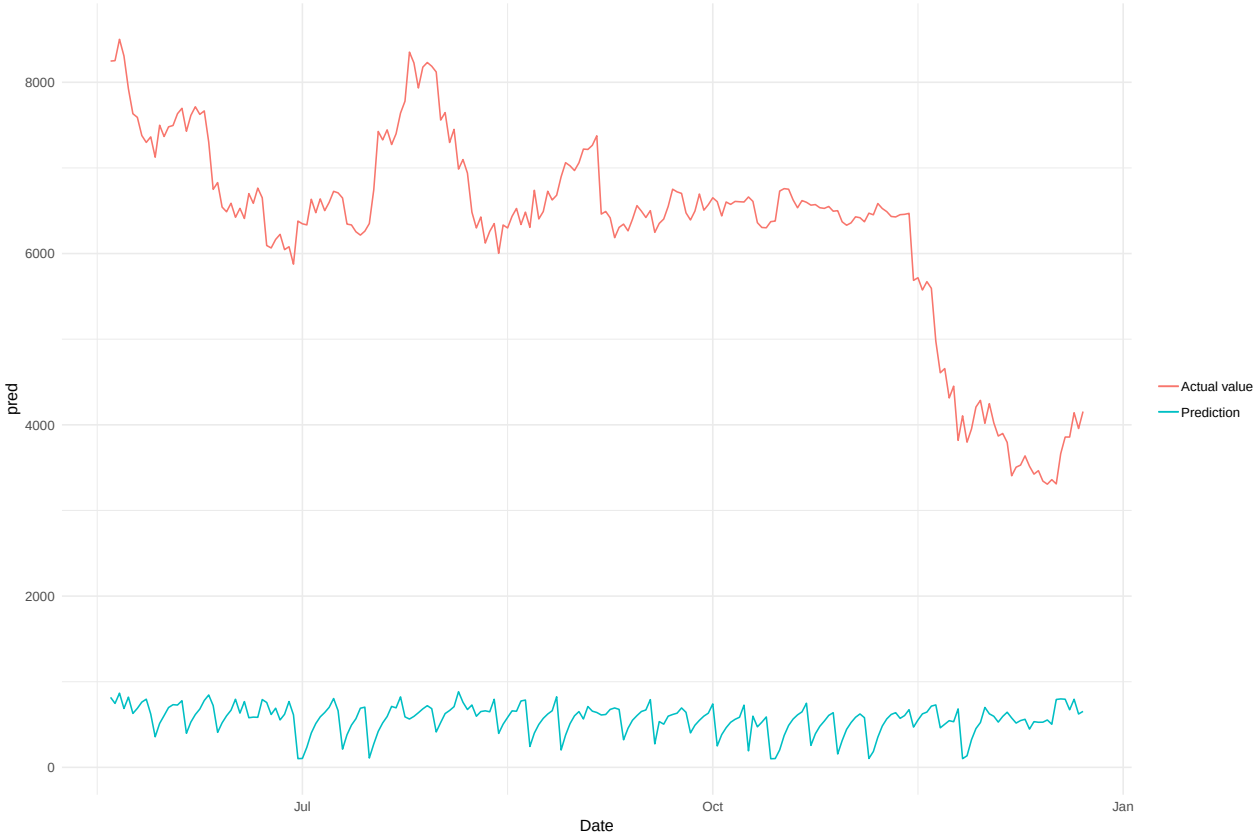


Figure 14: Predicted results vs. actual

As can be seen in Figure 14, the model is predicting the values to be far below the actual values and the predicted values are not following the overall trend in the data set. Both the metrics and the plot show that the model is performing significantly worse on the test set compared to the dense models and the benchmark. By looking at the plot, we can see that the model is predicting some values to be higher than others, meaning that it appears to capture some parts of the volatility in the data set.

6.3.2 Stacked GRU

6.3.2.1 Implementation of Tuning Strategy

In order to potentially increase the representational power of the RNN, we experimented with a stacked RNN by adding an extra GRU layer to our initial model. In addition to adding an extra layer, we specified that our first layer should return the full sequences, making the output of the first layer a 3D tensor of shape [batch size, timesteps, output features]. This is essential, considering the fact that RNNs only accept multidimensional inputs and this applies to each individual RNN layer as well. For the stacked layer GRU model, we also used the **tfruns** package in order to tune the hyperparameters. A list of the different hyperparameters we tuned can be seen in Table 11. Since each epoch in our stacked GRU model on average took more than 15 minutes to complete, we implemented an early stopping, stating that the model should stop if the validation loss had not improved in 30 epochs. By doing so, we were able to cut down the time spent on tuning models that did not improve our prediction results.

Table 11: Tuning parameters, stacked GRU

Tfruns
Units in layer 1
Units in layer 2
Steps per epoch
Number of epochs
Learning rate

6.3.2.2 Results

By running different models with different values for the parameters in Table 11 and by comparing the results, we found that the model in Table 12 yielded the best results on the validation set.

The model in Table 12 yielded an mae of 9.679,25 on the validation set. The mae for the best stacked GRU model was somewhat lower than the second best stacked GRU model, which yielded an mae of 9.805,78, and slightly lower than the validation mae from the single layer GRU. We then used the model that performed best on the validation set to predict on the test set. This yielded an mae of 6.237,91 and an mape 98.80%. The plot of the predictions against the actual values for the test set can be seen in Figure 15.

Table 12: Best model, stacked GRU

Parameter	Value
Layer 1 - GRU	32 units
Layer 2 - GRU	64 units
Output layer	1 unit
Optimizer	Adam (l = 0,001)
Epochs	41 (Early stopping)
Batch size	128
Steps per epoch	5

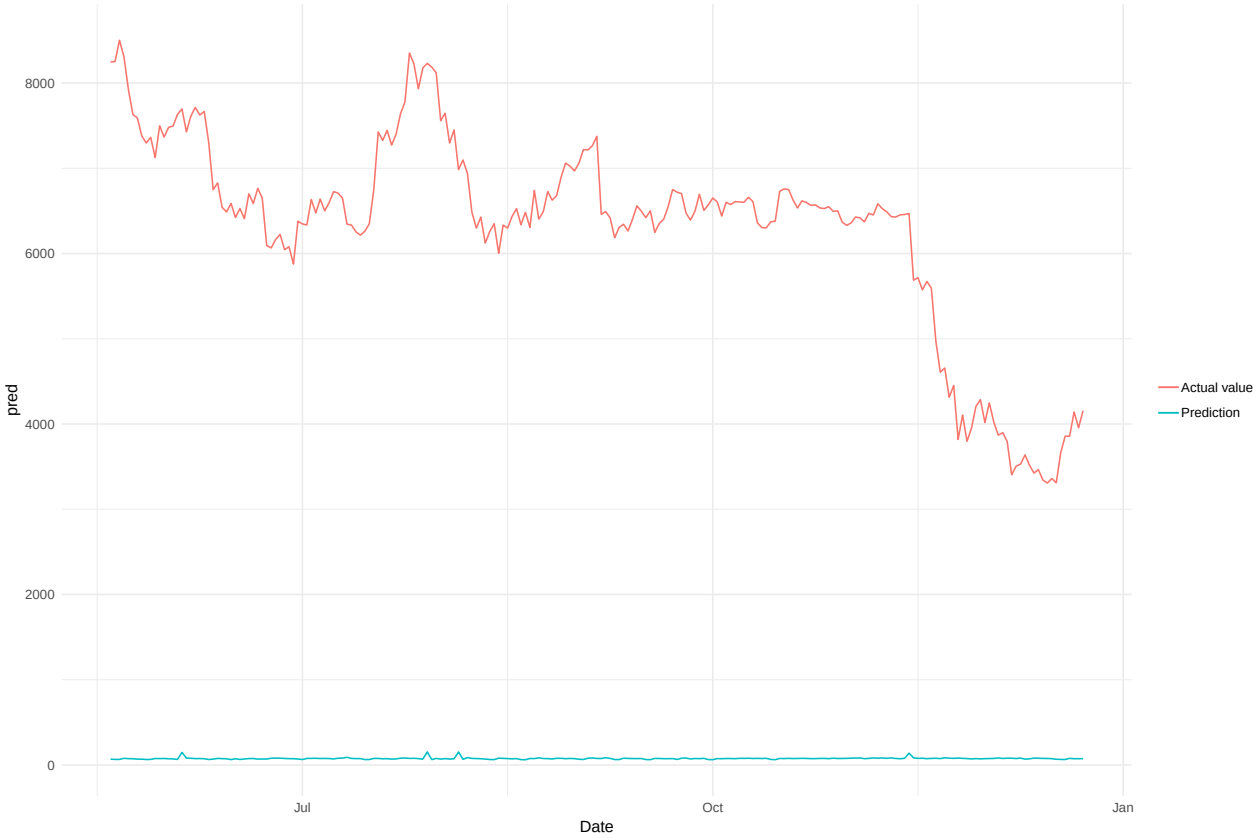


Figure 15: Predicted results vs. actual

The results from the test set show that the stacked GRU model performs worse than the single layer GRU and significantly worse than both the dense models and our benchmark. By looking at the plot in Figure 15, we can see that the predictions from the stacked GRU model are fairly stable, on a level far below the actual values. By looking at the graph, we see that the stacked GRU model was not able to capture the volatility at all. This can also be seen in Table 13, which

shows that the minimum-, mean- and median values are fairly close to each other.

Table 13: Descriptive statistics, stacked GRU

Mean	74,94
Median	74,82
Min	60,30
Max	266,67

6.3.3 Potential improvements

As was be seen by the results, the implementation of RNN models did not improve the results, compared to both our benchmark and the simple dense models. However, we note that due to the limitations in processing power and time, we were not able to fully utilize the potential of RNN and GRU in particular. For instance, we observed that for some of our simple dense models, the model was not able to reduce the loss until after layer 150 and even 200 on some occasions. When we ran RNN models, we ran 100 epochs at the most for our stacked GRU model and 200 epochs at the most for our single layer GRU model. This means that it is possible that our models could have improved the predictions if we were able to run the model, using more epochs. Further, we only used the past week to predict the next closing price 24 hours ahead. The fact that our models only have access to one week's worth of data is likely to limit the models' predicting power. With access to more computing power, it could have been interesting to see how the models would perform if they were able to analyze the full data set. Lastly, even more combinations of different hyperparameters, including batch size, steps per epoch, etc. could also have proved to increase the predictions and this too could have been interesting to investigate further.

6.4 Concluding remarks - DNN and RNN

When comparing the results from the best performing deep learning models in each section on the test set, we see there is a clear pattern. As can be seen in Table 14, despite the extra computational power required to run the more complex models, the simple models tend to outperform the more complex ones on all measures. The very best result was achieved by simply predicting the current closing price to be the closing price 24 hours ahead. Secondly, we can see that the recurrent neural networks perform far worse than all the other models and the most complex GRU model

had the lowest test result. It is an interesting observation that a more complex model does not necessarily mean that your results will improve. Both the single layer GRU and the stacked GRU performed worse than the previously discussed models and significantly worse than the benchmark. Elaborating further on this, one could think that our complex models in theory should be able to find and improve on the well-performing model that is our initial baseline, however, it is not able to do so. A theory as to why this is not the case, is that the complex models are not looking for a simple solution. The hypothesis space, the space of all possible two-layer networks with our configurations, is fairly complicated. Looking for a simple model in a space of complicated models can be very difficult, meaning that the simple solution might be unlearnable, despite the fact that it hypothetically is a part of the hypothesis space. As stated in Chollet & Allaire (2018), "a pretty significant limitation of machine learning in general is that unless the learning algorithm is hardcoded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem" (Chollet & Allaire, 2018, p 199).

Table 14: Model comparison - Deep learning

	Single layer GRU	Stacked GRU	DNN 3 - 2000 epochs	Benchmark
mae	5.744,91	6.237,91	1.548,00	159,54
mape	90.60%	98.80%	23.91%	2.00%

Lastly, we note that when it comes to markets, using machine learning models to predict future values is a difficult endeavor. As previously discussed, the market for Bitcoin has proven to follow the Efficient Market Hypothesis, meaning that simply predicting the future values based on only publicly available data has so far been unsuccessful. In general, past performance is not a good predictor of future returns when it comes to markets (Chollet & Allaire, 2018). It is therefore not surprising that the more complex models will not be able to make good predictions, given the nature of the data set and the task at hand. It is also important to keep in mind that the data set consists of extremely large fluctuations. Further, the market for cryptocurrencies, in the time period we have analyzed, has been described as a bubble fueled by speculation and irrational investments.

After an extensive elaboration on the first research question, in other words our regression problem, we will now turn to our second research question. Rather than predicting the exact price of Bitcoin, we will now try to predict the direction of the movement in Bitcoin prices. From now on, this

paper will therefore address a binary classification problem. The following section will accordingly elaborate on the underlying method that has been used to address this issue.

7 Model Ensembling

7.1 Introduction to Model Ensembling

Model ensembling has marked itself as a robust technique for possibly achieving improved results within the machine learning field. In this technique, predictions from several different models are pooled together with the purpose of providing better predictions. Hence, rather than relying on just one model, the technique puts emphasis on taking several independent models into consideration. This aspect makes the technique potentially very powerful, as different machine learning models tend to capture different aspects, trends and patterns. A combination of models may therefore provide a more accurate perspective, and as such also more precise predictions. That being said, diversity constitutes an important aspect of model ensembling. A good ensembled model should therefore not solely consist of independent models performing as well as possible, but also as different as possible. In that respect, combining models across the machine learning landscape may turn useful (Chollet & Allaire, 2018). Consequently, we find it interesting to combine neural networks with a selection of other machine learning techniques. With inspiration from the **Keras** legend *François Chollet* and one of his previous attendances at *Kaggle*, we will develop an ensemble model consisting of a neural network and two tree-based models, namely Gradient Boosting Machine and Random Forest.

7.2 Introduction to Random Forest and Gradient Boosting Machine

With respect to Random Forest and Gradient Boosting Machine, a brief reference should be made to decision trees. Decision trees are a tree-based method similarly structured as flowcharts, whereof classifying input data, as well as prediction of input-given output values, can be carried out in a visual manner (Chollet & Allaire, 2018). Random Forest introduces an additional element, namely creating significant amounts of specialized decision trees, followed by ensembling the respective outputs. Random Forest can be applied to a variety of machine learning issues, thus making it a good technique for many, yet often, trivial tasks (Chollet & Allaire, 2018). Similar to Random Forest, Gradient Boosting Machine has become a widely used machine learning technique, perhaps even a favorite in competitive machine learning environments such as *Kaggle*. The technique applies gradient boosting by considering weak aspects of previous models, often decision trees, to build an ensemble of weak predictions with the purpose of converting them to stronger predictions. Success-

fully applied, Gradient Boosting Machine tends to perform better than Random Forest. Similar to Random Forest, Gradient Boosting Machine is a technique that can be applied to a variety of machine learning problems (Chollet & Allaire, 2018).

7.3 Approach

As we've discovered so far in this report, predicting the Bitcoin price has turned out to be a difficult assignment with respect to achieving good predictions. To address our second research question, we will be looking at a binary problem, hereby predicting whether or not the direction of the close price is negative (zero) or positive (one). This is interesting, as looking at a binary problem – in contrast to the regression problem we've worked with so far, provides a new perspective to the data set. Looking at a binary classification problem similarly appears reasonable with respect to the nature of tree-based models. Further, as illustrated in the sections concerning data, we are dealing with a large data set. Like neural networks, tree-based methods and particularly Random Forest, are rather computationally expensive machine learning techniques. This naturally limits our possibilities with respect to proper model tuning. To address this, we have tuned our Gradient Boosting Machine and Random Forest models with inspiration and guidance from research papers within the machine learning field. That said, we have in the process of creating this chapter, spent a considerable amount of time on tuning. Further, all models were ran on the Autoencoder data set, as this lets us focus on the most relevant information, thereby reducing the time and computing power. Lastly, the intention of this chapter is not to beat our previous models, as they relate to a different research question. Moreover, the intention is neither to build exceptional stand-alone models, but rather to explore whether ensembling of models can provide value in the form of better predictions, compared to the individual performance of the respective models. In that regard, we will focus on three measures, namely accuracy, sensitivity and specificity.

7.4 Implementation

7.4.1 Tuning Strategy - Random Forest Model

Our Random Forest model was built using *caret*, a package streamlining model-building for prediction purposes. Caret incorporates various packages, such as *gbm* and *randomForest*. The *randomForest* was accordingly used to to build a Random Forest model. To form our tuning strategy with

respect to Random Forest, inspiration was taken from several articles. According to the article "Hyperparameters and Tuning Strategies for Random Forest", the model is known for functioning well with default values. Further, the algorithm is described as less tunable compared to many other algorithms. Yet performance improvements can be achieved through hyperparameter tuning. Some general guidelines are provided by the article with respect to n.trees and key hyperparameters, namely, mtry, sample size and node size (Probst, Wright, & Boulesteix, 2018).

Concerning n.trees, i.e. the amount of trees, the number should be set high, whereof a higher amount typically provides better performance. Mtry - the amount of variables sampled in a random manner at every split, sample size – size of sample drawn, and node size – lower limit of terminal node size, do, on the other hand, control the randomness associated with the Random Forest algorithm. Among these, mtry is described as the most influential parameter according to literature, and supported by experiments conducted in relation to the aforementioned article. Sample size and node size only have minor influence (Probst et al., 2018). With this in mind, and the fact that the Random Forest algorithm is quite computationally expensive, emphasis was put on testing various n.trees and mtry. As for trees, multiple research papers provide insight. In the article "How Many Trees in a Random Forest?", n.trees in the range of 64 to 128 are suggested, whereof, for instance, 256, 512 and 1024 n.trees provided insignificant improvements in accuracy (Oshiro, Perez, & Baranauskas, 2012). The article, "To tune or not to tune the number of trees in Random Forest", does, on the other hand, indicate substantial increase in accuracy – in particular between 10 to 500 n.trees (Probst & Boulesteix, 2018). Yet, as these results are achieved on quite small data sets, contrary to our larger data set, we find it necessary to try several values of n.trees on both sides of 512 n.trees. Last, for classification problems, mtry is defined by the square root of the number of predictor variables, in our case the square root of 5, i.e. approximately 2. The default value is, according to literature, a reasonable value, but improvements can sometimes be achieved by tuning this parameter further (Probst & Boulesteix, 2018). Hence, we find it interesting to try different mtry values - also above 2. Accordingly, we made the following tuning strategy:

Table 15: Tuning Strategy – Random Forest

Parameter	Values
ntree	64, 128, 256, 512, 1024
mtry	2, 3, 6

As illustrated in Table 15, a total 15 combinations were tested. This was done through the application of tuneRF, a built-in function in the **randomForest** package, which can be applied to tune the mtry parameter. The combination of 1024 n.trees and a mtry of 6, yielded the lowest **oob** error rate. Hence, we proceeded with 1024 n.trees and a mtry of 6.

7.4.2 Tuning Strategy - Gradient Boosting Machine Model

Similar to Random Forest, our **Gradient Boosting Machine** model was built in *caret*, using the incorporated **gbm** package. In order to form a tuning strategy, guidance and findings from two articles, namely "Generalized Boosted Models: A guide to the gbm package" and "Gradient Boosting Machines, a tutorial", were sought. The articles put emphasis on three tuning parameters in particular, hereby n.trees, shrinkage and interaction.depth. The first, n.trees, constitute the amount of trees to fit, i.e. the amount of iterations, whereof Ridgeway proposes n.tree amounts in the range of 3000 to 10 000 (Ridgeway, 2007). Shrinkage on the other hand, constitutes the learning rate. Here, Ridgeway proposes a rate in the range of 0.01 to 0.001, where a lower rate can lead to better performance, yet, at the expense of computing time and resources (Ridgeway, 2007). Further, interaction.depth defines the maximum depth of each tree. In the abovementioned article written by Natekin and Knoll, the benefit of complex trees over compact trees are discussed, whereof complex tree structures, with interaction.depth over 20, seldom appears to provide significant benefit compared to compact trees with a interaction depth of 5 (Natekin & Knoll, 2013). A few other aspects should also be mentioned. With respect to loss function, a Bernoulli distribution was chosen, as this is often used for classification problems. Further, the models were ran with a 5-fold cross validation, in line with Ridgeway's recommendation of applying, either, a 5-fold, or 10-fold cross validation (Ridgeway, 2007). It should be noted that a 10-fold cross validation would drastically increase the computing time, which is why we found it reasonable to choose a 5-fold cross validation instead. Based on this, the following tuning strategy was formulated:

As we can observe from Table 16, a total of 18 unique combinations were ran. This was carried out through a grid search, whereof two grid runs were performed, testing 9 combinations each. First, one run was done testing all unique combinations with a fixed shrinkage of 0.1. Second, a new run was carried out, testing all unique combinations with a fixed shrinkage of 0.01. The different parameter values were set by choosing the minimum value proposed in the abovementioned research articles, hereby 0.01, 3000 and 5 respectively. This was done to ensure proper parameter values,

Table 16: Tuning Strategy – Gradient Boosting Machine

Parameter	Values
n.trees	1000, 3000, 5000
shrinkage	0.1, 0.01
interaction.depth	3, 5, 7
cv.folds	5
distribution	bernoulli

while still taking our limited computing power and rather big data set into consideration. Yet, as putting some additional effort into tuning can provide value, further combinations on both the lower and upper end of 0.01, 3000 and 5 were tested. One exception was however made with respect to shrinkage, as applying a lower shrinkage than 0.01 turned out to be a very computationally expensive exercise. The best performing combination of parameters in run 1 were 0.1, 5000 and 7. Similarly, a combination of 0.01, 5000 and 7 provided the best results in run 2. Overall, the model associated with run 2 appears to be the best performing model. Hence, and perhaps not surprisingly, a lower learning rate seem to provide some additional value in this particular case. Going forward, we proceeded with the model resulting from the second run.

7.4.3 Tuning Strategy - Neural Network Model

Similar to our previous neural networks covered in this report, we applied the package **tfruns**, testing various combinations of parameters, yet in practice building the network sequentially, in order to find the best performing model. First of all, models with both one and several layers – including various number of units, were ran. ReLu activation was added to the respective layer(s), an activation function with a track record of performing very well, and as such also a widely applied activation function for deep learning purposes (Nwankpa, Ijomah, Gachagan, & Marshall, 2018). As we are dealing with a binary classification problem, a probability output in the range of 0 and 1 is required. Hence, the output layer consists of 1 unit and a sigmoid activation function, which ensures that the output values are squashed into the appropriate scale (Chollet & Allaire, 2018). With respect to regularizers, both lasso and ridge have been tested, as avoiding overfitting is very important. With respect to optimizers, two separate optimizers have been tried. Although one ideally would test a broader scale of optimizers, we found it reasonable to pick two, hereby Adam and RMSProp, and test these extensively for this model ensembling exercise. Adam was

chosen, as this optimizer is considered state-of-the-art by many machine learning professionals (Ruder, 2016), thus making it interesting to test. Further, RMSProp was selected as it appears as a versatile optimizer that can be applied to various problems, including our binary problem (Chollet & Allaire, 2018). As for loss the function, *binary_crossentrepy* was chosen, as it is considered the most appropriate, if not, the best choice for binary problems (Chollet & Allaire, 2018). Last but not least, the models were ran on 512 epochs and with a batch size of 256, enabling us to do some proper model-building, yet within reasonable amount of time and use of computing power.

Table 17: Tuning Strategy – Neural Network

Parameter	Values
Layer 1 - units	50, 100, 150
Layer 2 - units	50, 100, 150
Output layer	1
Activation	relu (layer(s)), sigmoid (output)
Regularizer	l1, l2 (layer(s))
Loss	binary_crossentropy
Optimizer	adam, rmsprop
Epochs	512
Batch	256

In practice, two runs were completed, the first being only with one layer, testing 50, 100 and 150 units, with two different types of regularizers, namely lasso and ridge. In addition two different optimizers, hereby Adam and RMSProp, were tested. Hence, the first run comprises a total of 12 different combinations. To limit the amount of combinations for the second run, it applies the best performing *layer 1* unit amount from the first run. Thereby, another 12 runs were performed. As one layer provided better results than two layers, a third layer was not tested. Overall, a combination of one layer with 150 units, l1 regularizer (l = 0.01) and Adam provided the best results. We therefore proceeded with this model combination going forward.

7.5 Results – Random Forest, Gradient Boosting Machine and Neural Network

7.5.1 Validation Results

The validation results for the respective models are illustrated in Table 18 and Table 19.

Looking at the validation results above, we can observe that our neural network provides the highest

Table 18: Confusion Matrix – Random Forest, Gradient Boosting Machine and Neural Network

	RF		GBM		NN	
	zero	one	zero	one	zero	one
zero	55482	57119	17572	95029	64203	48398
one	59563	64140	22407	101296	54303	69400

Table 19: Validation Results – Random Forest, Gradient Boosting Machine and Neural Network

	RF	GBM	NN
Accuracy	0.5062	0.5030	0.5654
Sensitivity	0.5290	0.5160	0.5891
Specificity	0.4823	0.4395	0.5418

accuracy of 56.5%, followed by Random Forest and Gradient Boosting Machine with 50.6% and 50.3% respectively. Similarly, the neural network achieves the highest sensitivity rate, whereof the neural network model was able to classify close to 58.9% of the actual positives correctly. In comparison, the Random Forest and Gradient Boosting Machine models classified positives correctly at rates of 52.9% and 51.6%. The specificity rates do however indicate that the models are performing slightly worse at classifying negatives correctly. Similar to accuracy and sensitivity, the neural network provides the highest rate, hereby 54.2%, followed by Random Forest and Gradient Boosting Machine with 48.2% and 43.9%. Consequently, the neural network, compared to our tree-based models, provides the strongest results on the validation set. Thus, looking at the validation set alone, the tree-based models appear to not provide any significant value compared to the neural network. Further, the Random Forest model outperformed the Gradient Boosting Machine model, even though we intuitively expected the opposite.

7.5.2 Test Results

The test results for the respective models are illustrated in Table 20 and Table 21 below.

On the test set, one can observe that the accuracy and sensitivity has decreased for all models. As for specificity, Random Forest and Gradient Boosting Machine performed better, contrary to the neural network. In fact, our neural network performs worse than both Random Forest and Gradient Boosting Machine with respect to all of the three measures. More precisely, Gradient

Table 20: Confusion Matrix – Random Forest, Gradient Boosting Machine and Neural Network

	RF		GBM		NN	
	zero	one	zero	one	zero	one
zero	34363	166037	29493	170907	42193	158207
one	35065	158376	28433	165008	47057	146384

Table 21: Test Results – Random Forest, Gradient Boosting Machine and Neural Network

	RF	GBM	NN
Accuracy	0.4894	0.4939	0.4788
Sensitivity	0.4882	0.4912	0.4806
Specificity	0.4949	0.5091	0.4728

Boosting Machine now provides the highest accuracy of 49.4%, followed by Random Forest model and neural network, with accuracies of 48.9% and 47.9%. Concerning sensitivity, all models perform worse at classifying positives on the test set compared to the validation set. The Gradient Boosting Machine provides the highest rate of sensitivity, hereby 49.1%. Random Forest and neural network follow closely with sensitivity rates of 48.8% and 48.1%. Looking at the specificity, Gradient Boosting Machine and Random Forest are classifying negatives correctly slightly better than on the validation set, with rates of 50.9% and 49.5%. The neural network on the other hand, with its rate of 47.3%, predicts worse with respect to specificity on the test set compared to the validation set. Overall, the Gradient Boosting Machine now appears to provide the best performance, followed by Random Forest and neural network; the opposite of what was observed on the validation set. Accordingly, and frankly quite interesting, the models appear to capture different aspects in the data.

7.6 Ensembling the Models

So far, we've discussed the results of each model. When it comes to ensembling models, there are several approaches that can be applied in regard to pooling predictions. One of these, is taking the average of the predictions, in practice weighting the predictions equally. Another approach is weighted average, whereof more weight is put on the predictions performing the strongest. Likewise, predictions performing the weakest, are weighted lower. Application of weighted average, optimized based on the basis of validation data, provide a solid starting point (Chollet & Allaire, 2018), and

will as such be applied. In more details, will we test three different ensemble weight strategies. The first one will be an equally weighted ensemble (weight 1), weighting Random Forest to 33.3%, Gradient Boosting Machine to 33.3% and neural network to 33.3%. The second strategy is based on a simple grid search, hereby testing combinations in the range of 0 to 0.75 - with intervals of 0.25, that in total summarize to 1 (weight 2). Not surprisingly, as the neural network, followed by Random Forest, gave the best scores individually, a combination of 25% weight on Random Forest and 75% provided the best results on the validation set in regards to weight 2. Last but not least, the third set of weights, we set manually based on intuition (weight 3). Based on the validation results, we hence put close to two third of the weights, more precisely 65%, on the neural network. Next, the Random Forest – being the second-best performing model, has been assigned a weight of 20%. Lastly, the Gradient Boosting Machine received a weight of 15%. A summary of the weights are presented in Table 22.

Table 22: Summary - Weights

	RF	GBM	NN
Weight 1	0.33	0.33	0.33
Weight 2	0.25	0.00	0.75
Weight 3	0.20	0.15	0.65

7.7 Results – Ensembled Models

7.7.1 Validation Results

The ensemble weight strategies discussed above, yielded the following validation results, as illustrated in Table 23 and Table 24.

Table 23: Confusion Matrix – Model Ensembles

	Weight 1		Weight 2		Weight 3	
	zero	one	zero	one	zero	one
zero	19758	92843	63548	49053	15604	96997
one	24625	99078	54934	68769	21480	102223

As we can observe, weight 2, i.e. 25% weight on Random Forest and 75% weight on neural network, provides the best performance with respect to accuracy, sensitivity and specificity, classifying pos-

Table 24: Validation Results – Model Ensembles

	Weight 1	Weight 2	Weight 3	Best model - NN
Accuracy	0.5029	0.5599	0.4986	0.5654
Sensitivity	0.5162	0.5837	0.5131	0.5891
Specificity	0.4452	0.5364	0.4208	0.5418

itives and negatives correctly at a rate of 58.4% and 53.6%. As weight 2 puts high emphasis on the neural network, which performed the strongest on the previously discussed validation set, this is not unexpected. Yet, the neural network model alone still outperforms the ensemble models on the validation set.

7.7.2 Test Results

With respect to test results, the ensemble weights provided the results illustrated in Table 25 and Table 26.

Table 25: Confusion Matrix – Model Ensembles

	Weight 1		Weight 2		Weight 3	
	zero	one	zero	one	zero	one
zero	13651	186749	30759	169641	2795	197605
one	16773	176668	35222	158219	1614	191827

Table 26: Test Results – Model Ensembles

	Weight 1	Weight 2	Weight 3	Best model - GBM
Accuracy	0.4832	0.4798	0.4942	0.4939
Sensitivity	0.4861	0.4826	0.4926	0.4912
Specificity	0.4487	0.4662	0.6339	0.5091

As we can observe, weight 3 – whereof most weight is put on the neural network, followed by Random Forest and Gradient Boosting Machine, provides the strongest performance, in practice outperforming the stand-alone Gradient Boosting Machine results. That is however just barely with respect to accuracy and sensitivity. The specificity is on the other hand significantly higher. Thus, the best model ensemble appears to classify positives correctly slightly better, yet significantly better with respect to negatives. Further, comparing with the validation set, weight 1 and 2 performs worse

at the test set. As for weight 3, the accuracy and sensitivity have decreased somewhat, while the specificity is significantly higher. Thus, the best model ensemble is worse at classifying positives correctly, yet better at classifying negatives correctly on the test set compared to the validation set. Overall, the model ensembling appears to have provided a model that performs better with respect to the three measures we've taken into consideration in this analysis.

7.8 Concluding Remarks - Model Ensembles

We have in this section experimented with model ensembling, applying neural networks and tree-based models. A few remarks should however be noted with respect to the model performance we've observed in this part. First, the three measures we've considered in this section does not necessarily provide the full picture of the respective model's performance. For instance, the models tend to have high **no information rates**, in most cases exceeding the accuracy rate significantly. As the no information rate tells something about the data's biggest class, a best guess – only knowing the class distributions, would provide a drastically higher hit rate (HRAnalytics, 2017).

Another observation to consider, is that the best individually performing models appear to perform better in all aspects. More precisely, none of the models appear to provide good performance with respect to just one or two of the measures. For example, the best performing model on the test set, Gradient Boosting Machine, performs better than its peers on all of the three measures. If one model had a particular strength with respect to the three measures, and the next model had another, that could have added additional value. Intuitively, we would expect some models to perform better in certain aspects, yet that was not the case. With respect to weighting, application of optimization algorithms, for instance Nelder-Mead, could have been interesting to implement. However, due to limited time, we found applying optimization algorithms to be out of scope. Another interesting observation is that the best performing model on the validation set is the worst performing model on the test set. The opposite is the case for the worst performing model on the validation set, which provides the strongest performance on the test set.

With reference to the results itself and the observations we have discussed, it should be mentioned that both the validation- and test set are rather special periods in the history of Bitcoin, with significant fluctutation in both price increase and price decrease. Training on a less volatile time period in the history of Bitcoin, followed by validation and testing on a time period that more

or less can be described as the opposite, is unsurprisingly leading to a hard time for the machine learning techniques.

Overall, these are only a few aspects of what we have observed working with model ensembling. Thus, one cannot necessarily say that our models, either individually or ensembled, provide sufficient performance. Further, it is interesting to observe that simply predicting the biggest class, indicated by the no information rate, would have provided a higher accuracy than what we achieved. If we were to use this as a benchmark, we can conclude that model ensembling could not yield higher accuracy than simply predicting everything to be equal to the biggest class. Yet it was truly interesting to experiment with both model ensembling and tree-based models, as it led us towards other techniques available in the machine learning field.

8 Conclusion

We have in this paper worked with various machine learning techniques, addressing two objectives in particular, namely i) predicting the Bitcoin close price on a 24-hours basis, and ii) predicting the direction of the Bitcoin movement on a 24-hours basis. Most of our efforts were put on application of deep learning techniques facing a regression problem. Secondly, we applied model ensembling, working towards solving a binary problem.

With respect to our first objective, none of our models were able predict the Bitcoin price better than our benchmark. We therefore conclude that applying machine learning techniques were not worthwhile our efforts. As for our second objective; although the application of model ensembling appears to provide improvements in performance, compared to the respective models' individual results, the performance - neither individually nor ensembled suggest predictability of the directions of the Bitcoin movement.

Consequently, our findings in this report appears to further strengthen the argument concerning the cryptocurrency market being unpredictable.

9 Limitations

This project has been a travel into what we, based on our background in business economics, consider unknown territory. Consequently, the project has been one of the, if not the most, challenging and time consuming projects throughout our time at the Norwegian School of Economics. First, as large amounts of data are key with respect to application of deep learning techniques, we have spent a considerable amount of time on gathering, cleaning, preparing and taking decisions related to data. Similarly, a lot of time has been spent on learning about and getting in-depth knowledge of the different concepts and techniques within the field of deep learning. One of the greatest challenges however, was to build and tune models in a proper manner, and most of our effort was focused here. As stated by Professor Walt Pohl, the only way to really learn deep learning, is to sit down and actually apply available deep learning techniques. Consequently, we've found it particularly important to experiment with various techniques, such as dense neural networks, recurrent neural networks, autoencoders and model ensembling.

Even buying and setting up a "deep learning computer" turned out to be a challenging task. Initially we attempted to install Ubuntu - unfortunately without luck. In fact, just installing Tensorflow and Keras with GPU-support on Windows was a troublesome task in practice taking several days worth of work. Yet the effort gave us the opportunity to explore more than we could otherwise have done and did as such boost our learning outcome significantly. That being said - to our surprise, despite buying and setting up a computer with a good graphic card, hereby Nvidia Geforce RTX 2070, our data set, models and tuning efforts have required more computing power than what the machine alone could provide. Thus, our laptops and several VMware virtual desktops have been running constantly in addition. Even with this in mind, there has been a trade-off between proper model building, computing power and time, thus providing some limits. Nevertheless, in a period of time like this, where deep learning is being applied to a variety of applications - with huge success, we truly appreciate the opportunity to experiment with and learn more about the craft of mastering deep learning.

10 References

Abeslamidze, S. (2018). New data: Over third of btc circulating supply lost while 22 % held by speculators. *Coinspeaker*. Coinspeaker. Retrieved from https://www.coinspeaker.com/new-data-over-third-of-btc-circulating-supply-lost-while-22-held-by-speculators

Allaire, J. (n.d.). TensorFlow for r. *TensorFlow for R*. Retrieved from https://tensorflow.rstudio.com/tools/training_flags.html?fbclid=IwAR29MsVrFzQIfCeZOSBI_bBqk_JHnHIeLjUPi12MIycRiaT1s6w6uYyZ35c

Amini, A. (2018). *YouTube*. YouTube. Retrieved from https://www.youtube.com/watch?v=JN6H4rQvwgY

Barchart. (2019). Cryptocurrency market capitalizations. *Barchart.com*. Retrieved from https://www.barchart.com/crypto/market-capitalizations

Bartos, J. (2015). Does bitcoin follow the hypothesis of efficient market? *International Journal of Economic Sciences*, *IV*(2), 10–23. https://doi.org/10.20472/es.2015.4.2.002

Blockchain: The mystery of mining difficulty and block time. (2018). *Good Audience*. Good Audience. Retrieved from https://blog.goodaudience.com/blockchain-the-mystery-of-mining-difficulty-and-block-time-f07f0ee64fd0

Brownlee, J. (2017). A gentle introduction to exploding gradients in neural networks. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

Brownlee, J. (2018a). A gentle introduction to dropout for regularizing deep neural networks. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

Brownlee, J. (2018b). How to configure the number of layers and nodes in a neural network. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/

Brownlee, J. (2018c). What is the difference between a batch and an epoch in a neural network? *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

Brownlee, J. (2019a). Gentle introduction to the adam optimization algorithm for deep learning. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

Brownlee, J. (2019b). How to choose loss functions when training deep learning neural networks. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/

Brownlee, J. (2019c). How to configure the learning rate hyperparameter when training deep learning neural networks. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/

Chainalysis. (2018). Bitcoin's $30 billion sell-off. *Chainalysis Blog*. Retrieved from https://blog.c

hainalysis.com/reports/money-supply

Chollet, F., & Allaire, J. J. (2018). *Deep learning with r* (1st ed.). Greenwich, CT, USA: Manning Publications Co.

CoinDesk. (2018). What is segwit? *CoinDesk*. Retrieved from https://www.coindesk.com/information/what-is-segwit

Crosby, M., Pattanayak, P., Verma, S., & Kalyanaraman, V. (2016). Blockchain technology: Beyond bitcoin. *Applied Innovation*, *2*(6-10), 71.

Datarobot. (2019). Feature engineering for automated machine learning | dataset features. *English*. Retrieved from https://www.datarobot.com/wiki/feature-engineering/

Dertat, A. (2017). Applied deep learning - part 3: Autoencoders. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Developers, G. (2019). Regularization for simplicity: Lambda | machine learning crash course | google developers. *Google*. Google. Retrieved from https://developers.google.com/machine-learning/crash-course/regularization-for-simplicity/lambda

Donges, N. (2018). Recurrent neural networks and lstm. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/recurrent-neural-networks-and-lstm-4b601dd822a5

Fan, C., Sun, Y., Zhao, Y., Song, M., & Wang, J. (2019). Deep learning-based feature engineering methods for improved building t energy prediction. Retrieved from https://www.sciencedirect.com/science/article/pii/S0306261919303496

Goldacre, B. (2011). Unemployment is rising – or is that statistical noise? | ben goldacre. *The Guardian*. Guardian News; Media. Retrieved from https://www.theguardian.com/commentisfree/2011/aug/19/bad-science-unemployment-statistical-noise

Gozzoli, A. (2018). Practical guide to hyperparameter searching in deep learning. *FloydHub Blog*. FloydHub Blog. Retrieved from https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/?fbclid=IwAR1aNsFzwaLJIKaLl0oUfJM-k4F5Etuv6lBLVB68SOtTH6yWGWunZkrXFuk

Gray, C. (2018). The alpha scientist. *The Alpha Scientist Atom*. Retrieved from https://alphascientist.com/feature_engineering.html

Hayden, L. (2018). PCA analysis in r. *DataCamp Community*. Retrieved from https://www.datacamp.com/community/tutorials/pca-analysis-r

Higgins, S. (2017). From $900 to $20,000: Bitcoin's historic 2017 price run revisited. *CoinDesk*. CoinDesk. Retrieved from https://www.coindesk.com/900-20000-bitcoins-historic-2017-price-run-revisited

HRAnalytics. (2017). Tutorial: How to assess model accuracy. Retrieved from https://www.hranalytics101.com/how-to-assess-model-accuracy-the-basics/

Hubens, N. (2018). Deep inside: Autoencoders. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f

Jain, S. (2018). An overview of regularization techniques in deep learning (with python code).

Analytics Vidhya. Retrieved from https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

Koehrsen, W. (2018). Why automated feature engineering will change the way you do machine learning. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/why-automated-feature-engineering-will-change-the-way-you-do-machine-learning-5c15bf188b96

Kostadinov, S. (2017). Understanding gru networks. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

Lopes, M. (2017). Dimensionality reduction - does pca really improve classification outcome? *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/dimensionality-reduction-does-pca-really-improve-classification-outcome-6e9ba21f0a32

Maladkar, K. (2018). Why is random search better than grid search for machine learning. *Analytics India Magazine*. Retrieved from https://www.analyticsindiamag.com/why-is-random-search-better-than-grid-search-for-machine-learning/?fbclid=IwAR3n4WLnSnPE5s6SHcJ3_clPmnxWcC_7PNR-qR7BRmM_0eqehjZrzV9OEik

Natekin, A., & Knoll, A. (2013). Gradient boosting machines, a tutorial. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3885826/

Nguyen, M. (2018). Illustrated guide to lstm's and gru's: A step by step explanation. Retrieved from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21/

Nwankpa, C. E., Ijomah, W., Gachagan, A., & Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. Retrieved from https://arxiv.org/pdf/1811.03378.pdf

Oehm, D. (2018). PCA vs autoencoders for dimensionality reduction. *Gradient Descending*. Retrieved from http://gradientdescending.com/pca-vs-autoencoders-for-dimensionality-reduction

Oshiro, T. M., Perez, P. S., & Baranauskas, J. A. (2012). How many trees in a random forest? Retrieved from https://www.researchgate.net/publication/230766603_How_Many_Trees_in_a_Random_Forest

Pascoal, C. (2019). Tutorial: Understanding linear regression and regression error metrics. *Dataquest*. Retrieved from https://www.dataquest.io/blog/understanding-regression-error-metrics/

Peetz, D., & Mall, G. (2017). Why bitcoin is not a currency but a speculative real asset. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3098765

Perervenko, V. (2016). Third generation neural networks: Deep networks. *MQL5*. MQL5.Community. Retrieved from https://www.mql5.com/en/articles/1103#1_2

Probst, P., & Boulesteix, A.-L. (2018). To tune or not to tune the number of trees in random forest. *Journal of Machine Learning Research*. Retrieved from http://www.jmlr.org/papers/volume18/17-269/17-269.pdf

Probst, P., Wright, M., & Boulesteix, A.-L. (2018). Hyperparameters and tuning strategies for random forest. Retrieved from https://www.researchgate.net/publication/324438530_Hyperpara

meters_and_Tuning_Strategies_for_Random_Forest

Rampurawala, M. (2019). Classification with tensorflow and dense neural networks. *Heartbeat*. Heartbeat. Retrieved from https://heartbeat.fritz.ai/classification-with-tensorflow-and-dense-neural-networks-8299327a818a

Rathi, A. (2018). Dealing with noisy data in data science. *Medium*. Analytics Vidhya. Retrieved from https://medium.com/analytics-vidhya/dealing-with-noisy-data-in-data-science-e177a4e32621

Ridgeway, G. (2007). Generalized boosted models: A guide to the gbm package. Retrieved from http://www.saedsayad.com/docs/gbm2.pdf

Ruder, S. (2016). An overview of gradient descent optimization algorithms. Retrieved from https://arxiv.org/abs/1609.04747

Skymind. (n.d.). A beginner's guide to neural networks and deep learning. *Skymind*. Retrieved from https://skymind.ai/wiki/neural-network

Tamuly, B. (2019). Bitcoin [btc]: Daily average block size hits an all-time high of 1.3 mb. *AMBCrypto*. Retrieved from https://ambcrypto.com/bitcoin-btc-daily-average-block-size-hits-an-all-time-high-of-1-3-mb

Taylor, D. H. (2018). Bitcoin: Hash rate says higher price. *Seeking Alpha*. Retrieved from https://seekingalpha.com/article/4208887-bitcoin-hash-rate-says-higher-price

Tuwiner, J. (2019). What is hash rate? *What is Hash Rate? 3 Things to Know (2019 Updated)*. Retrieved from https://www.buybitcoinworldwide.com/mining/hash-rate/

Upadhyay, Y. (2019). Feed forward neural networks. *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/feed-forward-neural-networks-c503faa46620

Walia, A. S. (2017). Activation functions and it's types-which is better? *Towards Data Science*. Towards Data Science. Retrieved from https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f?fbclid=IwAR3J4NfJ5USfzQsQqNWKS0bbeiHPa5x1xzcOZT4JYbHNx7szNY4JihQFu5Y

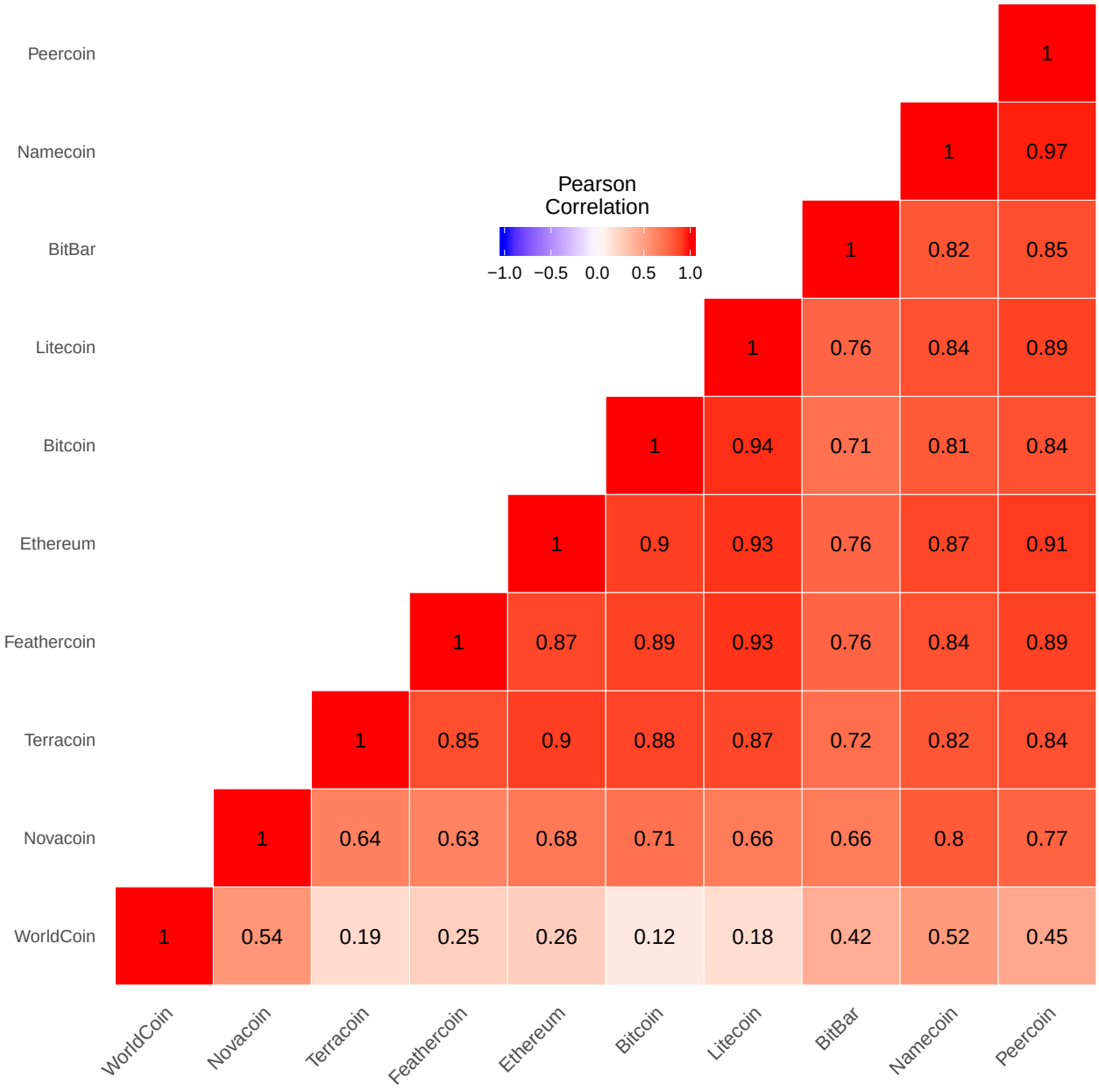
Weller, M. (2018). Recurrent neural networks for time series forecasting. *Novatec*. Retrieved from https://www.novatec-gmbh.de/en/blog/recurrent-neural-networks-for-time-series-forecasting/

Wolframm, A. (2019). Block size matters: How the bitcoin price signals what size is right. *Cointelegraph*. Cointelegraph. Retrieved from https://cointelegraph.com/news/block-size-matters-how-the-bitcoin-price-signals-what-size-is-right

Zamparo, L., & Zhang, Z. (2015). Deep autoencoders for dimensionality reduction of high-content screening data. Retrieved from https://arxiv.org/pdf/1501.01348.pdf

11 Appendices

11.1 Appendix A: Correlation Matrix



11.2 Appendix B: Training loss, simple dense

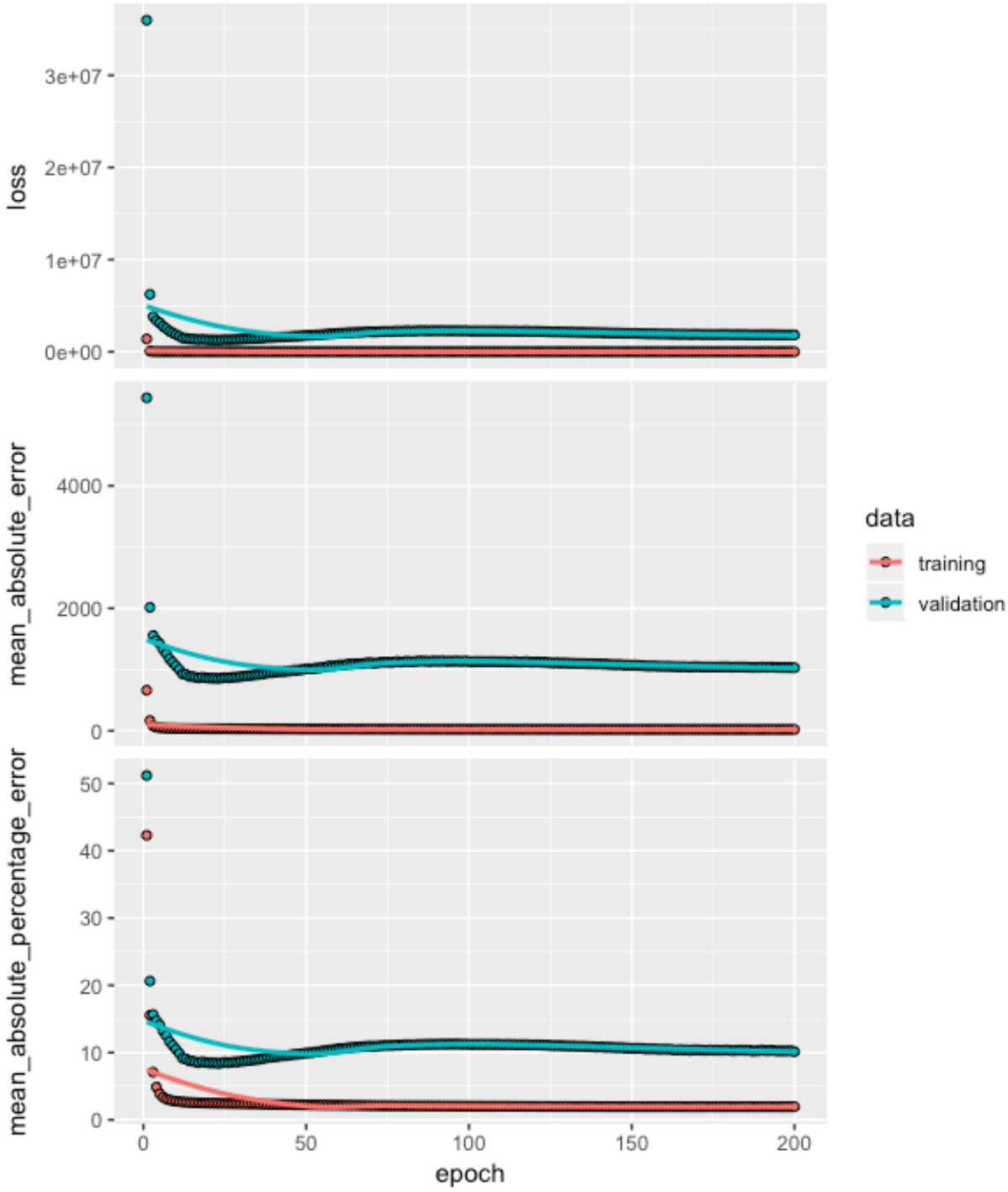


Figure 16: Training loss, simple dense