Norwegian University
of Life Sciences

Master's Thesis 2021 30 ECTS

Realtek
Ulf Geir Indahl

Non-ferrous metal price forecasting with Recurrent Neural Networks – How do they perform when forecasting multiple timesteps ahead?

Martin Bø
M.Sc Data Science

(This page is intentionally left blank)

Preface

This thesis concludes my eight years as an NMBU student after first completing a master's degree in business administration and now in data science. The data science program has been a great learning experience with both amazing teachers and classmates. I would like to thank my supervisor Ulf Geir Indahl for invaluable feedback on my thesis over the past few months. I would also like to thank Oliver Tomic and Kristian Hovde Liland for engaging and inspiring courses in machine learning which gave me the inspiration for this thesis. Finally, I would like to thank my family, friends, and my girlfriend Hedda for motivating and supporting me throughout my studies and during the pandemic in particular.

Abstract

This thesis aims to forecast the daily price of aluminum, copper and zinc from the London Metal Exchange five days ahead based on prices the previous five days using different recurrent neural networks. A “last-known observation” approach was used as a baseline for these models to beat which repeats the price at timestep five of the input data as the prediction for the next five days. Variables used for training and forecasting includes the prices of oil, gas, nickel, lead, tin, a US dollar index, aluminum, copper and zinc. Our results find that none of the single- or multi-layer LSTM or GRU models were able to out-perform the baseline model and in many cases the baseline significantly out-performed the recurrent neural network models. In general, the GRU models performed slightly better than the LSTM models, but not for all the metals. Further work could be done on multi-step commodity price forecasting by choosing a different time horizon or using intra-day data for a larger dataset. Other explanatory variables such as iron ore or coal could be included in the modeling and more complex networks such as the ResNet and LSTnet could be implemented.

Abbreviations

ABC – Artificial bee colony

ARIMA – Autoregressive integrated moving average

ARMA – Autoregressive moving average

EMD - Empirical Mode Decomposition

ETF – Exchange-traded fund

GRU – Gated Recurrent Unit

ICE – Intercontinental Exchange

ILZSG – International Lead and Zinc study group

LME – London Metal Exchange

LSTM – Long Short-Term Memory

LSTNet – Long- and short-term time-series network

MAE – Mean Absolute Error

MAPE – Mean Absolute Percentage Error

MSE – Mean Squared Error

NN – Feed-forward Neural Network

NYMEX - New York Mercantile Exchange

RMSE – Root Mean Squared Error

RNN – Recurrent Neural Network

SSA – Singular Spectral Analysis

VAR – Vector autoregression

VMD - Variational Mode Decomposition

WTI – West Texas Intermediary

Table of figures

Table of contents

1. Introduction

This thesis will consider short term multi-step metal price forecasting using recurrent neural networks (RNN). The commodity prices are gathered from Thomson Reuters Datastream (Datastream, 2021) at NMBU which collects data from various exchanges such as London Metal Exchange (LME). Aluminum, Copper and Zinc are the metals that will be forecasted as they are the most traded non-ferrous metals (LME, 2021).

The motivation for the thesis is to determine if RNNs are suitable to forecast prices multiple days ahead and if the results vary significantly between various metals and the different model architectures. More specifically, we focus on forecasting the price of various metals five days ahead based on the price the previous five trading days.

Metal prices are reported daily which means the interval between each observation will be one day and it is the daily close price that will be used. A direct multi-step ahead forecast will be utilized where all the predictions (five days ahead) are made simultaneously. This could also be done in a feed-back or autoregressive fashion but will not be implemented in this paper.

There are various interested parties when it comes to forecasted prices of the mentioned metals. Speculators that trade in short-term *futures contracts*, which is an agreement between two parties to buy or sell a quantity of for example a commodity at a certain time in the future at a predetermined price (Carter, 2012), can benefit from price forecasts if they deviate from the price in the market. This can lead to profits if the model forecasts the price better than the futures contracts present in the market. If this methodology is effective for even longer-term forecasting, producers and consumers of the metals can use this information in their hedging strategies where they can utilize long or short contracts to hedge their risk.

The present thesis will touch on a variety of different subjects, but the focus will be on Data Science and related methodology. Both econometrics, finance and commodities will not be studied in dept, but an introduction and definitions will be given where it is deemed necessary.

The thesis is structured as follows – Chapter 2 will include background information, relevant literature references, and a description of the dataset together with some relevant descriptive statistics. Chapter 3 will describe the methodology and theory used in the thesis, which is divided in to six parts. The first part describes a common workflow in machine learning, the second part covers sequential data and timeseries while the third part introduces a forecast

baseline. The fourth part describes feed-forward neural networks, the fifth part goes in dept on different recurrent neural networks and final part covers various performance metrics. Chapter 4 presents the results from the modeling, chapter 5 will compare the results and discuss them and chapter 6 will conclude the thesis.

2. Background and theory

In the following we describe some introductory financial- and commodity theory, including price drivers for the target commodities. We also include references to literature on forecasting in commodity markets with emphasis on data scientific approaches including descriptive statistics and some explorative data analysis.

2.1 Non-ferrous metals as financial commodities

Similar to stocks, exchange-traded funds (ETFs) and other financial derivatives, commodities can be traded on exchanges. A derivative can be defined as a financial contract where the value is linked to (or derived from) the value of an underlying asset (Carter, 2012). The underlying of these derivatives can be a commodity as considered in this thesis. There are often several different interested parties when it comes to dealing with commodities as a financial instrument. Speculators seeking profitable returns on their investments as well as consumers and producers that have an interest in forecasting the price of these commodities. Such contracts are traded on exchanges such as the London Metal Exchange (LME) and the Intercontinental Exchange (ICE) and will be the main exchanges used in this thesis. The speculator will commonly use futures or options contracts when making trades. A futures contract can be defined as an "obligation to buy or sell a specific quantity and quality of a commodity or financial instrument at a certain price on a specified future date" (Carter, 2012) while an option can be defined as an "option to buy or sell a specific quantity and quality of a commodity or financial instrument at a certain price on a specified future date (Carter, 2012).

For example, if trader A agrees on a futures contract with trader B where 100 barrels of *crude oil* will be delivered from trader A to trader B in six months at a price of 50$ per barrel. Both parties are bound to this contract and if the price in the market increases to for example 75$, trader B would have made a 25$ profit per barrel while trader A would have made a 25$ loss per barrel. Contracts like these can either have physical delivery on the underlying commodity or a financial settlement, where speculators mostly trade in cash-settled contracts while consumers and producers trade in both types of contracts. Options works in a similar fashion to futures contracts, but the contract does not have to be exercised and the issuer of the contract gets paid a premium by the counterpart for the right to exercise the option. A *call option* (also called a buy option) is an option where one party has the right but not an obligation to buy a commodity at an agreed upon price before a predetermined date. If the price in the market is lower than the agreed upon price, it would result in a loss to exercise the option and would therefore not be exercised. However, if the price of the commodity in the

market is higher than the agreed upon price (called strike), exercising the contract would yield a profit of the difference between the market price and the strike price minus the premium paid for the contract.

While the abovementioned contracts consider a trade at some point in the future, one can also trade commodities in the present at the spot market. In the spot market trades are settled continuously and the spot price represents the price of the commodity when taking delivery immediately in some ways like purchases in for example the grocery store. For simplicity, this thesis will only consider spot prices and indices on the various commodities to avoid problems with settlement and rolling futures contracts.

The dataset used in the thesis includes timeseries with prices or indices of

- Crude oil
- Brent oil
- Nickel
- Lead
- Tin
- US dollar
- Natural gas
- Zinc
- Copper
- Aluminum.

All the data has been acquired from Thomson Reuters Datastream which the School of Economics and Business at NMBU has licenses for and includes daily observations between 31.12.1994 and 01.01.2020. This equals 6524 rows of data for ten different features. Crude oil is the West Texas Intermediate grade oil delivered at Cushing Oklahoma, traded at the New York Mercantile Exchange (NYMEX) and denoted in USD per barrel. Brent oil is the oil produced in the North Sea traded at the Intercontinental Exchange and is also denoted in USD per barrel. Natural gas is represented by the S&P GSCI index of natural gas which starts at 100 (S&P, 2021). An index will catch the price changes but not the prices themselves at a particular time, but since the timeseries data will be normalized prior to the analyses this does not matter.

The ferrous metals, i.e., Nickel, Lead, Tin, Zinc, Copper and Aluminum are traded at the London Metal Exchange in dollars per metric ton. There are also various requirements such as purity the metals need to have which are specified in the contracts at LME.

The USD index tracks the dollar value compared to several other currencies. When the dollar gains value compared to other currencies the index raises and vice versa. Euro, Yen, Pounds, Canadian Dollars, Swedish Krona and Swiss Franc are the currencies the index is based on (ICE, 2020).

2.2 Price drivers

As most things, prices on commodities are driven by supply and demand, where a price is given by the intersection between the supply- and demand curve. To be able to find good variables for forecasting the prices of aluminum, copper, and zinc, one needs to look into what drives the supply of the commodity and what drives the demand. Since this thesis is written in a data scientific perspective, we will not go much more in dept on the economic and financial theory behind various price drivers but instead look at the fundamentals that can affect the supply or demand side of a commodity. In short this means that if the supply is low and the demand is high the prices will rise and vice versa. The production process of the metals will not be covered in detail as it is outside the scope of this thesis, but a brief introduction will be given.

2.2.1 Aluminum

In production of Aluminum the main costs are related to the use of electricity when transforming bauxite to pure aluminum. Additional costs are due to labor, raw materials, and shipping. Since the price of electricity varies from country to country and region to region it is hard to use the electricity price itself as input in the models. However, other energy commodities are often highly correlated with the price of electricity since products like oil and natural gas are used in electricity production. Therefore, oil and natural gas prices will be included in the modeling as a proxy for the electricity price.

When it comes to the demand side, aluminum is mostly used in the industry. Some of the main industries include automobiles, construction, packaging and aviation. Aluminum is traded in USD worldwide and therefore companies buying or selling aluminum may face significant currency risk. An important variable to model the aluminum price could therefore be the US dollar. If the dollar price increases compared to a local currency, the producer will

profit from this increase while the consumer will have a loss and vice versa for a decrease in the price. Therefore, US dollar seems to be an important variable for modeling the aluminum price.

There exists a vast number of aluminum alloys, where aluminum is the predominant metal and copper, magnesium, silicon, zinc, tin etc. could be the alloying metal. In microeconomic terms these other metals could be considered complementary goods (hot dogs and hot dog buns) and therefore the price of these other metals could affect the price of aluminum. This does not only occur with alloys of aluminum, but also when aluminum is used together with other goods and commodities such as polymers, carbon fiber, steel, titanium etc. in for example airplanes. Therefore, also other metals and fabrics can be used as explanatory variables for forecasting the aluminum price.

2.2.2 Copper

Copper is a metal with electrical and thermal conductive power and as shown in figure 1 43% of copper is used within building construction, 21% in electrical and electric products, 19% in transportation equipment, 10% for consumer and general products as well as 7% for industry machinery and equipment (ILZSG, 2021).

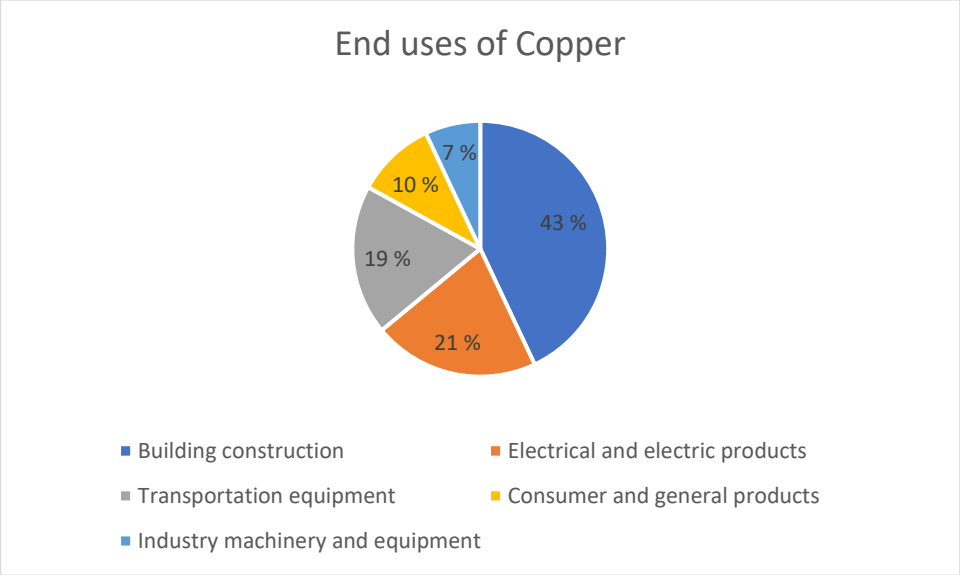


Figure 1 - End uses of copper (ILZSG, 2021)

Copper is extracted from copper ore through melting at high temperatures, which similarly to zinc and aluminum require a high amount of energy. As with the other metals, energy is a key input in production and could therefore be a good explanatory variable. As described in chapter 2.2.1, the currency risk rationale also applies for copper, and therefore the USD can also be a good variable to include for modeling the price of copper.

Copper is used in many of the same sectors as aluminum and zinc, so other metals such as lead, tin, nickel, zinc and aluminum make sense to include.

2.2.3 Zinc

When it comes to production of zinc the main costs are similar to aluminum and copper where costs related to mining (labor, machines, fuel, electricity) and refining (electricity). In this case as with the other metals, we therefore consider petroleum products as a proxy for electricity costs as well as a direct cost for fuel when operating machines used in the mining process.

Around 50% of all zinc that is produced in the world goes to galvanization which is to coat iron or steel with a protective layer of zinc. Products galvanized with zinc are used in construction, industry, automobiles etc. Zinc alloys accounts for 17% of the consumption as do brass and bronze (ILZSG, 2021).

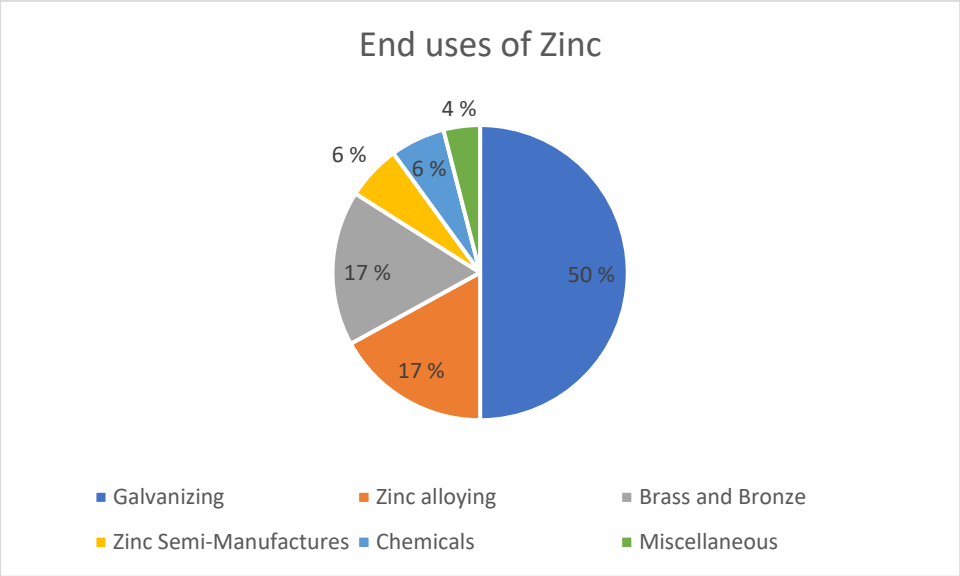


Figure 2 - End uses of zinc (ILZSG, 2021)

As with aluminum and copper, the same currency risk applies for production and consumption of zinc and can therefore be a good explanatory variable for modeling zinc prices as well.

The main take-aways from the complementary- and substitute goods apples for zinc as well, hence the price of copper, aluminum and other metals will be relevant for forecasting the zinc price.

2.3 Literature review

In this section previous literature of commodity price forecasting will be covered. The focus will be on data scientific approaches but a brief coverage of papers within the field of econometrics and finance will be covered as well. Other methods for commodity price forecasts such as microeconomic theory models, input-output, mathematical programming and computable general equilibrium models also exists (Labys, 2006), but will not be covered as they are not within the scope of this thesis.

Lehfeldt (1914) was among the first that analyzed commodity prices and the interactions with their demands using statistical methods such as regression. Slutsky (1927) and Yule (1927) used simple linear differential equations influenced by stochastic shocks to forecast and model a vast number of time series within economics. The equations used by Slutsky and Yule explained autoregressive processes where a variable was predicted based on previous values of that variable. This autoregressive approach was formalized by Box et al. (1970) where they introduced the autoregressive integrated moving average (ARIMA) model. A methodology for timeseries forecasting referred to as the Box-Jenkins method was introduced and starts with a model identification, estimation of parameters, validation and finally predictions.

The volatility of copper and aluminum based on three months futures as well as spot prices were analyzed by Figuerola-Ferretti and Gilbert (2008). A bivariate FIGARCH model was applied, which is a fractionally integrated generalized autoregressive conditional heteroscedastic (FIGARCH) model that describes the persistence of volatility in a timeseries (Tayefi & Ramanathan, 2016). The volatility in these metal prices showed a long-term memory process which means that the price changes from day-to-day are related.

Ahti (2009) applied nonlinear models to data on non-ferrous metals from London Metal Exchange in the period from 1970-2009. He used a Smooth transition autoregressive model (STAR) and a feed-forward artificial neural network as his non-linear models and compared the results with linear models such as random-walk and autoregressive moving average (ARMA) models. The results presented was based on an out-of-sample evaluation for daily, weekly and monthly data. For the weekly and monthly data, he found negligible differences in performance between the linear and nonlinear models for all metals besides tin.

Malliaris and Malliaris (2009) implemented an artificial neural network (NN) to forecast the price and analyze the interactions between gold, oil and the euro. A long- and short-term

relationship was observed between the commodities, and they also found that oil had the biggest impact on the other commodities.

A decision tree algorithm was applied by Liu et al. (2017) on a dataset from London Metal Exchange containing copper prices and the authors claims this was the first time decision trees was used for copper price predictions. Their predictions were accurate in both the short and long term and scored a mean absolute percentage error (MAPE) under 4%. Decision trees has the advantage that that it is not necessary to set assumptions regarding for example stationarity, cointegration and the Gauss-Markov assumptions that apply for linear regression (Theil & Collection, 1971)

So far in this literature review the focus has been on econometric methods performed on commodity data sets. Bandara et al. (2021) covers several papers using various RNN over the years, and they find that even though RNNs were used already in the 1990s, their use did not drastically increase until around 2015. The next few paragraphs will cover different uses of various RNN models used in various contexts.

Jue et al. (2019) analyzed agricultural commodities as they have very complex price formations and prices are difficult to forecast. Agricultural commodities, like most other commodities, display nonlinear characteristics and the authors used a three-part approach to this problem. First, they used three different denoising techniques specifically singular spectral analysis (SSA), empirical mode decomposition (EMD) and variational mode decomposition (VMD). This was done to remove external noise in the time series. As a secondary step, they combined these denoising techniques with forecasting models such as autoregressive integrated moving average regression (ARIMA), support vector regression (SVR), recurrent neural network (RNN), gated recurrent neural network (GRU) and long-short term memory RNN (LSTM). As the third step, the artificial bee colony algorithm (ABC) mentioned in the title of the paper, was used to forecast heterogeneous, semi-heterogeneous and homogeneous combinations. Their results indicated that the semi-heterogeneous forecast combination performed better than the other two combinations.

Ouyang et al. (2019) forecasted global agricultural futures prices from multivariate time series. Their dataset included a combination of long- and short-term information as well as both linear and nonlinear data structures. Therefore, traditional methods within the field of econometrics such as autoregressive integrated moving average (ARIMA) and vector autoregression (VAR) are not suitable as they struggle with nonlinear data. Instead, a Long-

and Long- and Short-Term Time-series Network (LSTNet) was used. This method is based on the Lai et al. (2018) paper "modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks" and applies a combination of a convolutional layer, a recurrent layer, and a recurrent skip. This method is also autoregressive, meaning it takes the output of one timestep as input in the prediction of the next timestep. The first convolutional layer "*aims to extract short-term patterns in the time dimension as well as local dependencies between variable*" (Lai et al., 2018) according to the authors. The recurrent component has a Gated Recurrent Unit (GRU) and uses RELU as the activation function. Their results indicated that the LSTNet performed better than their baseline RNN, CNN, ARIMA and VAR models, especially as the time horizon extended further into the future.

2.4 Descriptive analysis of the data

In figure 3 the price development for aluminum, copper and zinc on the London Metal Exchange (LME) from 30.12.1994 to 01.01.2020 is shown. For aluminum, the price has been fluctuating between 1136 and 3271 dollars per metric ton with a mean around 1800. Before the financial crisis in 2008 we can see a rapid increase in the prices of the commodities followed by a sharp decrease during and after the financial crisis. Some of the same characteristics seems to be apparent for both copper and zinc as well where both had a significant price increase before the financial crisis and a similar price decrease during and after the financial crisis. For copper one can see a quick recovery sending the price close to 10000 dollar per metric ton around 2012 before a steady decrease towards 5000 dollars in 2016. The price for zinc seems more stable than the two abovementioned metals, as it had fairly flat development from 1994-2004. Similar to the other metals a price spike followed by a decline was apparent around the financial crisis and reached a bottom around 2009 around 1200 dollars per metric ton. From around 2010 the price has been hovering around 2000 dollars per metric ton with a top in 2018 around 3000 dollars.

For all the metals in figure 3 the price movements seem erratic and has no clear direction or trend over a longer period of time.

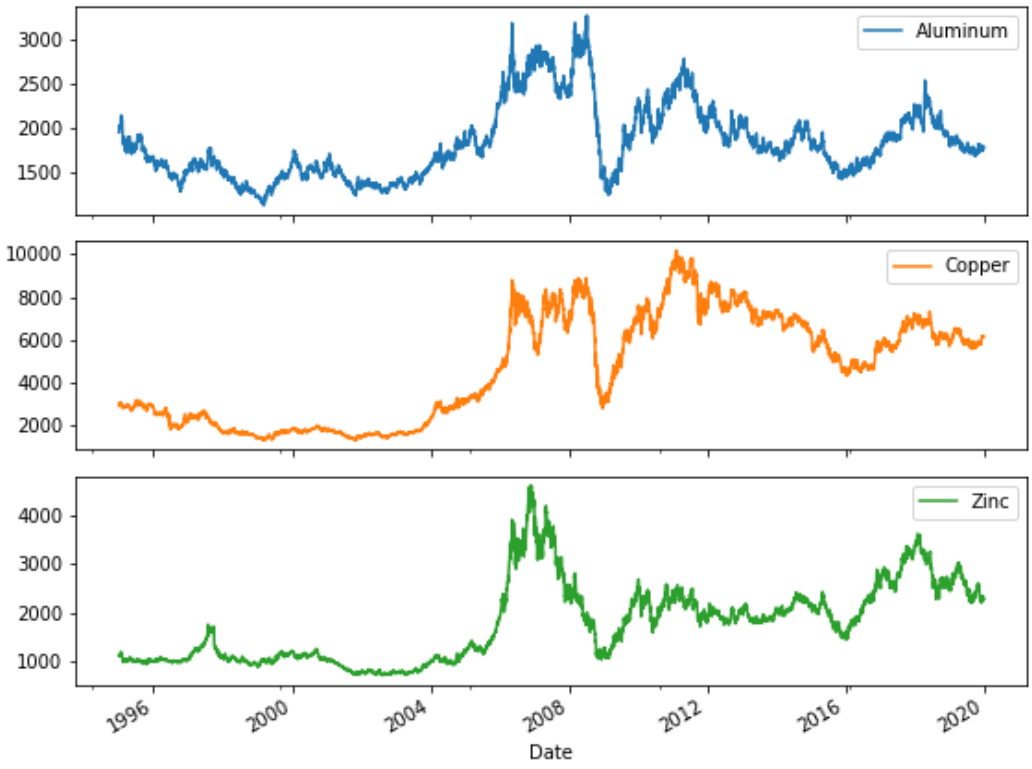


Figure 3 - Price development 1995-2020 for aluminum, copper and zinc

Some commodities have seasonal characteristics in their prices due to for example weather and therefore two new features called “Time of year signal” has been included. They are represented using the sine and cosine functions together with the Pandas function to_datetime (McKinney & Others, 2010) and the datetime library in python. The signals for the first 250 observations, which is roughly equal to one year in the financial markets can be seen in figure 4.

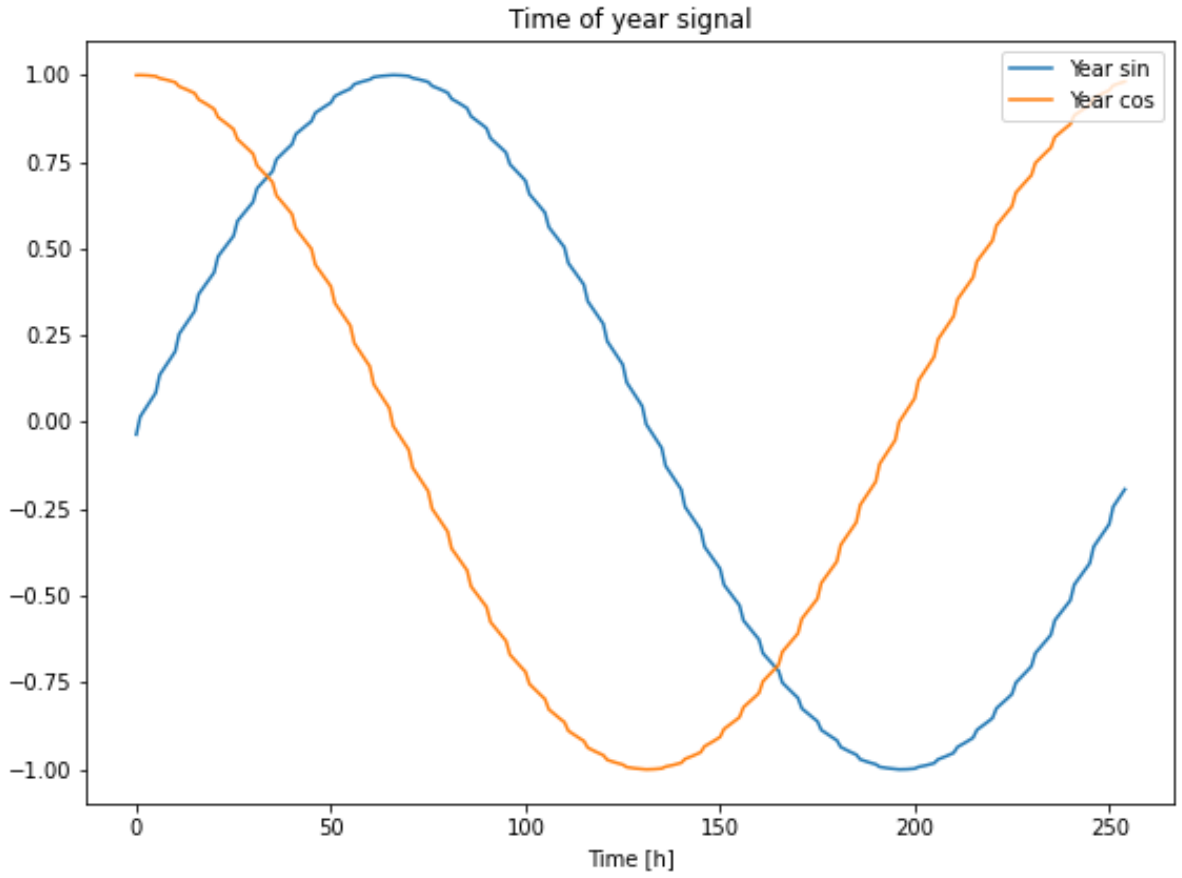
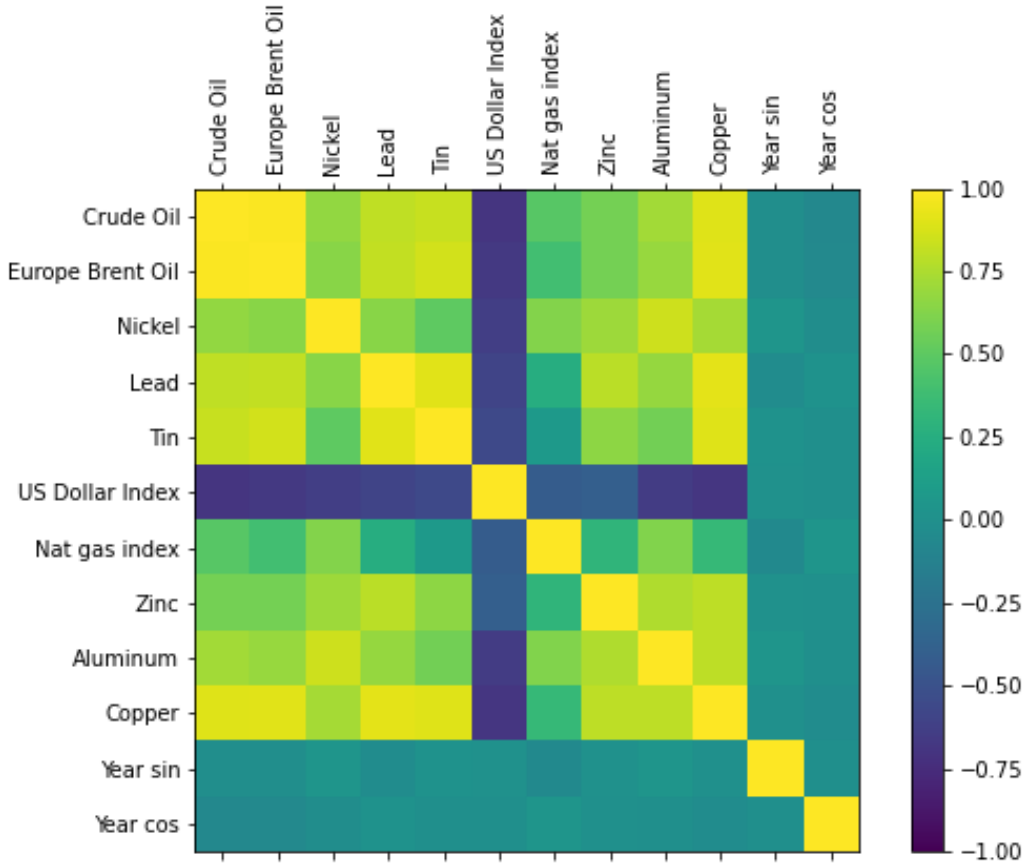


Figure 4 - Time of year signal for approximately one year in trading days

Figure 5 shows the correlations between the features included in this thesis. As noted earlier in this chapter, the non-ferrous metals are all relatively highly correlated with each other where for example the correlation coefficient between copper and lead is 0,92. Copper is also highly correlated to the oil products (about 0,9) and as well as the other metals above (about 0,73). Aluminum has the highest correlation with nickel (about 0,85) and zinc (about 0,76). The correlation between aluminum and the brent and crude oil price are also high (about 0,7), slightly lower for natural gas (about 0,62). Zinc shows the highest correlation with nickel and lead at about 0,71 and 0,79 respectively, while its correlation to the energy commodities is lower (oil is about 0,58 and natural gas about 0,31). Other interesting findings in the

correlation matrix is that crude and brent oil are almost identical with a correlation coefficient of 0,99, while the correlation to natural gas is below 0,5.

As described in chapter 2.2, the USD could be a price driver for the various commodities as it has an inverse relationship with commodities that are being exported. An increase in the US dollar compared to other currencies leads to an increased cost for consumers to buy commodities because it is traded in US dollars and vice versa (Ganapathyraman et al., 2018). This is seen in our dataset as well, where the USD index has a negative correlation with all the commodities where the strongest negative correlation is to crude oil and copper at 0,69.



	Crude Oil	Europe Brent Oil	Nickel	Lead	Tin	US Dollar Index	Nat gas index	Zinc	Aluminum	Copper	Year sin	Year cos
Crude Oil	1,00											
Europe Brent Oil	0,99	1,00										
Nickel	0,67	0,64	1,00									
Lead	0,81	0,81	0,65	1,00								
Tin	0,83	0,86	0,50	0,91	1,00							
US Dollar Index	-0,69	-0,67	-0,63	-0,59	-0,56	1,00						
Nat gas index	0,48	0,39	0,63	0,24	0,08	-0,42	1,00					
Zinc	0,58	0,59	0,71	0,79	0,66	-0,39	0,31	1,00				
Aluminum	0,72	0,69	0,85	0,68	0,57	-0,64	0,62	0,76	1,00			
Copper	0,90	0,91	0,73	0,92	0,90	-0,69	0,34	0,80	0,80	1,00		
Year sin	-0,01	-0,01	0,05	-0,02	0,02	0,01	-0,05	0,01	0,05	0,00	1,00	
Year cos	-0,06	-0,06	-0,02	0,02	-0,01	0,00	0,04	0,00	0,00	-0,03	0,00	1,00

Figure 5 - Correlation matrix plot and lower correlation matrix

Figure 6 shows descriptive statistics using the .describe() function on the dataset which is stored in a pandas DataFrame (McKinney & Others, 2010). From the *mean* column it is clear that these variables are on different scales and needs to be normalized before modeling. Because of the differences in scale, the standard deviation also differs by a lot, which means that the data needs to be adjusted for both the mean and standard deviation. This is done by subtracting the mean and dividing by the standard deviation from the training data.

	count	mean	std	min	25%	50%	75%	max
Aluminum	6524.0	1837.551916	415.886822	1136.20	1516.0000	1764.500	2051.0625	3271.25
Copper	6524.0	4704.419517	2495.610261	1318.25	1997.8750	4865.625	6937.6250	10179.50
Crude Oil	6524.0	53.223506	29.094225	10.73	26.9950	50.045	73.9700	145.66
Europe Brent Oil	6524.0	55.142147	32.842935	9.10	25.5075	51.570	75.4275	143.95
Nickel	6524.0	13723.657051	7933.837224	3730.50	7910.7500	12157.500	16842.5000	54050.00
Lead	6524.0	1421.740877	795.374039	400.75	595.3750	1576.000	2087.0000	3989.00
Tin	6524.0	12939.499448	7307.129729	3601.00	5710.0000	12315.000	19650.0000	33265.00
US Dollar Index	6524.0	91.199117	10.901071	71.33	82.1800	89.525	97.9825	120.90
Nat gas index	6524.0	328.799801	171.396318	125.11	198.1675	281.665	392.8700	927.93
Zinc	6524.0	1773.613177	805.233121	722.75	1044.4500	1776.250	2292.0625	4603.00

Figure 6 - Descriptive statistics on the raw data

The violin plot in figure 7 shows the distribution and some summary statistics for the dataset. The white dot in the middle of the plot represents the median value of the dataset, the vertical height represents how wide the distribution is and the horizontal distance within each plot shows the frequency of observations at that point. From the figure it is clear that especially nickel, aluminum and lead have the most extreme high values but most of the variables still have distributions that differ from the gaussian distribution. Lead, Tin and copper appears to have similar distributions and the same goes for crude oil and brent oil. These observations match the conclusions drawn from the other figures in the chapter.

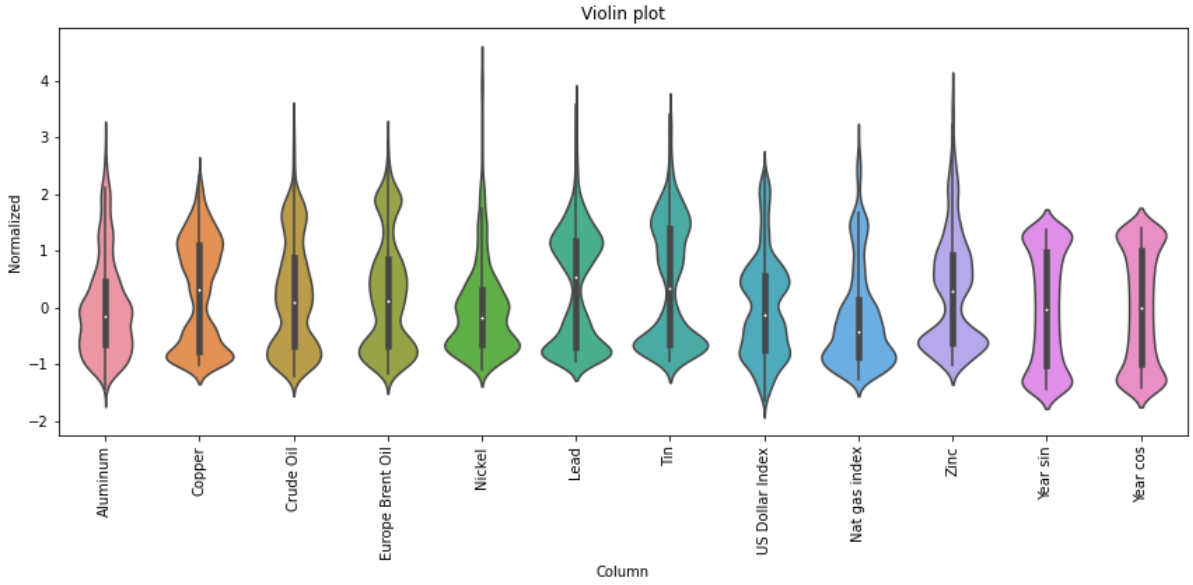


Figure 7 - Violin plot showing the normalized data distributions

3. Methodology

This chapter will cover the methods chosen for forecasting the present work. We start with a description of sequential data and how the dataset is split up into shorter sequences representing inputs and targets. Thereafter we discuss a forecast baseline to be used as a comparison to the various models in this thesis. For comparison with the baseline model, we describe a vanilla feed-forward neural network (NN) to be used as a first model approach before the main approach based on recurrent neural networks (RNN) are described. The various methods presented in this chapter are implemented in python through the Keras (Chollet & Others, 2015) and TensorFlow (Martín Abadi et al., 2015) libraries, but other libraries such as Pandas (McKinney & Others, 2010), NumPy (Harris et al., 2020), Matplotlib (Hunter, 2007) and Seaborn (Waskom et al., 2017) have also been used.

3.1 Universal workflow in a machine learning process

The workflow presented here is based on chapter 4.5 in the book "Deep learning with python" by Chollet (2017).

1. **Defining a problem and preparing a dataset**

The first step in this process is to identify a problem and to prepare a dataset. The focus in this thesis is a regression problem where a timeseries dataset have been acquired with the variables described in chapter 2. Regression problems within finance are often static forecasts one timestep ahead. As an extra challenge that can provide a bigger benefit this thesis will forecast prices multiple days ahead. The focus in this thesis is to search for models that can predict multiple steps (days) ahead that when successful gives more value to the forecasts as described in the motivation of this thesis in chapter 1. Domain specific knowledge has been discussed in chapter 2.2 to identify relevant data to be included in the forecasts. It is important to acknowledge that using historical data to predict the future outcomes in timeseries modeling assumes that the response(s) behave consistently with the available historical data, which is not necessarily always the case.

2. **Choosing a measure of success**

In a machine learning process, one needs to define what is meant by a successful model, and how to measure success. In the present work, the candidate models will be compared to a baseline model using a "last-observation approach". This approach will be described in further detail in chapter 3.3. Our definition of success is if some more complex model

alternative (such as RNN) can outperform the baseline and simpler models. This leads to the following question: *"Is a more complex model necessarily a better model?"*.

Any measure of success requires some precise measurement of model accuracy. Because we consider a regression problem (timeseries forecasting) in this thesis, various forecast error metrics will be discussed in chapter 3.6.

3. **Evaluation protocol during training**

We also need to define a procedure for monitoring the training process of our models. Examples of this is *k-fold cross validation* which is useful when you have a low number of samples, *iterated k-fold with shuffling* which is used for performing highly accurate model evaluation when the dataset is not large enough to *maintain a hold-out validation set*, which is the method that will be used in this thesis.

4. **Pre-processing and visualizing data**

Pre-processing and visualization of data is a crucial step in a machine learning process. This lays the foundation for further model selection, training and scoring. According to Anaconda (2020), about 2/3 of the time spend in a data science project is spent on data preparation and visualization.

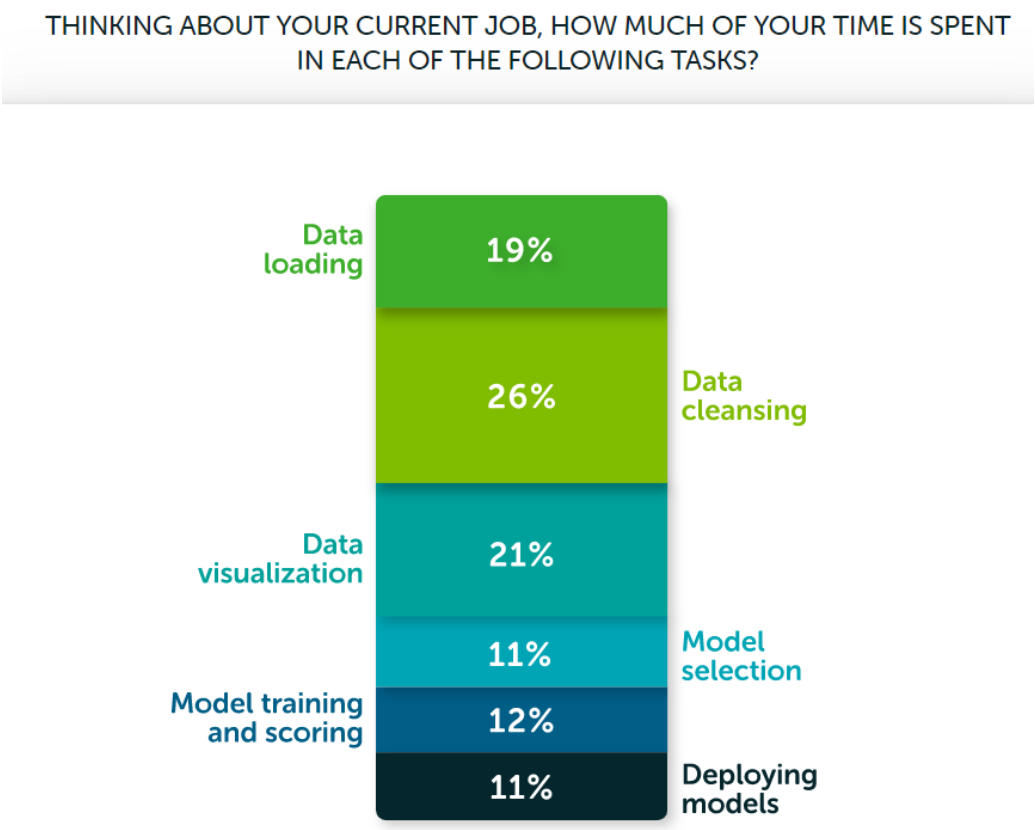


Figure 8 - Time spent working on different tasks in Data Science (Anaconda, 2020)

The dataset we analyze in this thesis has been acquired from Thomson Reuters Datastream (Datastream, 2021) on a standard timeseries format with variables in the columns and the price at time t in the rows. Since all the data is gathered from the same source, all the data is therefore on the same format which simplifies the data cleaning process. Some important steps is to identify possible missing values and outliers, visualize the distribution and identify relationships between different variables (see chapter 2.4). Feature engineering can also come in handy to create new useful features. We will consider catching time-specific information with a "time of year signal" as described in chapter 2.4.

5. Developing a model that beats the baseline

After defining a baseline model for the problem, the next step is to choose and train a model that hopefully performs better than the baseline. For a regression problem the choice of baseline tends to be more problem-specific depending on the characteristics of the regression. For problems with timeseries data one needs to consider how many steps ahead that is being predicted. In commodity price forecasting, the prices from one day to another does not change drastically so a simple choice of a baseline model could be to use the last known price for

predicting the price at the next timestep. This will be the choice of baseline model that we will use in the present work. Performance of the baseline will be measured by the metrics mentioned in step 2. Further details are described in chapter 3.6.

When it comes to what models to choose, one needs to consider what kind of data that one has access to and what kind of problem at hand. It is often a good idea to start with a simpler type of model that is easy to implement and interpret. This is so that the researcher does not have to spend unnecessary time developing and tuning a very complex model if the problem can be solved by for example a linear regression. Since we are working with a regression problem using timeseries data it is logical to consider models that can process data sequentially and has a concept of time. Therefore, RNNs are the topic of this thesis.

6. **Scaling up – Increasing the capacity of the model**

If the simple model approach from step 5 was unsuccessful it is time to consider if the model is sufficiently complex and powerful. If this is not the case, one should increase the capacity of the model by for example adding layers, adding nodes or units as well as increasing the number of training epochs. When increasing the complexity of the model, one should monitor the loss and validation loss of the model and see when the model starts to overfit, resulting in poorer predictions for the validation data compared to the training data.

7. **Regularization and hyperparameter tuning**

In the case of overfitting, some inclusion of regularization and further hyperparameter tuning for the training process must be considered. A good model should be based on a compromise between capturing the essential patterns in the data without overfitting to the training data. This can be achieved with regularization parameters such as L1 and L2 in for example Lasso, Ridge and Elastic Net regression (Raschka & Mirjalili, 2017). Inclusion of the dropout strategy for regularization to prevent overfitting will be discussed further in chapter 3.5 about Recurrent Neural Networks. The dropout method is also a common technique used for fully connected dense networks. Other possibilities like adjusting learning rate, increasing or decreasing the number of nodes, and increasing or decreasing the number of hidden layers is commonly used in this step.

3.2 Sequential data and timeseries

This thesis will use sequential data and the subset timeseries for forecasting purposes, which has different characteristics than for example the famous MNIST (LeCun & Cortes, 2010) and Boston Housing dataset (Harrison & Rubinfeld, 1978) which are much used for illustrating multiclass classification and regression (without time information) problems. A unique characteristic of sequential data is that the observations appear in a certain order which means they are dependent on each other. These dependencies need to be taken into consideration when facing a forecasting problem.

Raschka and Mirjalili (2017) illustrates the main types of sequence modeling in figure 9

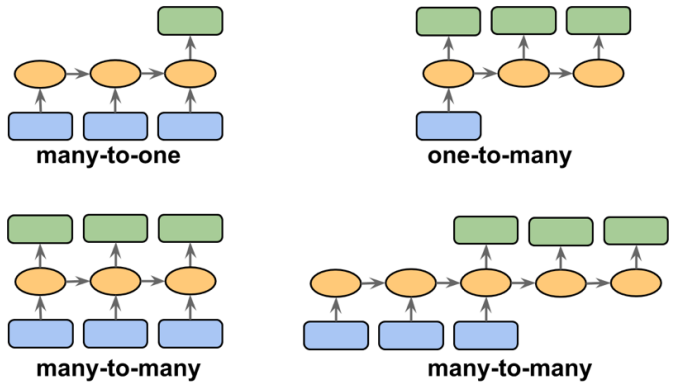


Figure 9 - Different types of sequence modeling (Raschka & Mirjalili, 2017)

where many-to-one refers to a situation where the input data is a sequence, and the output is a fixed size vector and not a sequence. An example of this can be sentiment analysis where the input is a text, and the output is a classification label. One-to-many is a situation where the input data is a standard format (not a sequence) and the output is a sequence. An example of this can be captioning of images where the inputs are images, and the output is a text describing the content of the image. Many-to-many sequence modeling comes in two variations, synchronized and delayed. An example of synchronized many-to-many modeling can be video classification (sequence of images) where every frame (image) is classified at various timesteps. The delayed many-to-many approach is similar to the synchronized many-to-many, but the predictions are offset in time. An example of this can be commodity price forecasts where the inputs are historical prices and the outputs are future prices for multiple timesteps.

The latter is the topic covered in this thesis. Here both the inputs and outputs of the model are sequences, where an input sequence has multiple dimensions (features) while the corresponding output is a single feature sequence.

For making multiple predictions over several timesteps, it is necessary to split the dataset into smaller sequences. In forecasting one is often interested in predicting some outcome a certain number of timesteps ahead (the output) based on the information from a certain number of previous timesteps (the input). The number of input timesteps will be referred to as the input width and the number of output timesteps will be referred to as the target width, while the difference between the two will be referred to as the offset, i.e., synchronized or delayed many-to-many described in the previous paragraph. An illustration is presented in figure 10 where the input width is the first six timesteps and the label width is the seventh timestep with a width of one. The offset is one which means a delayed many-to-one approach since the prediction is only one timestep ahead.

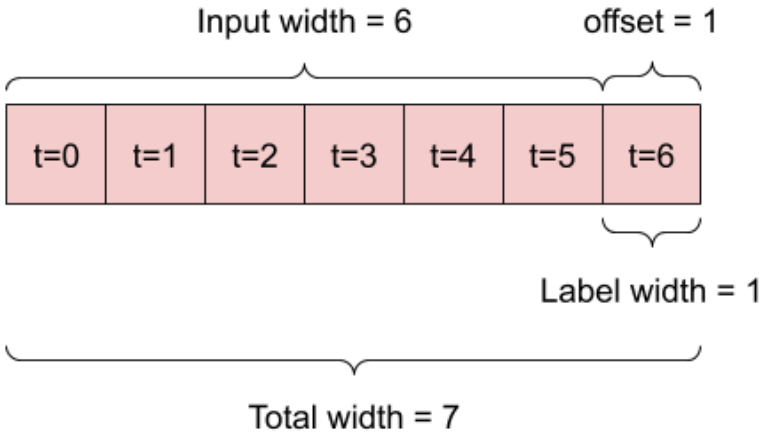


Figure 10 - Data sequence with input width 6, offset of 1 and a label width of 1 (Tensorflow, 2021)

Figure 11 shows how the sequence is split into inputs and targets or labels with the same information as in figure 10. The red array shows the original indices of the entire window, the blue array shows the inputs after the split and the green box shows the target index. This window can be adjusted to fit the goals of the problem at hand, and the specific window used in this thesis will be five timesteps (days) as input width and five timesteps (days) as target width with an offset of five.

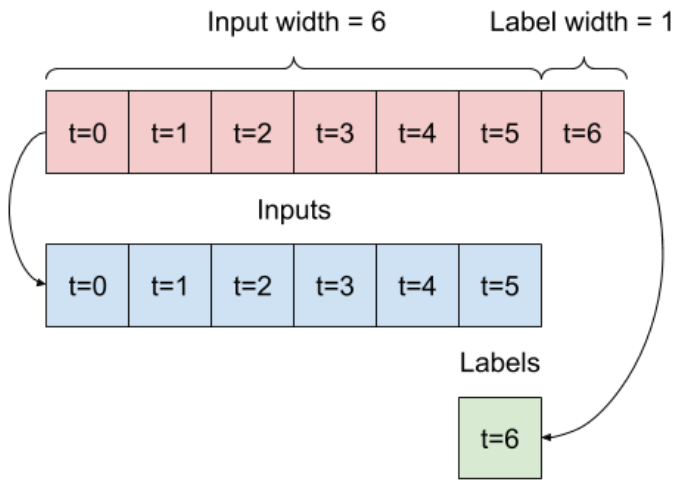


Figure 11 - Data sequence split into inputs and label or target (Tensorflow, 2021)

3.3 Forecast baseline

As described in chapter 3.1 one needs to define a forecast baseline that suits the problem at hand. Since timeseries data is order-dependent and we have seen in chapter 2.4 that the data rarely changes very much from one timestep to another, a reasonable prediction could be that the price at time t+2 equals the price at price t+1 for a single-step prediction. Figure 12 shows an example for a data window of total width of 48, where the input width is 24 and the output width is 24. I.e. the previous 24 timesteps are taken as input to predict the following 24 timesteps. The last known observation at timestep 24 is then chosen as the prediction the next 24 days shown by the crosses in the figure. The true values or targets are the green dots, and the performance of this baseline approach can be calculated with a variety of performance metrics as will be discussed in subchapter 3.6. However, in this particular example the mean absolute error (MAE) was used, meaning a mean absolute error of 0,0934 should be the minimum requirement for a more complex model to beat in this example.

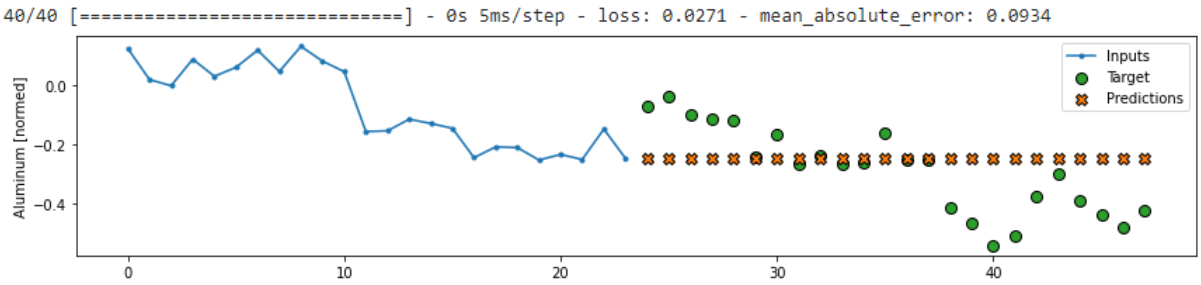


Figure 12 - Plot of a sequence with 24 as input and 24 as output with an offset of 24

3.4 Feed-forward Neural Network (Deep Neural Networks)

A neural network can be considered an extension of the Adaptive Linear Neuron algorithm (Adaline) (Raschka & Mirjalili, 2017) which can be seen in figure 13.

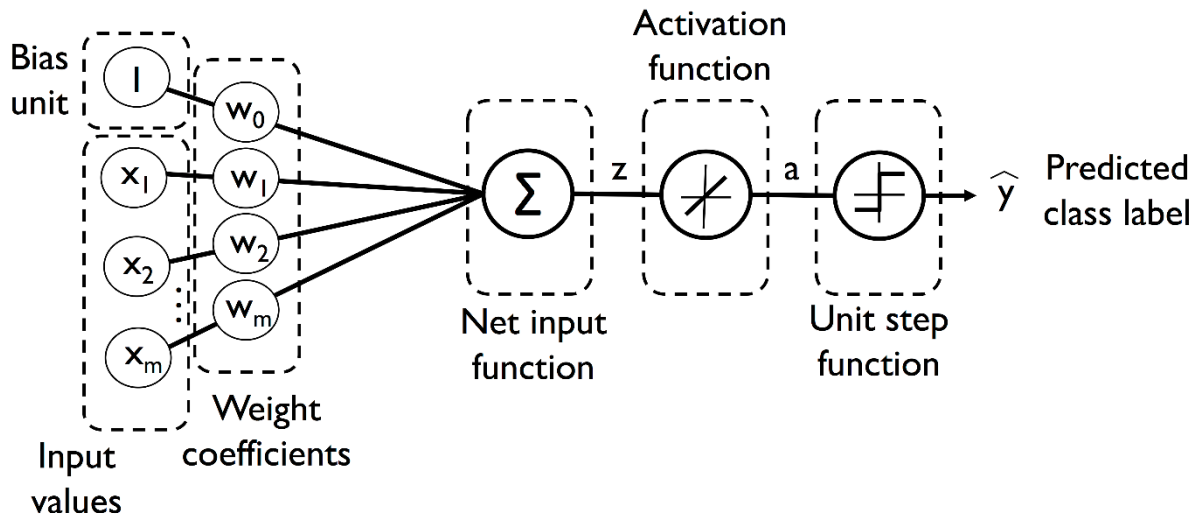


Figure 13 - Adaline for classification problems (Raschka & Mirjalili, 2017)

Deep Neural Networks extends the structure of the Adaline classifier, by having multiple neurons in multiple layers. Algorithms such as gradient decent or stochastic gradient descent are often used in the modeling process for adjusting the weights every time the algorithm repeats over the whole training set (epoch) for gradient descent or batch-wise for stochastic gradient descent. The weights are updated on each epoch with formula 3.1

$$w \coloneqq w + \Delta w, where\ \Delta w = -\eta \nabla J(w) \tag{3.1}$$

which in short means to optimize the cost function j(w) and the weights gets updated inversely to the gradient $\nabla J(w)$. The gradient also gets multiplied by the learning rate η assure proper convergence of the learning process. The activation function in the Adaline classifier is a linear combination of the net input and through the unit step function connected to the output layer.

For DNN the structure looks similar, but with a few essential changes. Figure 14 shows the structure of a deep neural network

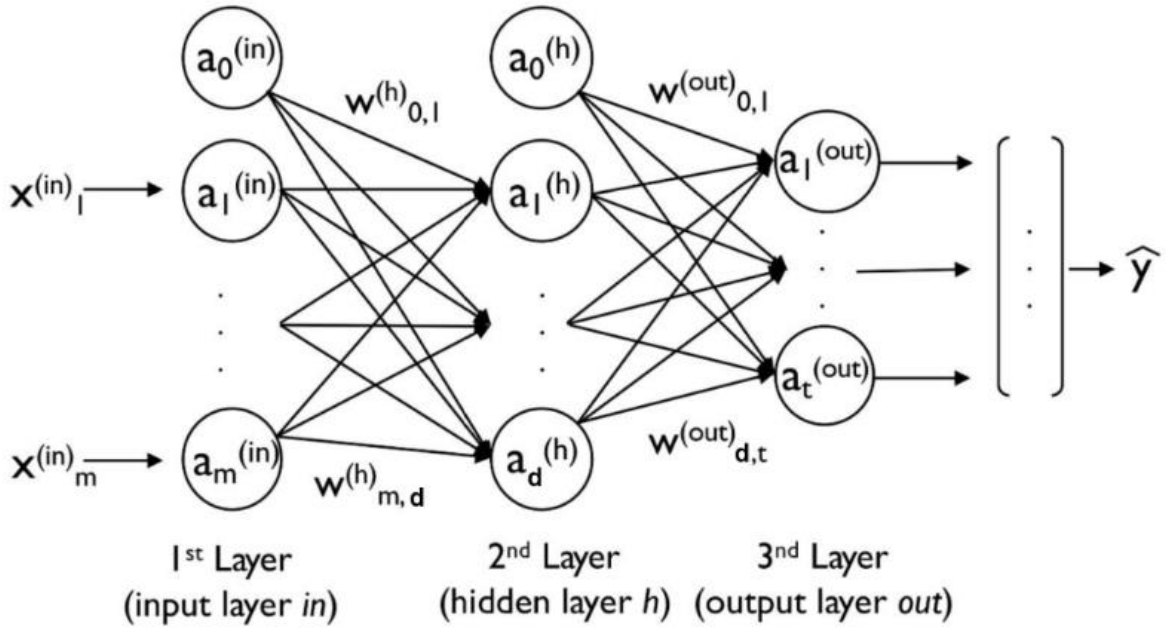


Figure 14 - One layer fully connected neural network (Raschka & Mirjalili, 2017)

including an input layer with a bias unit a_0^{in} and the inputs $x_1, \ldots, x_m$, which is fully connected to all the neurons in the second (hidden) layer except for the bias $a_0^{(h)}$. This again connects to the output layer which in a regression case produces a continuous output value. The neurons inside the network, for example a_I^h can be viewed as the net input and activation of a single Adaline neuron. A fully connected neural network with an input layer with three input units plus bias, a hidden layer with four hidden units plus bias and three units in the output layer for a three-class classification problem yields 31 weights in total. 16 weights comes from the hidden layer with four weights for the bias (1x4) and 12 weights for the input-to-hidden layer (3x4), while three weights comes from the hidden layer bias(1x3) and 12 weights for the hidden-to-output layer(3x4), which in total yields 16+15=31 weights to train. This illustrates that a very small network with only one hidden layer, three inputs, three outputs and four hidden units in the hidden layer yields a high number of weights. Regularization by controlling the number of weights, the number of neurons in a layer and the number of layers is a way to prevent against overfitting in neural network modeling.

While the identity function is used as the activation in Adaline, other functions such as the sigmoid, the hyperbolic tangent and the rectified linear unit (ReLU) are among the well-known choices in neural network modeling. The sigmoid/hyperbolic activation functions may

cause problems with vanishing gradients in the backpropagation training (Hochreiter et al., 2001). The reason for this is that the backpropagation method computes the gradients with the chain rule that may lead to weight updates of very small values when the errors become large, and if numbers between -1 and 1 are multiplied with each other enough times, the product will approach zero. As a result of this the training will become very slow and might not converge. The nonlinear ReLU activation is defined as

$$\phi(z) = \max(0, z) \tag{3.2}$$

and does not suffer from this weakness because its derivative is always 0 or 1, (0 when z is below 0 and 1 when z is above zero). In short, the ReLU activation function eliminates the vanishing gradient problem for vanilla neural networks, and is therefore often used for deep neural networks using the backpropagation algorithm. However, vanishing gradients can be an even bigger problem for RNN and will be covered in chapter 3.5.3.

(Raschka & Mirjalili, 2017)

3.5 Recurrent Neural Networks

3.5.1 Brief overview of recurrent neural networks

In chapter 3.4 we gave a description of feed-forward neural networks (NN), where the input signals flows through various weights and layers to produce an output. Such networks have no concept of time because the inputs are processed independently. Therefore, standard feed-forward neural networks are not very suitable for modeling of timeseries problems. Recurrent neural networks on the other hand include the possibility of feedback mechanisms to process information the same way as biological intelligence, where the information is processed incrementally while a memory of what is being processed is being kept so it can consider past information while processing new information (Chollet, 2017). This is however a simplification. RNN iterates through information while keeping an internal state where information from the previous timestep is being stored and used as an input in the next step. A simple illustration of the difference between a feed-forward neural network and a recurrent neural network can be seen in figure 15 where h and $h^{(t)}$ is a simplification of the hidden model architecture.

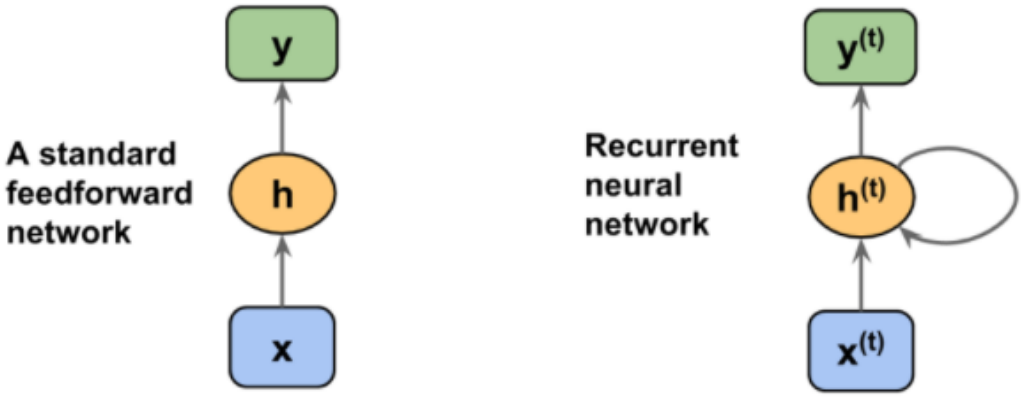


Figure 15 - Simplified structure of a feed-forward neural network and a recurrent neural network (Raschka & Mirjalili, 2017)

A simple recurrent neural network starts with an internal state at time t which often is initialized to zero. After the initialization, the network iterates in a simple for-loop over the inputs at time t, where the output at time t is a function of the input at time t and the internal state at time t, passed on from the previous timestep t-1. The internal state is then updated based on the output at time t, then the network iterates over the next timestep. A very simple implementation of this was done by Collet (2018) which explains this in pseudocode in 3.3.

```
state_t = 0
for input_t in input_sequence:
  output_t = f(input_t, state_t)
  state_t = output_t
```

(3.3)

This pseudocode can be further improved by writing the function f as the activation function of the dot-product between a weight matrix W and the input at time t plus the dot-product of a weight matrix U and the state at time t plus a bias vector. This is shown in pseudocode in 3.4

```
activation(dot(W, input_t) + dot(U, state_t) + bias)
```

(3.4)

The pseudocode in 3.3 illustrates the main difference between a NN and a RNN where the hidden layers in a NN only receives the net preactivation (input signal before the activation function is applied) from the input layer while the units in a hidden layer in a RNN receives the net preactivation from the input layer and the activation from the same hidden layer at the previous timestep, called state_t in formula 3.3. In simple terms the computational mechanisms in RNN includes feedback information from previous timesteps combined with the input from the current timestep.

Like with NN, RNN layers can be stacked (called multilayer RNN) to improve the capacity of the network. This is easily implemented in Keras (Chollet & Others, 2015) by specifying that all the recurrent layers besides the last recurrent layer should use “return_sequence = True” so the layers return a 3d tensor with the complete sequence of consecutive outputs with (batch_size, timesteps, output_features) instead of a 2d tensor with the output from the last timestep with (batch_size, output_features). However, by stacking multiple layers the model may easily overfit. After making sure a network has sufficient capacity for the problem at hand, regularization is usually included in the training process. Recurrent dropout is a common technique to prevent against overfitting in a fashion similar to the use of dropout when training feed-forward neural networks. This is done by randomly setting some inputs in a layer to zero which is supposed to reduce the coincidental connections from the training set that is fed into that layer. The implementation of recurrent dropout is not as straight forward as dropout in vanilla neural networks. Gal (2016) worked out in his PHD thesis that the same pattern (also called mask) of dropout should be applied to every timestep in the network. For other types of layers such a LSTM and GRU, discussed in chapter 3.5.4 and 3.5.5, Gal proposed that *a temporally constant dropout mask allows the network to properly propagate its learning error*

through time when it's applied to the inner recurrent activation of a layer. Gal has contributed to the Keras package by implementing two dropout arguments in the different RNN layers. One argument is the *dropout* argument that specifies the dropout rate for the input units in the layer. The second argument is the *recurrent dropout* that specifies the dropout rate of the recurrent units in the network.

(Chollet, 2017)

3.5.2 Computation of activations in Recurrent Neural Networks

The activations and computations in a recurrent neural network are more complex than in a feed-forward neural network because of the recurrent edge (also called internal state). The following section will go through the computation of activation in the hidden units and the calculation of output units.

The notation in this section is based on Raschka and Mirjalili (2017), where the weight matrix connecting the input layer $x^{(t)}$ and the hidden layer h is denoted by W_{xh} where the x represents the input and h represents the hidden layer. The weight matrix connected to the recurrent edge is denoted as W_{hh} and the weight matrix connecting the hidden layer h and the output layer y is denoted as W_{hy}. In this instance h represents a single hidden layer, but the same notation can be applied to multilayer RNN using $h_1 - \cdots - h_n$. The notation explained is shown in figure 16.

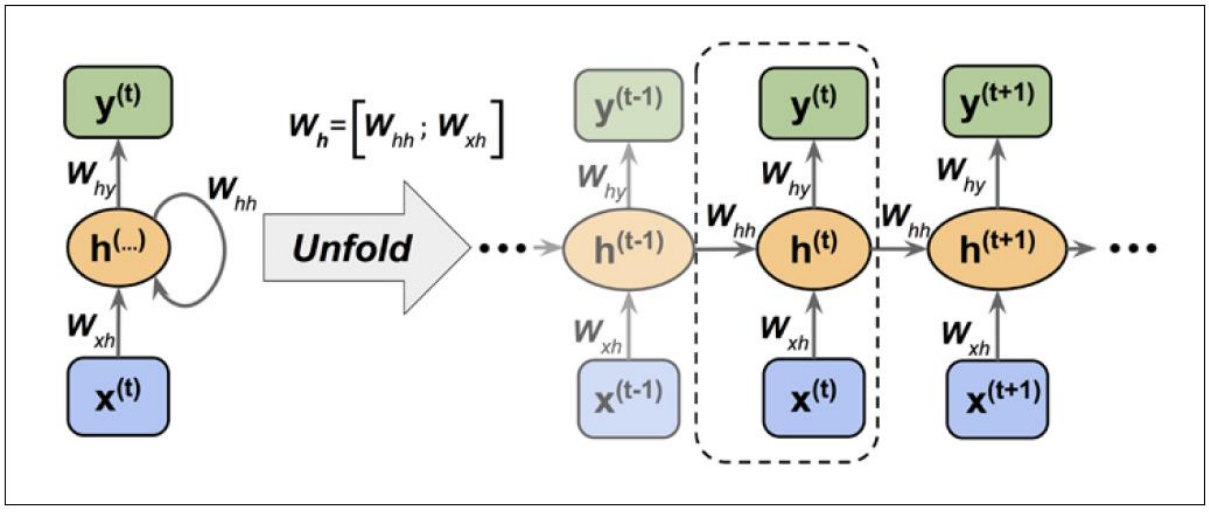


Figure 16 - Unfolded structure of a recurrent neural network (Raschka & Mirjalili, 2017)

By defining the net input as $z_h^{(t)}$, the bias vector for the hidden units as b_h and the activation function of the hidden layer as $\phi(\cdot)$, one can compute the net input in the hidden layer as

$$z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h \quad (3.5)$$

where $h^{(t-1)}$ is the activation from the same hidden layer in the previous timestep. Based on the net input, the activation in the hidden layer can be calculated as follows in 3.6.

$$h^{(t)} = \phi_h\left(z_h^{(t)}\right) = \phi_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h) \quad (3.6)$$

The activation calculation of the output layer is slightly less complicated compared to the hidden layer, as it does not take input from different timesteps. This is shown in formula 3.7.

$$y^{(t)} = \phi_y(W_{hy}h^{(t)} + b_y) \quad (3.7)$$

(Raschka & Mirjalili, 2017)

3.5.3 Vanishing gradient problem

In feed-forward neural networks the backpropagation algorithm is used to adjust the weights in the network. This is done by through so-called backpropagation, which can be understood as going backwards through the network from one sample at the time from output to input when updating the weights. The gradient, or derivative, of the loss function with respect to the weight are calculated using the chain rule. This process is done for one layer in the network at the time starting at the final layer in the model. When working with recurrent neural networks, the same type of approach is used but is then called *backpropagation through time*. The key take-aways from the backpropagation algorithm is when the errors are computed in the hidden layers, a multiplication with the derivative of the activation function is done. For activation functions like the sigmoid and tanh, this becomes a problem because of their derivatives shown in figure 17 and 18.

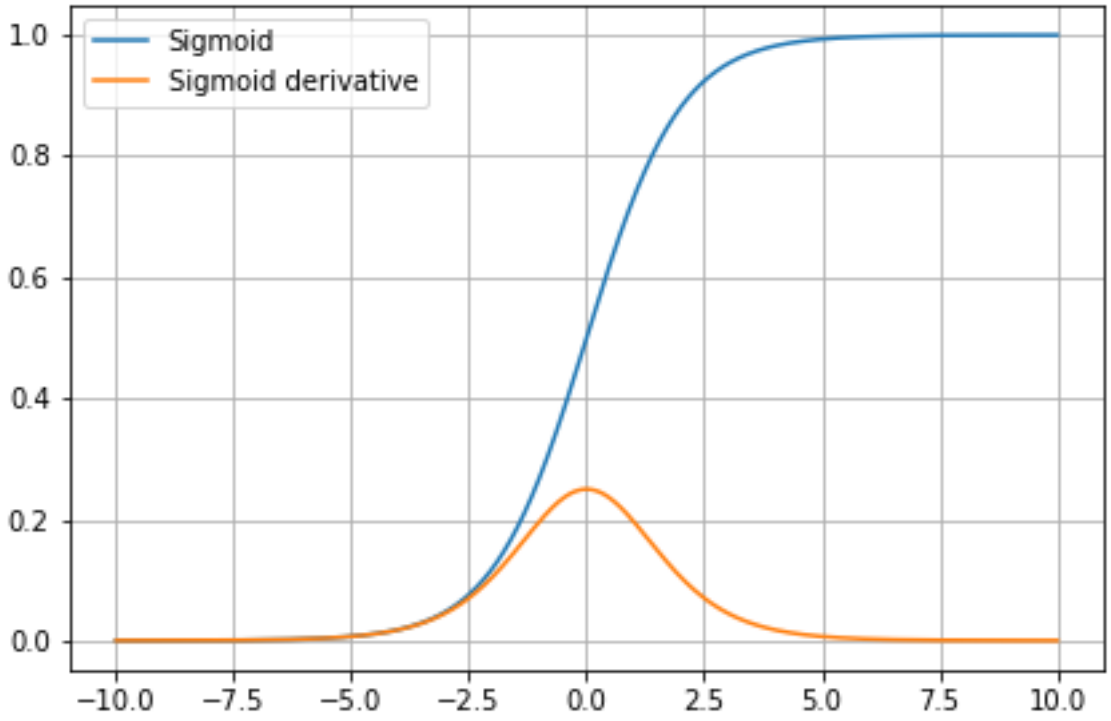


Figure 17 - Plot of the sigmoid function and its derivative in the range -10 to 10

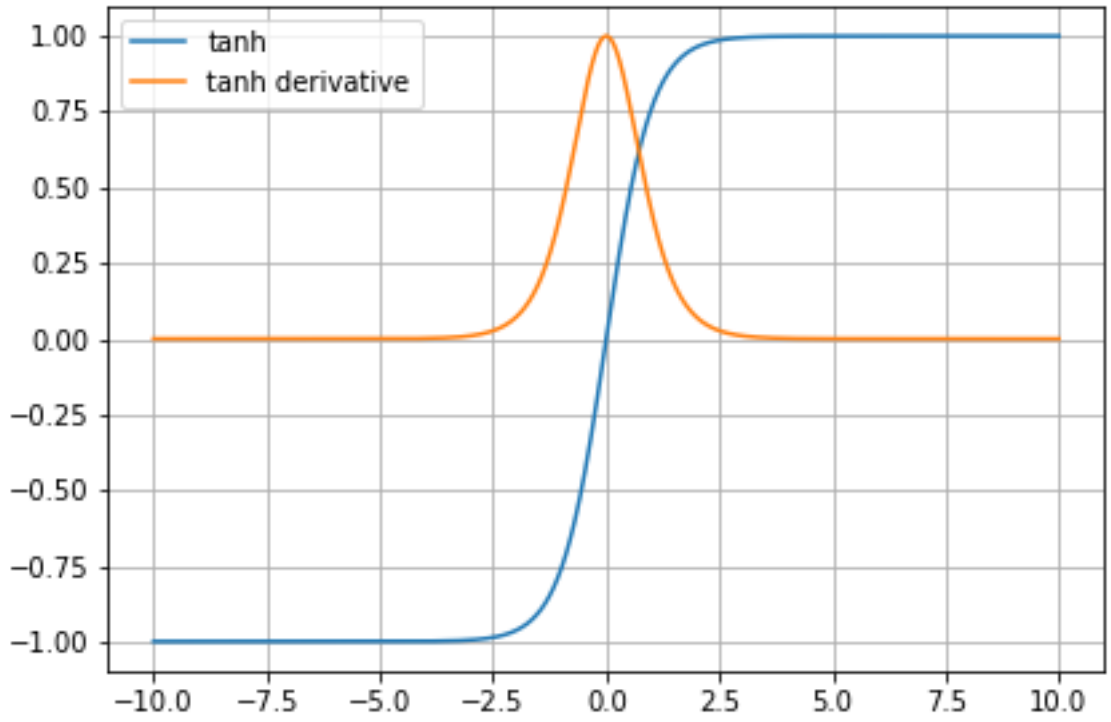


Figure 18 - Plot of the hyperbolic tangent function and its derivative in the range -10 to 10

From figures 17 and 18 it is clear that the derivative of these two activation functions lies between 0 and 1 and is close to 1 in a large region. This is the basis for the vanishing gradient

problem which occurs in multilayer neural networks because when the backpropagation algorithm updates its weights, numbers between 0 and 1 will be multiplied repeatedly. An illustration is shown in figure 19 where a net input has been given to the sigmoid function to produce an activation. The derivative of the sigmoid function can be written as $\sigma(1-\sigma)$ where σ is the sigmoid function.

Sigmoid activation	1 - sigmoid activation	Derivative sigmoid
0,1	0,9	0,09
0,2	0,8	0,16
0,3	0,7	0,21
0,4	0,6	0,24
0,5	0,5	0,25
0,6	0,4	0,24
0,7	0,3	0,21
0,8	0,2	0,16
0,9	0,1	0,09

Figure 19 - Output from the sigmoid activation function and its derivative

As illustrated in figure 19 the gradient when using the sigmoid activation function will always be a number between 0 and 0.25. This is the process for a single sample in a single layer, so when multiple gradients eventually are multiplied together, the product will approach 0. This is known as the vanishing gradient problem. The reason why this is especially relevant for recurrent neural networks is that the backpropagation algorithm leads to exponentially more calculations because the gradients are calculated for both the layer aspect and the time aspect of the network. The same principle applies for so called *exploding gradients* as well, where the same logic in figure 19 is applied but with multiplications of numbers larger than 1 leading to very high values. There are several strategies to solve this problem, and in the next subchapter we will consider two derivatives of RNN, namely Long-short term memory (LSTM) units and Gated recurrent units (GRU) that was developed to solve the problem of vanishing and exploding gradients.

3.5.4 Long-short term memory (LSTM)

Long-short term memory units was suggested by Hochreiter and Schmidhuber (1997) to solve the issue with vanishing- and exploding gradients. LSTM units are structured similarly to a simple RNN layer but has some additional advanced features. A simple RNN transfers information from one timestep to another using the activation from the previous timestep, which can lead to information getting lost when operating over many timesteps. LSTM units attempts to solve this by transferring information through the timesteps in a more robust way, often referred to as *carry*, denoted by C. By structuring the LSTM cell in this way, it prevents older signals from diminishing steadily over time as it injects information from the previous timestep. Figure 20 illustrates the unfolded structure of a LSTM cell

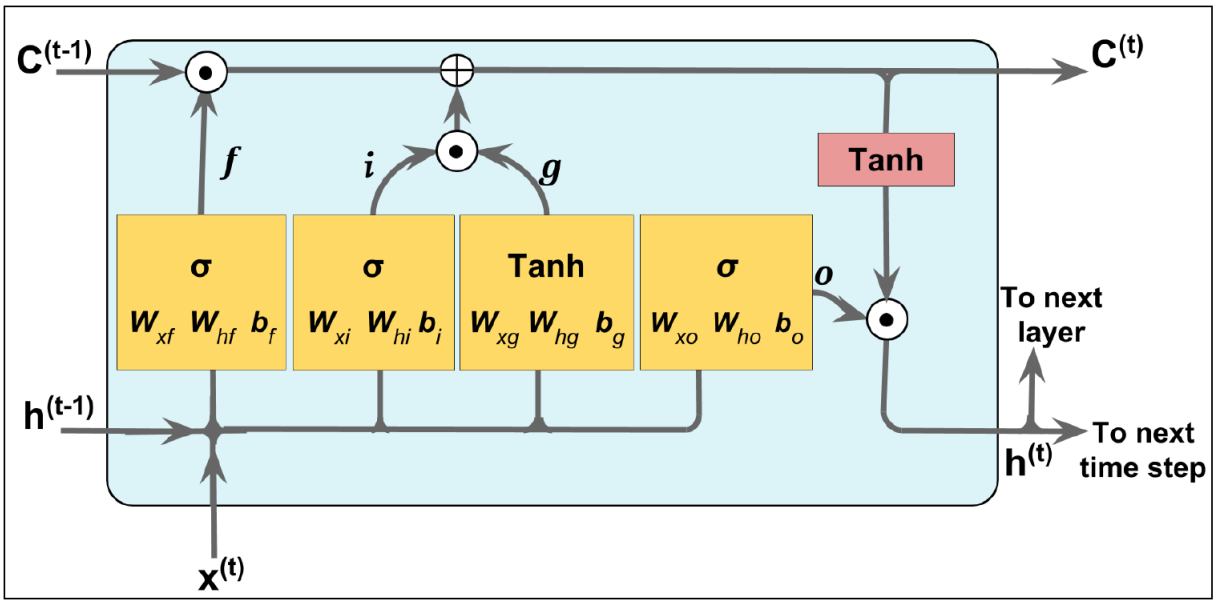


Figure 20 - In dept structure of a LSTM cell (Raschka & Mirjalili, 2017)

where a cell takes the input data from the current timestep $x^{(t)}$, $h^{(t-1)}$ which is the hidden unit activation at the previous timestep and $C^{(t-1)}$ which is the cell state at the previous timestep as inputs. As output it gives the activation $h^{(t)}$ to both the next timestep and the next layer if the networks is a multilayer network as well as the cell state $C^{(t)}$ to the next timestep. The carry (the overall process that includes the cell state) is often thought of as a conveyor belt that runs parallel through a layer of LSTM cells, where the information is not changed much besides some linear operations like element-wise multiplication denoted $\odot$ and the element-wise addition denoted $\oplus$. The cell has the capability to remove or add information to the cell state with f, i *and* g in figure 20 called *gates*. The *forget gate*, f, the *input gate*, i and the *output gate*, o uses the sigmoid activation function and the *input node*, g uses the hyperbolic tangent activation function. The yellow boxes can be thought of as a separate

neural network responsible for controlling different pieces of information in the cell, including weight matrices, W and bias units, b.

With all the parts of the LSTM cell specified, one can walk through the cell to see how the calculations processing the information are working. The first interaction in the cell is where the forget gate decides what information to pass on or suppress, essentially meaning how much of the previous cell value $C^{(t-1)}$ to use. This is done via element-wise multiplication with the output of the sigmoid activation, which is shown in figure 17 (chapter 3.5.3) to take values between 0 and 1 leading to a reduced value of the cell state. A value of 0 means keeping nothing and 1 means keeping everything. This is done so the cell state does not grow indefinitely by the element-wise addition used for the input gate and input node. When computing this activation, the same structure as formula 3.6 with the sigmoid activation function where the subscript x refers to the input, the subscript h refers to the hidden unit activation in the previous timestep and the subscript f refers to the forget gate.

$$f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f) \tag{3.8}$$

The next step in this process is to decide on what new information to add or remove from the cell state. This is done by the input gate and the input node. The task of the input gate is to decide on which values of the input to update while the input node creates a vector of new candidate values which could be added or removed from the cell state. Afterwards the activation of these gates is element-wise multiplied together before being added to the cell state. Hyperbolic tangent is used to produce activations between -1 and 1 so the input node is able to both add and remove information. The computation of the activation is much like in formula 3.8 and is shown in formula 3.9 and 3.10.

$$i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \tag{3.9}$$

$$g_t = tanh(W_{xg}x^{(t)} + W_{hg}h^{(t-1)} + b_g) \tag{3.10}$$

Here the weight matrices are denoted with the subscripts i for the input gate and g for the input node. Now that all the pieces to calculate the cell state are in place, the computation of the cell state is shown in formula 3.11.

$$C^t = (C^{(t-1)} \odot f_t) \oplus (i_t \odot g_t) \tag{3.11}$$

As described above, the forget gate regulates how much information from the previous cell state that is being kept, while the input gate and input node adds or removes information from the input and previous activation $h^{(t-1)}$.

After the information from the input gate and input node are added or removed from the cell state, the output gate decides on how much of the new cell state that goes to the output. This is done via a sigmoid activation function computed based on the input and the hidden layer activation at the previous timestep as shown in formula 3.12.

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o) \tag{3.12}$$

where the subscript *o* refers to the output gate and while the other notation and subscripts are equal to the other gates. The final hidden unit activation is then calculated using the activation from the output gate combined with the hyperbolic tangent of the cell state, shown in figure 18. The computation of this activation is shown in formula 3.13.

$$h_t = o_t \odot \tanh(C^{(t)}) \tag{3.12}$$

This activation is then passed to both the next layer and to the next timestep.

(Olah, 2015)

3.5.5 Gated Recurrent unit (GRU)

Gated Recurrent unit (GRU) is a variation of the LSTM suggested by Cho et al. (2014). The concept of the GRU is that it combines the input gate and the forget gate to one gate called the *update gate*. The cell state and the hidden state activation from the previous timestep is also merged. These changes to the LSTM cell make the GRU computationally more efficient and may also work better than the LSTM in some scenarios. However, as is often stated in Data Science: "Machine learning is often more of an art than a science" is also the case for RNN. Greff et al. (2015) studied the differences between LSTM variants and found no significant difference in performance on their tasks of speech recognition, handwriting recognition, and polyphonic music modeling. Cho et al. (2014) found that GRU tends to show better performance than LSTM on smaller datasets, which could be an advantage in this paper. An overview of a Gate Recurrent Unit is shown in figure 21.

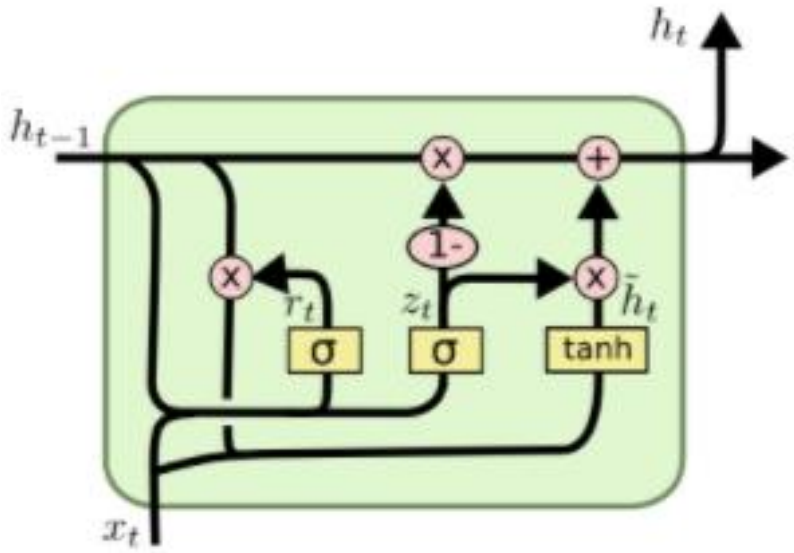


Figure 21 - In dept structure of a Gated Recurrent Unit (Olah, 2015)

The *reset gate* r_t is calculated in formula 3.13

$$r_t = \sigma(W_{xr}x^{(t)} + W_{hr}h^{(t-1)} + b_r) \tag{3.13}$$

z_t is called the *update gate* and is calculated in the same way and is shown in formula 3.14

$$z_t = \sigma(W_{xz}x^{(t)} + W_{hz}h^{(t-1)} + b_z) \tag{3.14}$$

The last gate called the *candidate activation vector* $\tilde{h}_t$ is calculated in a similar way but includes an element-wise multiplication between the reset gate and the hidden unit activation from the previous layer and utilizes the hyperbolic tangent activation function.

$$\tilde{h}_t = tanh(W_{x\tilde{h}}x^{(t)} + W_{h\tilde{h}}(h^{(t-1)} \odot r_t) + b_{\tilde{h}}) \tag{3.15}$$

The *output vector* is calculated using the gates described above either directly or indirectly and is shown in 3.16

$$h_t = (1 - z_t) \odot h^{(t-1)} + z_t \odot \tilde{h}_t) \tag{3.16}$$

(Cho et al., 2014)

3.6 Performance metrics

Evaluation of the model predictions of competing models requires some choice of performance metrics. Since this thesis works with timeseries data and a regression problem, a metric suitable to those criteria should be used. Most performance metrics used for regression contains the difference between the predicted value and the observed value, also called the forecast error. This number can be either positive or negative and will be positive if the observed value is larger than the predicted value and vice versa. However, this metric is not very useful by itself, as if you sum all the forecast errors and they contain both positive and negative errors they might cancel each other out. Therefore, most performance metric uses either squared or absolute values. Metrics like these are often used for comparison both in- and out-of-sample evaluation as well as for in-sample validation.

The *Mean Squared Error* (MSE) can be defined as in formula 3.17:

$$MSE = \frac{1}{n}SSE, where\ SSE = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{3.17}$$

where n is the number of samples, $y^{(i)}$ the observed value and $\hat{y}^{(i)}$ is the predicted value and is interpreted as the average value of the squared errors. MSE will always be non-negative and a value of 0 means a perfect accuracy for the model as the observed value equals the predicted value for all samples. Another popular metric is the *Mean Absolute Error* (MAE) and using the same notation as in formula 3.17 can be defined as

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right| \tag{3.18}$$

which represents the average of the absolute errors. As with MSE, MAE will always be non-negative and a value 0 is achieved when the observed values equal the predicted values for all samples. Based on these two metrics one can define various other metrics. One popular metric is the *Root Mean Squared Error* (RMSE) which can be defined as following

$$RMSE = \sqrt{MSE} \tag{3.19}$$

and is interpreted as the quadratic mean of the predicted and observed values. RMSE is always non-negative and will be 0 if the observed values equal the predicted values for all samples. Another metric that is popular due to its interpretability is the *Mean Absolute Percentage Error* (MAPE) which can be defined as

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y^{(i)} - \hat{y}^{(i)}}{y^{(i)}}\right| \tag{3.20}$$

The main advantage of MAPE is that it can easily be interpreted as the average absolute percentage error. In practice MAPE can be hard to use and will fail if the observed value is zero due to zero division. This could be the case in this thesis as the prices are normalized and some values may end up being zero or very close to zero. Therefore, MAPE will not be used in this thesis.

RMSE has the advantage that it is more interpretable than the MSE but might still cause confusion as it is defined as the square root of the average squared errors. A disadvantage of RMSE is that each error does not influence the sum in direct proportion which is the case with MAE (Pontius et al., 2008). According to Pontius et al. (2008) MAE is also less sensitive to outliers than RMSE due to the square and root calculations. MSE will be used to calculate the loss in the model which will be minimized during training. RMSE will be the choice of performance metric.

4. Results

In this chapter the forecasting results from modeling the aluminum, copper and zinc price is presented. For each of the metals a baseline model, a dense network model, a single-layer LSTM network model, a multi-layer LSTM network model, a single-layer GRU network model and a multi-layer GRU network model has been trained and tested using a training-, validation-, and test-set containing 70%, 20% and 10% of the original dataset, respectively. When it comes to training of the models, MSE is the metric that is being minimized during training while RMSE is being used when the predictions on the validation- and test data is presented. As previously mentioned, the input to the model is 5 timesteps (days) and the output is also 5 timesteps (days), which in practice predicts one week ahead (in trading days) based on the previous week. The models displayed in this chapter has been chosen based on their training- and validation loss. Training loss was the main criterion, but if the validation loss differed a lot from the training loss other model architectures were experimented with. Most models had relatively similar training- and validation loss, but for some of the more complex architectures we struggled to find a good compromise between training- and validation loss which is seen in the following subchapters.

4.1 Aluminum

The baseline model

The approach for the baseline model is described in chapter 3.3 and is shown in figure 22 and uses the last known observation at timestep four (indexed from zero, meaning day five in practice) to predict the next 5 timesteps.

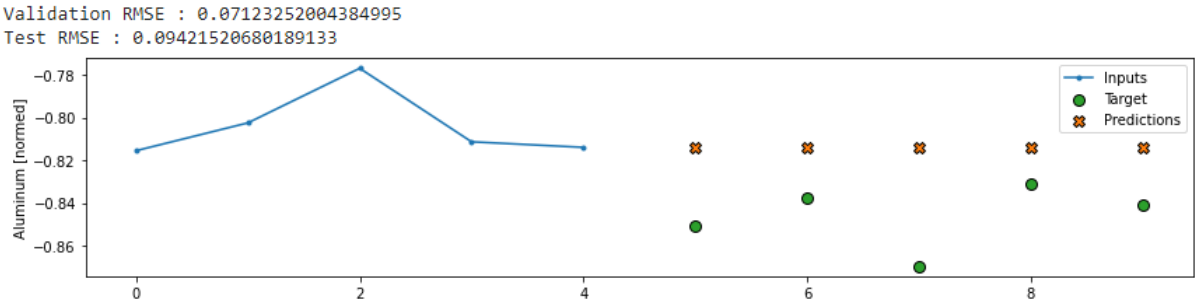


Figure 22 - Prediction of a single sequence (Baseline aluminum)

From this example we can see that our approach in general will produce useful results as the price changes from one timestep to another is not very drastic. However, the longer the time horizon and the more volatility in the commodity price will lead to worse results for this

baseline. The code for producing the baseline can be found in the Appendix. This approach yielded a RMSE of 0,0712 in the validation set and a RMSE of 0,0942 in the test set. The results from all the models with be summarized and analyzed in the discussion in chapter 5.

The dense network model

The next model is a dense model with a single layer with one unit as well as a layer with the number of output steps (5) times the number of predicted features (1) as the number of units used to reshape the output. This can be considered one of the simpler models for a problem like this. The model does in total have 71 parameters and a more detailed summary of the model structure can be found in the appendix.

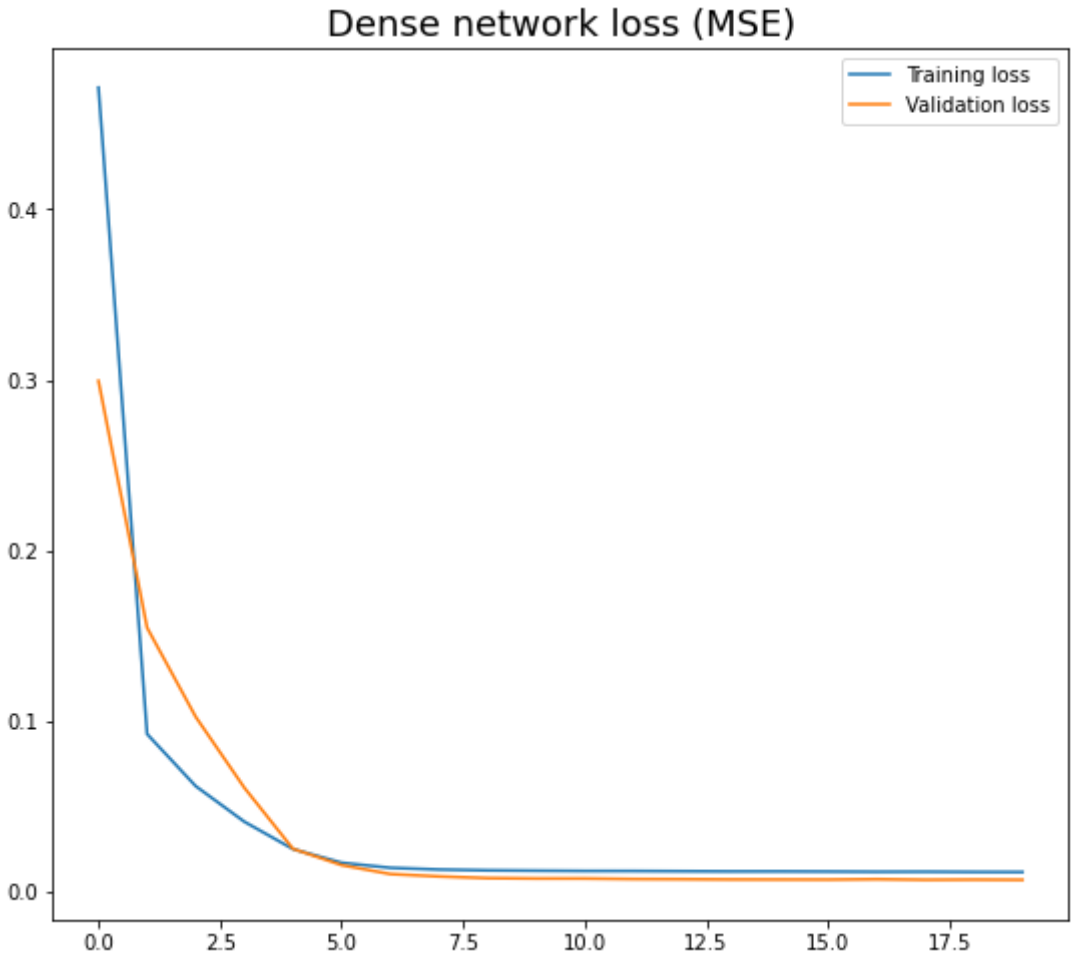


Figure 23 - Training- and validation loss (Dense network aluminum). Y-axis displays the MSE and the x-axis shows the epochs during training

The loss measured by the MSE for the training set and validation set is shown in figure 23. We can see the diminishing returns of training when the number of epochs is increased. This is especially apparent from around epoch five. The model does not seem to overfit, as the distance between the training loss and validation loss does not increase when increasing the

number of epochs. The reason why the training stopped at 18 epochs is that the *Patience* parameter in Keras' EarlyStopping (Chollet & Others, 2015) function has been set to three which means the number of epochs with no improvement after which training will be stopped. This applies for all the other trainable models as well.

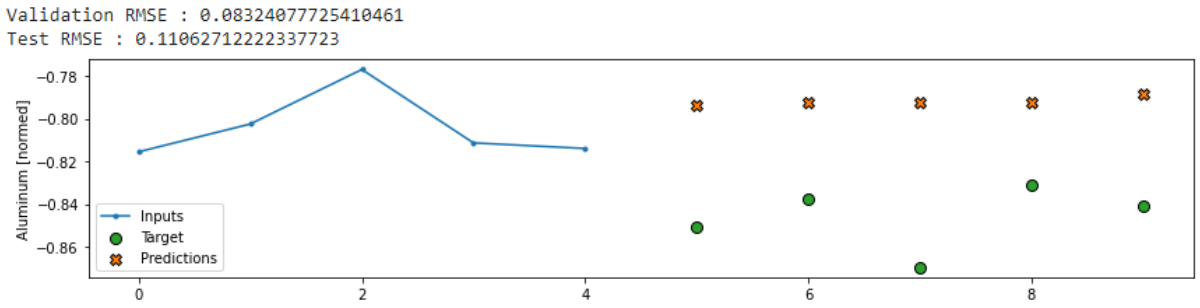


Figure 24 - Prediction of a single sequence (Dense network aluminum)

Figure 24 shows an example sequence where the dense model has predicted the output for fove timesteps. The predictions from the dense model does not change much from one timestep to another but they are not equal as with the baseline. Since we know that the prices does not change much from one timestep to another, the kernel_initializer parameter has been set to initialize to zeros using the Keras zeros initializer (Chollet & Others, 2015). To be able to predict multiple timesteps ahead the final layer of the model reshapes to match the number of output steps times the number of targets to predict. In this case we only predict the price of one metal at the time, five timesteps ahead. This approach yields a slightly higher RMSE than the baseline with a validation RMSE of 0,0832 and a test RMSE of 0,1106. The model performs slightly better on the validation data than the test data and could indicate a slight overfit, but this difference is minimal.

The single-layer LSTM model

The first of the more complex models is a single-layer LSTM model where the number of units is set to 16, yielding a total of 1 941 parameters. This is calculated by multiplying the input shape (12) with the number of units in the hidden layer (16), plus the number of units in the layer squared, plus 16 accounting for the bias. This number is then multiplied by four because of the four gates inside a LSTM cell. In total this yields $\#\ Parameters = \big((12 * 16) + (16 * 16) + 16\big) * 4 = 1856$ and is the number of parameters in the LSTM layer. In this model there is also a dense layer before reshaping the output where the number of parameters is calculated as the number of units in the previous layer (16) times the number of units in the dense layer (5) plus the number of units in the dense layer (5) which accounts

for the bias. This yields (16*5)+5=85 which gives the model a total of 1856+85 = 1 941 parameters. Calculating the number of weights by hand can be tedious and hard once the number of layers or units increases. Keras displays the number of parameters in the summary() method for the model (Chollet & Others, 2015) which is shown in the appendix. Therefore, we will not provide calculation of the number parameters for the other models. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output at the final timestep. A detailed summary of the model can be found in the appendix.

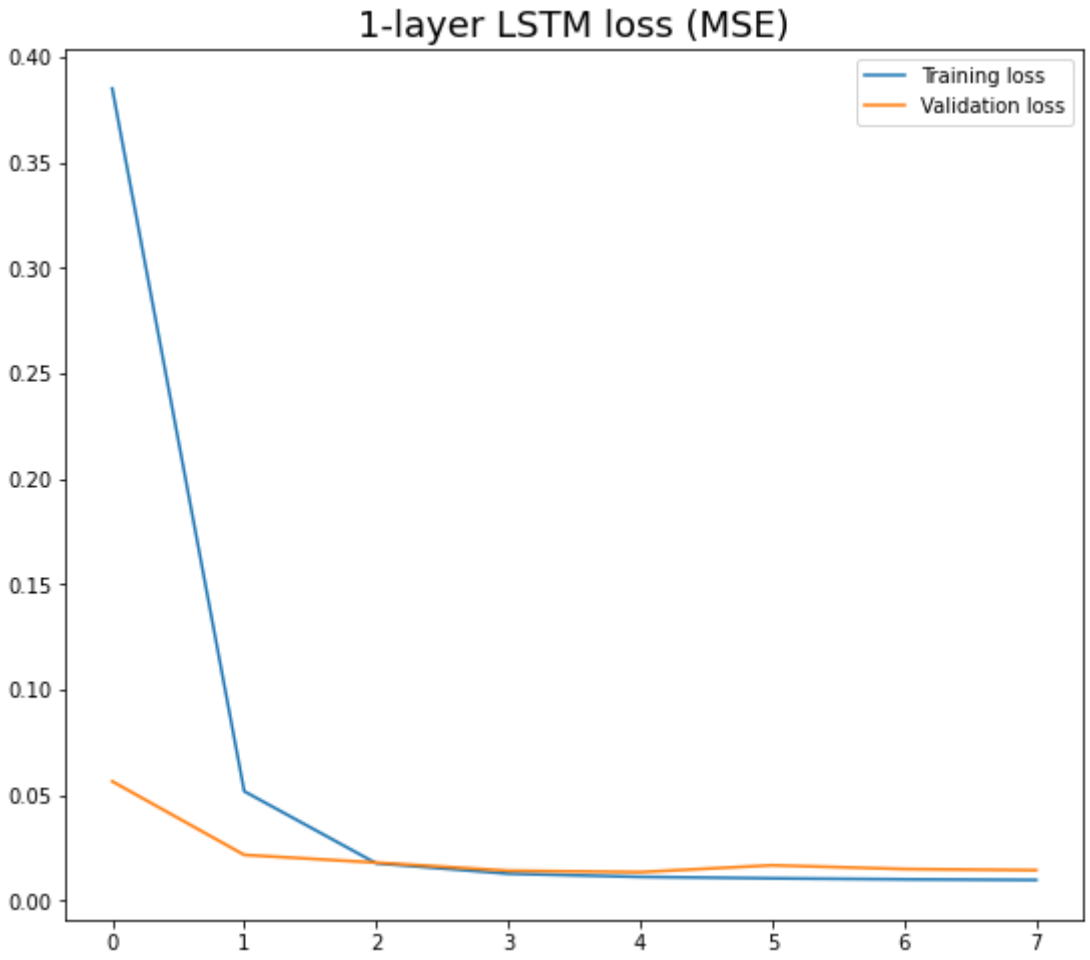


Figure 25 – Training- and validation loss (single-layer LSTM aluminum). Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer LSTM model shown in figure 25, which seems to train at a faster rate than the dense model, as the training loss is not significantly reduced after epoch 2. This makes sense as the LSTM model has a higher capacity than the dense model. The validation loss is slightly higher than the training loss but based on this figure one cannot prove that the model is overfitting.

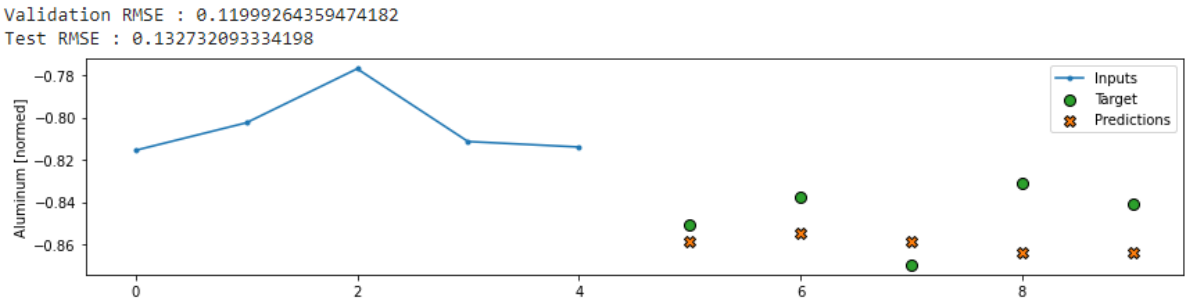


Figure 26 - Prediction of a single sequence (single-layer LSTM aluminum)

The same example sequence as for the other models is shown in figure 26 and like the two previous models, the LSTM model seem to predict values near the last known observation in this specific sample, but closer to the targets than the previous two models. As with the dense model, the weights are initialized to zero yielding small changes from the last known observation. In total over the entire validation and test set, the single-layer LSTM model yielded a RMSE of 0,12 and 0,1327, respectively. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes well.

The Multi-layer LSTM model

The multi-layer LSTM model is built up with three LSTM layers consisting of 128 units and a dense layer with 128 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first two layers and False on the third layer. This structure is required when stacking LSTM layers where all layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. In total this structure yields 352 517 parameters giving the model a lot more capacity than previous models. A detailed summary of the model can be found in the appendix.

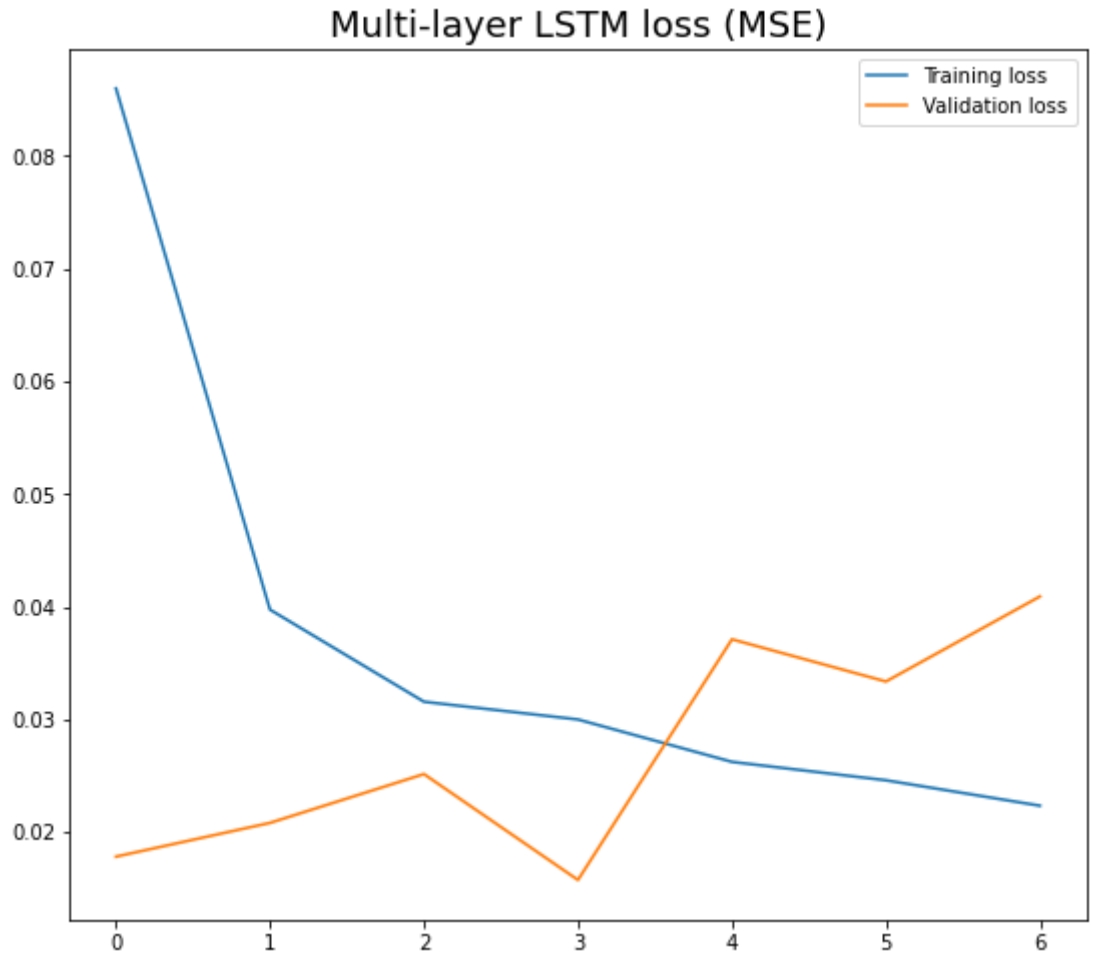


Figure 27 – Training- and validation loss (multi-layer LSTM aluminum). Y-axis displays the MSE and the x-axis shows the epochs during training

The training loss for the multi-layer LSTM model is quite different compared to the previous models. The training loss have a steady decline over all the epochs. The validation loss however swings a lot more over the epochs and increases steadily after the third epoch. The exact reason behind this behavior is hard to conclude, but one possibility could be that since the dataset is relatively small the model struggles predicting some samples in the validation set during the training of the model. Since the validation set is so small, this could be more critical for the validation part. Due to the large capacity of the network, this erratic movement of the validation loss could also be a result of overfitting. This might be indicated by the validation RMSE and test RMSE seen in figure 28 as the model performs considerably worse than the previous models.

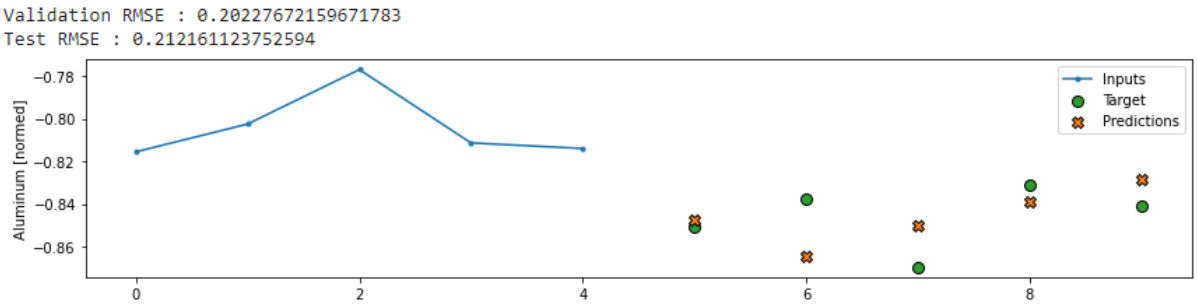


Figure 28 - Prediction of a single sequence (multi-layer LSTM aluminum)

The example shown in figure 28 we can see that the model predicts the sequence quite well, especially for timestep five, eight and nine. However, this is just one sample and in total the model performs worse than all the previous models with a validation RMSE of 0,2022 and a test RMSE of 0.2122. A small difference between the validation and test RMSE indicates a model that does not overfit but the learning curves in figure 27 may indicated that the model is too complex and struggles to make good predictions due to the poor performance on the validation and test set.

The single-layer GRU model

As an alternative to the LSTM layers, this model applies a single GRU layer where the number of units is set to 32, yielding a total of 4 581 parameters. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output of the final timestep. A detailed summary of the model can be found in the appendix.

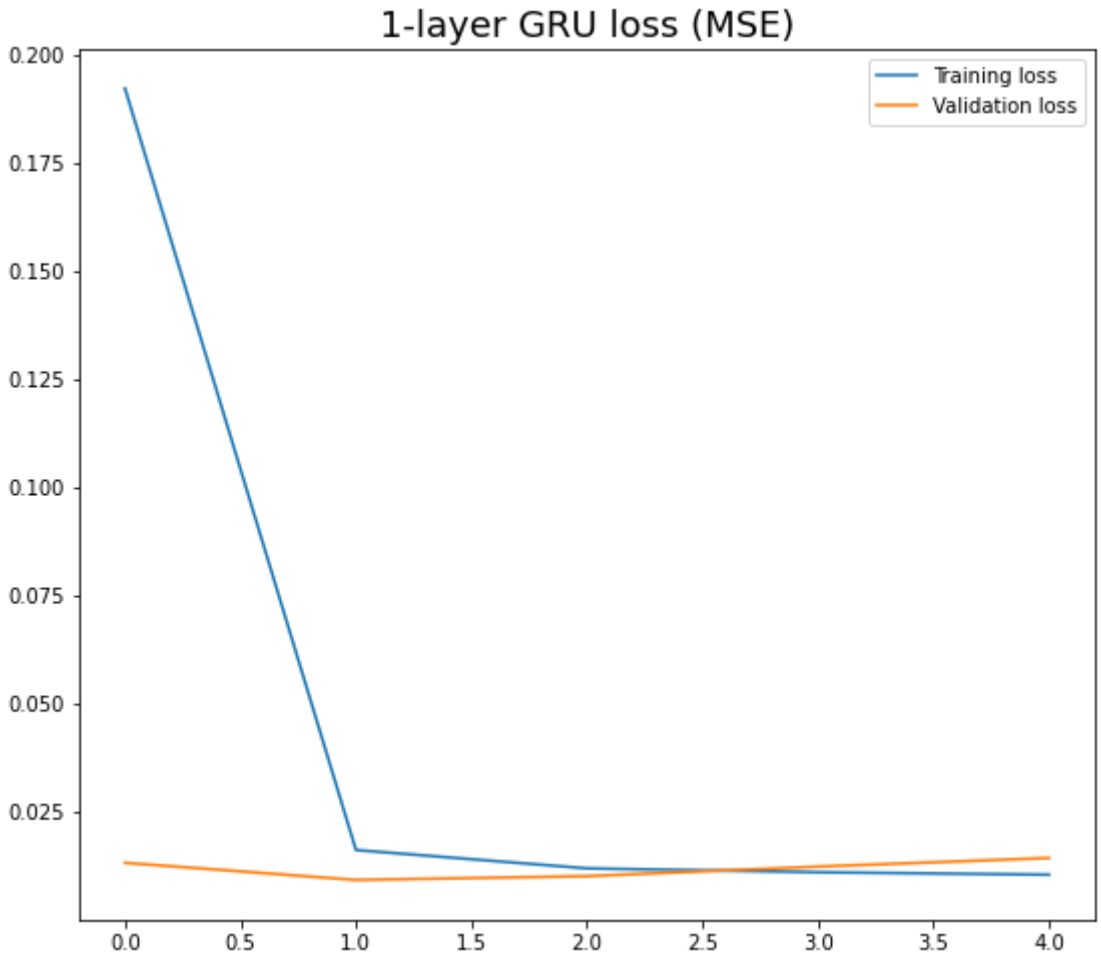


Figure 29 - Training- and validation loss (single-layer GRU aluminum) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer GRU is model shown in figure 29, which seems to train at a faster rate than the single-layer LSTM model, as the training loss is not significantly reduced after the first epoch. The validation loss is slightly lower than the training loss for a little over the first half of the epochs and slightly higher for the last epoch. These differences are very small, and the model seems to train well without overfitting.

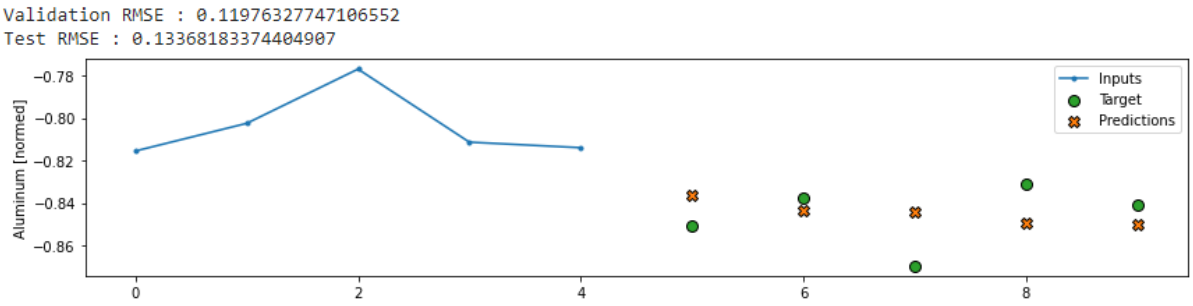


Figure 30 - Prediction of a single sequence (single-layer GRU aluminum)

From the example shown in figure 30 we can see that the model quite accurately predicts timestep five, six and nine. However, this is just one sample and in total the model performs worse than the baseline and the dense model but better than the multi-layer LSTM and about even with the single-layer LSTM model with a validation RMSE of 0,1198 and a test RMSE of 0,1337. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes well which is the case for this model.

The multi-layer GRU model

The multi-layer GRU model is built up in the same fashion as the multi-layer LSTM model with three GRU layers consisting of 128 units and a dense layer with 128 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first two layers and False on the third layer. This is structure required when stacking GRU layers where all layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. In total this structure yields 269 829 parameters giving the model a solid capacity but with fewer parameters than the multi-layer LSTM, even though the model architecture is the same. This is because of the way the GRU layer is built up compared to the LSTM layer, which is described in detail in chapter 3.5.4 and 3.5.5. A detailed summary of the model can be found in the appendix.

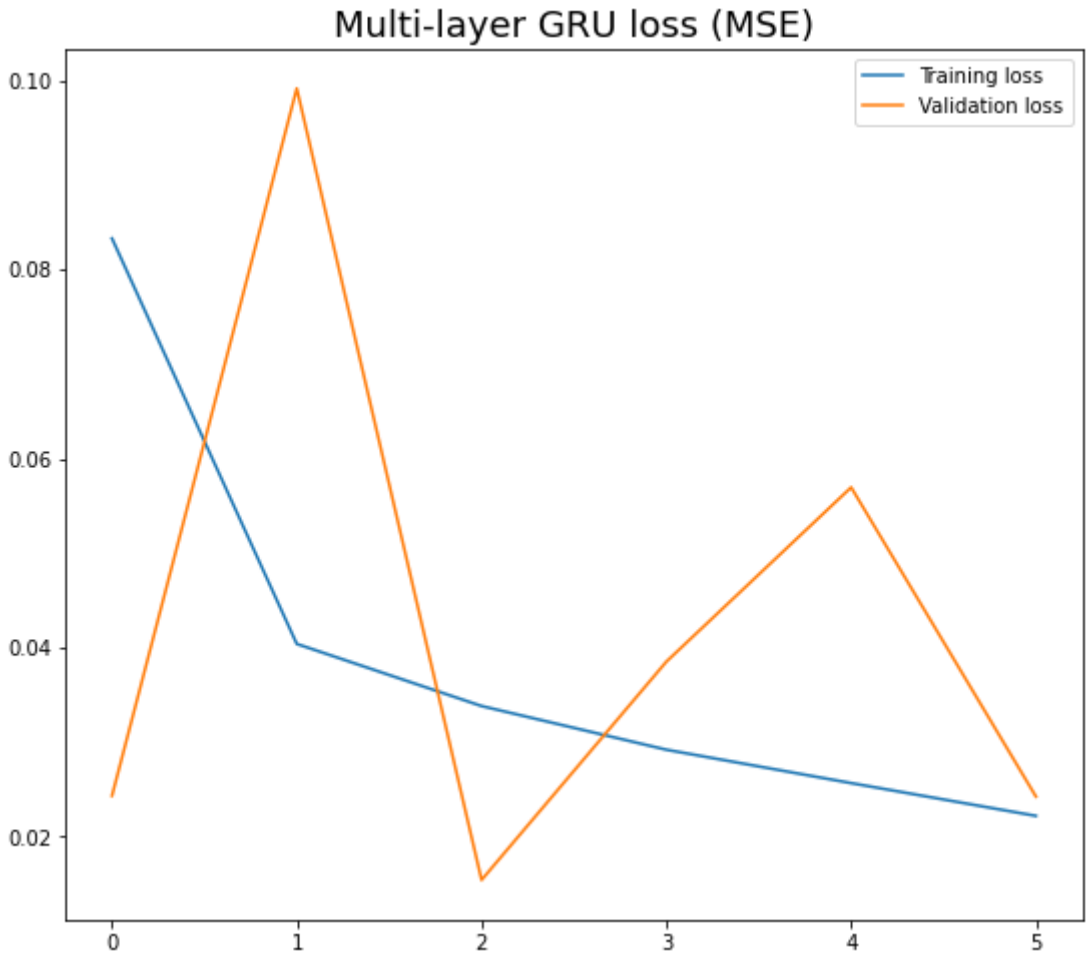


Figure 31 - Training- and validation loss (multi-layer GRU aluminum) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the multi-layer GRU model shown in figure 31. The training stops after five epochs where the training loss is reduced in the greatest amount in the first epoch with a steady decrease until epoch five. The validation loss however is more erratic and largely varying from one epoch to another, in the same fashion as the multi-layer LSTM model. This could be due to the large capacity of the model with close to 300 000 weights to be trained. Results like these could indicate overfitting but comparing the validation and test results in figure 32 does not directly support this theory.

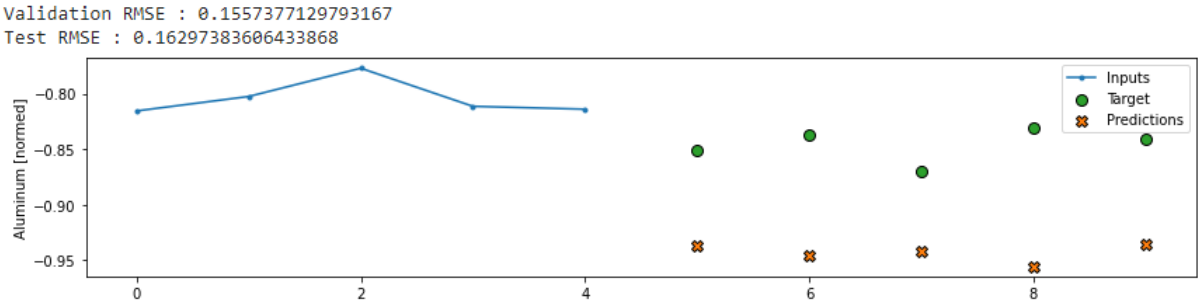


Figure 32 - Prediction of a single sequence (multi-layer GRU aluminum)

The example in figure 32 shows that the multi-layer GRU does not predict well for this particular sample. All in all, this model performs worse than all the other models besides the multi-layer LSTM with a validation RMSE of 0,1557 and a test RMSE of 0,163. The learning curves in figure 31 indicate that the model is too complex and fails to make good predictions as indicated by the poor performance on the validation and test set in the same way as the multi-layer LSTM model.

4.2 Copper

The baseline model

The approach for the baseline model is described in 3.3 and is shown in figure 33 and uses the last known observation at timestep four (indexed from zero, meaning day five in practice) to predict the next five timesteps.

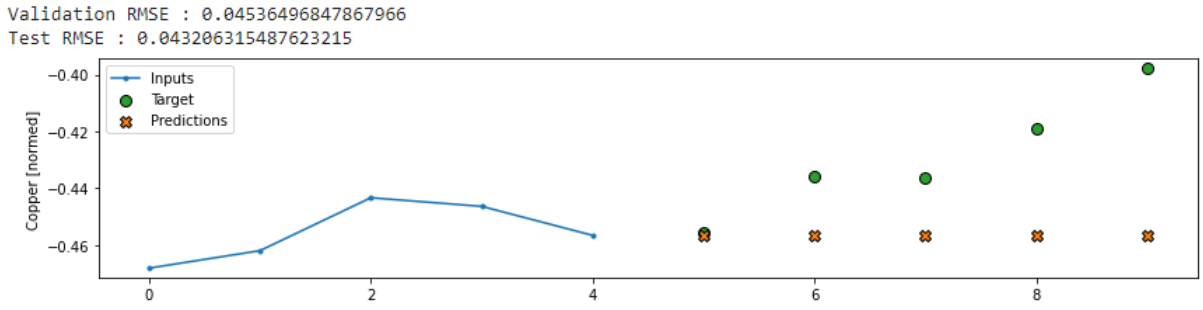


Figure 33 - Prediction of a single sequence (Baseline copper)

From this example we can again see that this approach in general will produce decent results when the price changes from one timestep to another are not very drastic. However, the longer the time horizon and the more volatility in the commodity price will lead to worse results for this baseline. The first prediction is almost spot on, but as the price increases from timestep five to nine, the baseline predicts further away from the target. The code for producing the baseline can be found in the Appendix and is the same for all metals. This approach yielded a

RMSE of 0,0454 in the validation set and a RMSE of 0,0432 in the test set. The results from all the models summarized and discussed in chapter 5.

The dense network model

The next model is a dense model with a single layer with one unit as well as a layer with the number of output steps (5) times the number of predicted features (1) as the number of units used to reshape the output. This can be considered one of the simpler models for a problem like this. The model does in total have 71 parameters and a more detailed summary of the model structure can be found in the appendix.

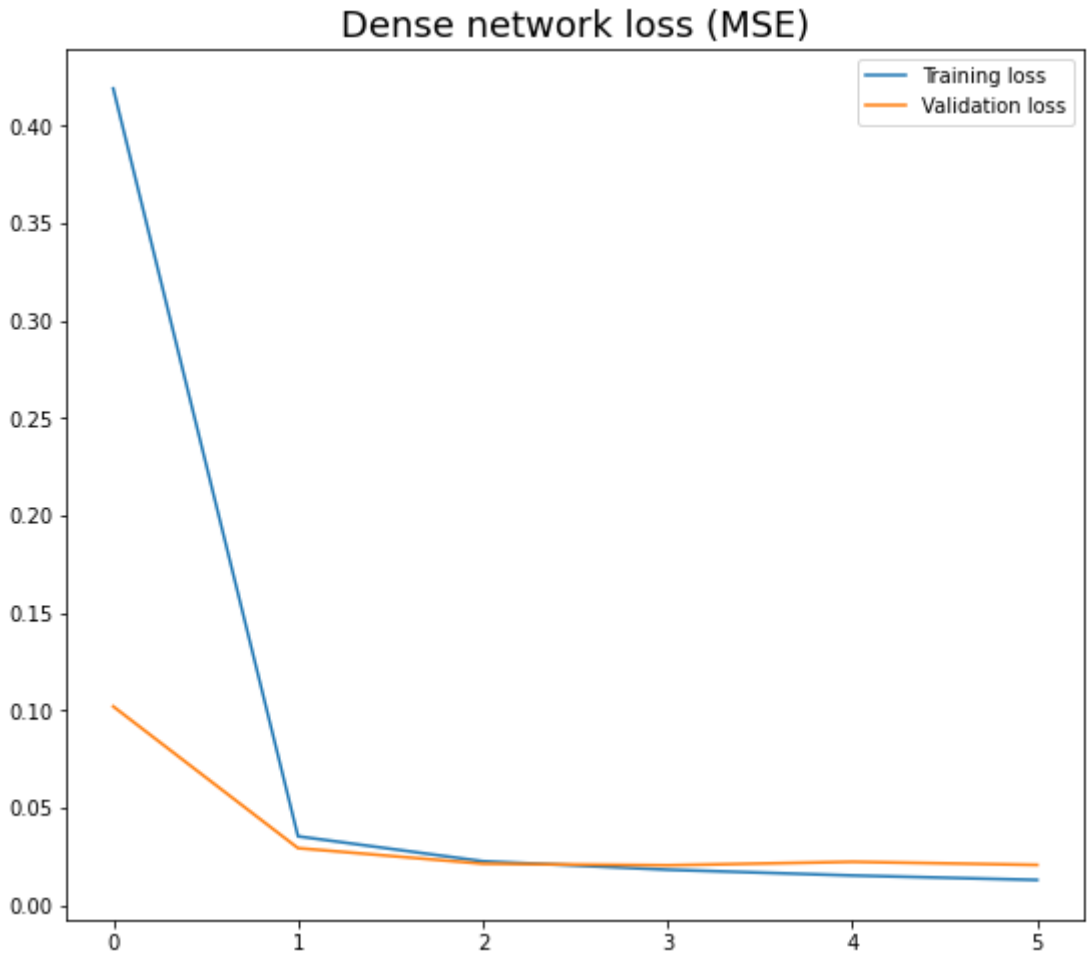


Figure 34 - Training- and validation loss (dense network copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training loss measured by the MSE for the training set and validation set is shown in figure 34. We can see the diminishing returns of training when the number of epochs is increased. This is especially apparent from around epoch 1 and 2. The model does not seem to overfit, as the distance between the training loss and validation loss does not increase when increasing the number of epochs. The reason why the training stopped at five epochs is that

the *Patience* parameter in Keras' EarlyStopping (Chollet & Others, 2015) function has been set to three which means the number of epochs with no improvement after which training will be stopped. This applies for all the other trainable models as well.

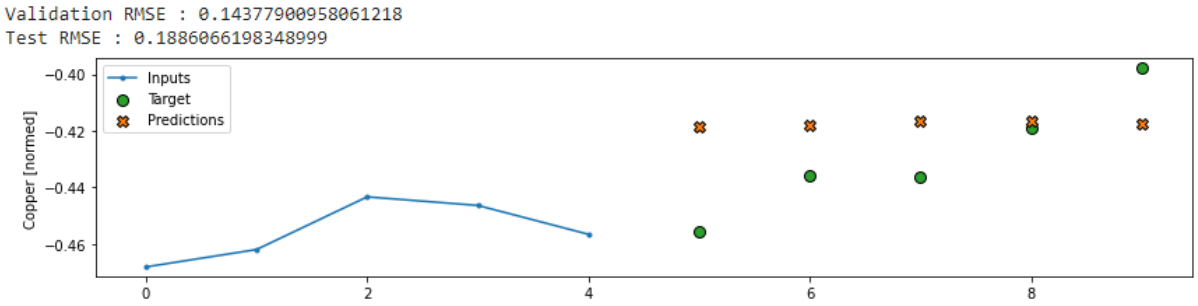


Figure 35 - Prediction of a single sequence (dense network copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

Figure 35 shows an example sequence where the dense model has predicted the output for 5 timesteps. The predictions from the dense model does not change much from one timestep to another but they are not identical as with the baseline. Since we know that the prices does not change much from one timestep to another, the kernel_initializer parameter has been set to initialize to zeros using the Keras' zeros initializer (Chollet & Others, 2015). This is the case for all the other models as well and can be seen in the appendix. To be able to predict multiple timesteps ahead the final layer of the model reshapes to match the number of output steps times the number of targets to predict. In this case we only predict the price of one metal at the time, five timesteps ahead. This approach yields a much higher RMSE than the baseline with a validation RMSE of 0,1438 and a test RMSE of 0,1886.

The single-layer LSTM model

The single-layer LSTM model for forecasting the copper price consists of a single LSTM layer with 32 units, yielding a total of 5 925 parameters. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output at the final timestep. A detailed summary of the model can be found in the appendix.

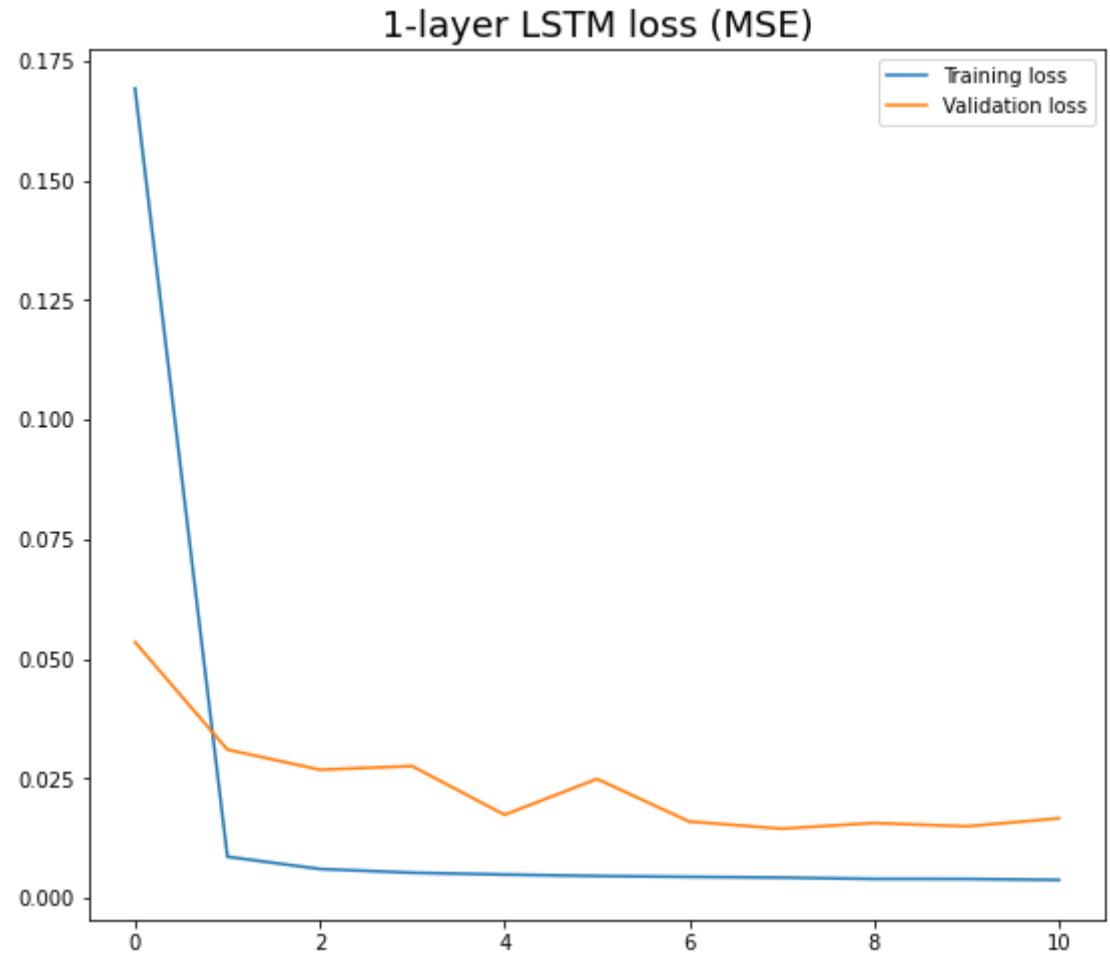


Figure 36 - Training- and validation loss (single-layer LSTM copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer LSTM model is shown in figure 36, and we can see the training loss is not significantly reduced after the first epoch. This has been the case for several models with larger capacity, while models with a lower number of parameters often needs more epochs for the loss to flatten. The validation loss is slightly higher than the training loss but based on this figure one cannot prove that the model is overfitting.

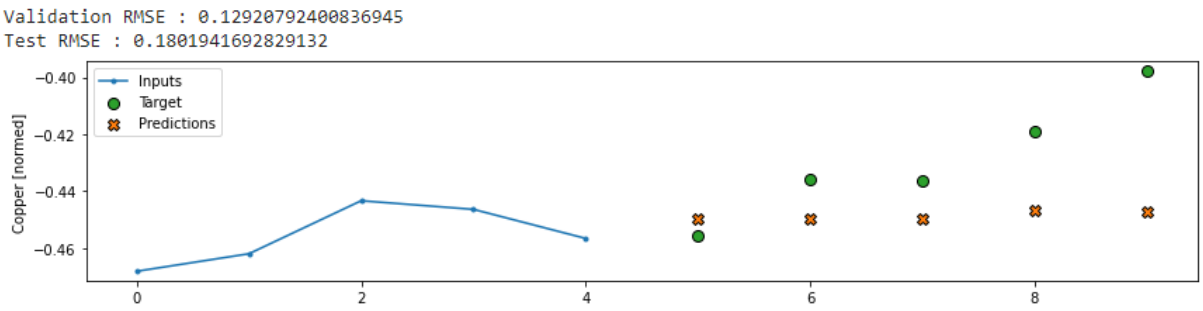


Figure 37 - Prediction of a single sequence (single-layer LSTM copper)

The same example sequence as for the other models is shown in figure 37 and like the two previous models, the LSTM model seem to predict values near the last known observation in this specific sample, but closer to the targets than the dense model. As with the dense model, the weights are initialized to zero yielding small changes from the last known observation. This is however just one sample and would differ if another sample were picked. In total over the entire validation and test set, the single-layer LSTM model yielded a RMSE of 0,1292 and 0,1802, respectively. A difference between the validation and test RMSE as large as this may indicate that the model has been slightly overfit to the training data.

The multi-layer LSTM model

The multi-layer LSTM model used for forecasting the copper price is built up with two LSTM layers consisting of 32 units and a dense layer with 32 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first layer and False on the second layer. This structure is required when stacking LSTM layers where all the layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. This structure yields a total of 15 301 parameters giving the model a lot more capacity than previous models for copper. A detailed summary of the model can be found in the appendix.

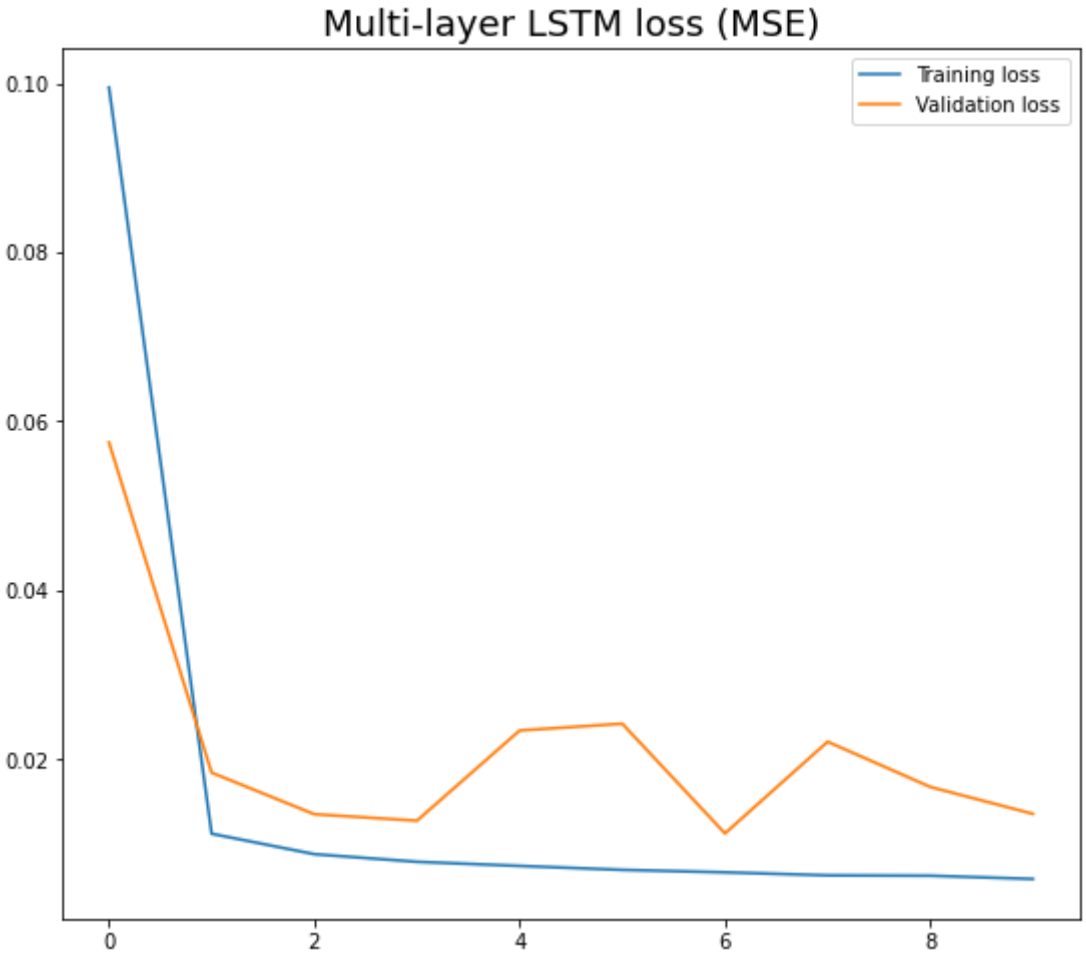


Figure 38 - Training- and validation loss (multi-layer LSTM copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training process for the multi-layer LSTM model stops after nine epochs and have most of the progress in the first epoch. The training loss have a steady decline over all the epochs starting to flatten more out after the first epoch. The validation loss is slightly more volatile without differing too much from the training loss. This model has one less layer and fewer parameters per layer than the multi-layer LSTM model developed for aluminum and seems to be a more stable model.

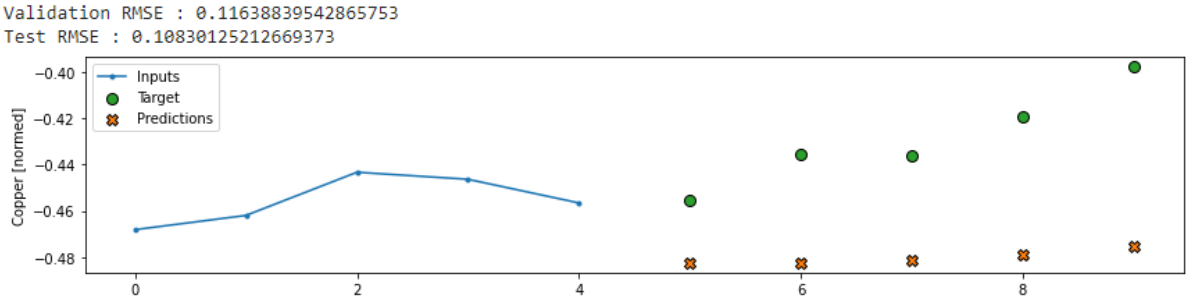


Figure 39 - Prediction of a single sequence (multi-layer LSTM copper)

From the example shown in figure 39 we can see that the model seems to predict the sequence in a similar way to the previous models and the predictions does not change much from one timestep to another. However, this is just one sample and in total the model performs worse than the baseline but better than the dense and single-layer LSTM models with a validation RMSE of 0,1163 and a test RMSE of 0,1083. This model also performs slightly better on the test data than the validation data and does not indicate any overfitting as the model seems to generalize well on new data.

The single-layer GRU model

As an alternative to the LSTM layers, this model applies a single GRU layer where the number of units is set to eight, yielding a total of 573 parameters which is the lowest number of parameters thus far for the LSTM and GRU models. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output at the final timestep. A detailed summary of the model can be found in the appendix.

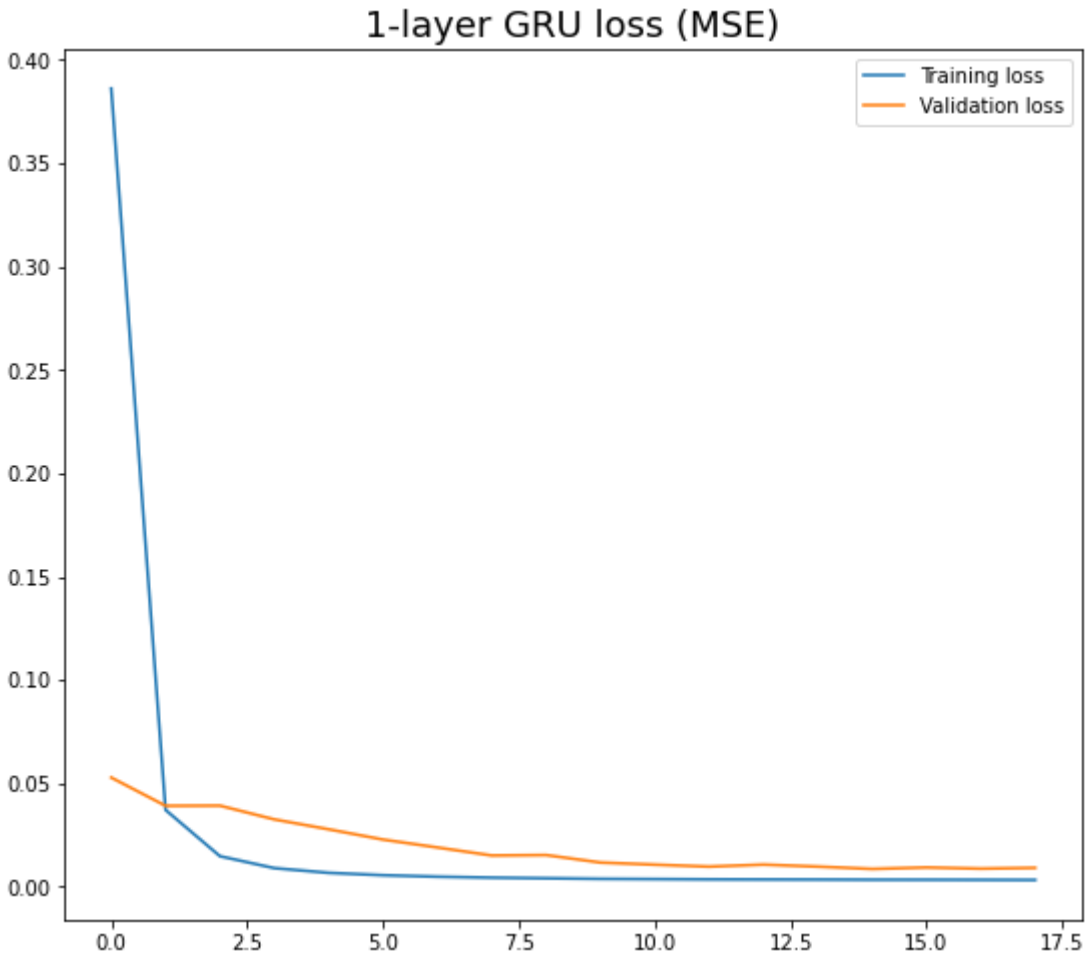


Figure 40 - Training- and validation loss (single-layer GRU copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer GRU model shown in figure 40, and we can see that the majority of the reduction in the loss happens before the third epoch. The validation loss is slightly higher than the training loss after the first epoch, but as the number of epochs increases the difference between the training and validation loss does not increase. This indicates that the model does not overfit to the training data.

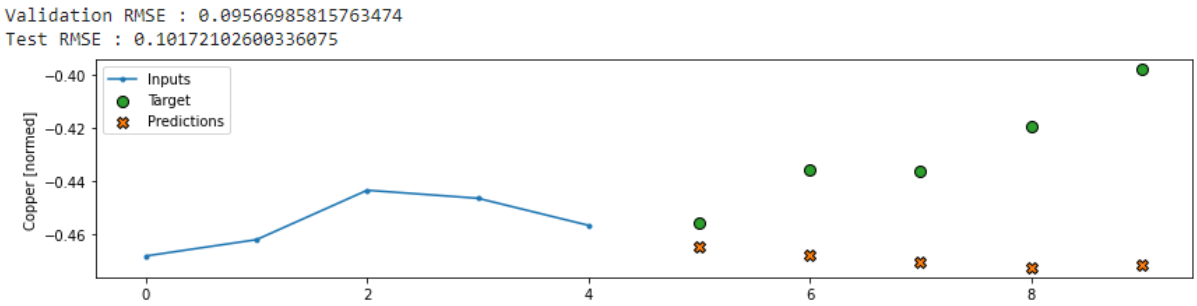


Figure 41 - Prediction of a single sequence (single-layer GRU copper)

From the example shown in figure 41 we can see that the model accurately predicts in a similar fashion to the LSTM models where the first prediction is the best one. However, this is just one sample and in total the model performs worse than the baseline but better than all the other models forecasting the copper price with a validation RMSE of 0,0956 and a test RMSE of 0,1017. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes well.

The Multi-layer GRU model

The multi-layer GRU model is built up in the same fashion as the multi-layer LSTM model for copper with two GRU layers consisting of 32 units and a dense layer with 32 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first layer and False in the second layer. This structure is required when stacking GRU layers where all layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. In total this structure yields 11 973 parameters giving the model a solid capacity but with fewer parameters than the multi-layer LSTM, even though the model architecture is the same. This is because of the way the GRU layer is built up compared to the LSTM layer, which is described in detail in chapter 3.5.4 and 3.5.5. A detailed summary of the model can be found in the appendix.

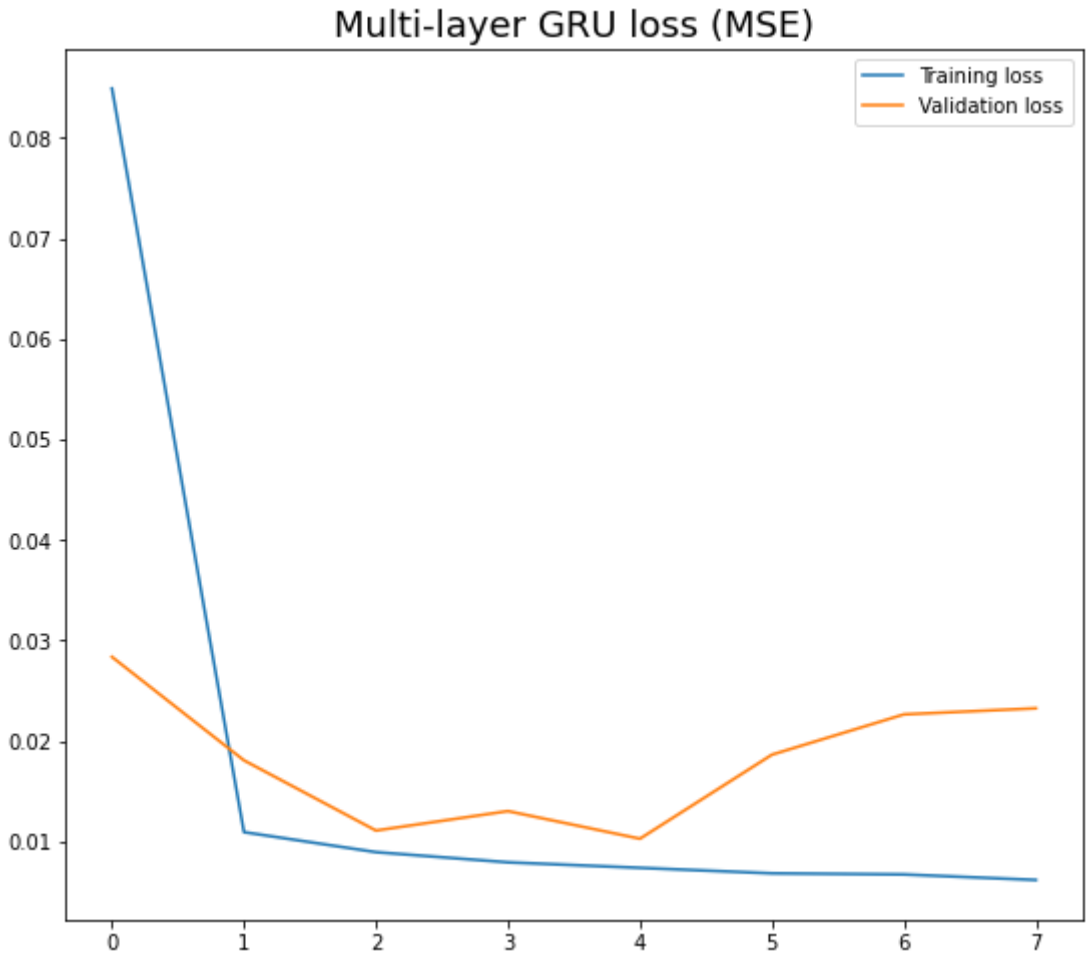


Figure 42 - Training- and validation loss (multi-layer GRU copper) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the multi-layer GRU model is shown in figure 42. Training stops after seven epochs where the training loss is reduced the most in the first epoch with a steady decrease until epoch seven. The validation loss however is slightly more erratic and increases after four epochs. This could be due to the large capacity of the model with 11 973 weights to be trained. Results like these could indicate overfitting but the validation and test RMSE are almost identical which does not indicate overfitting.

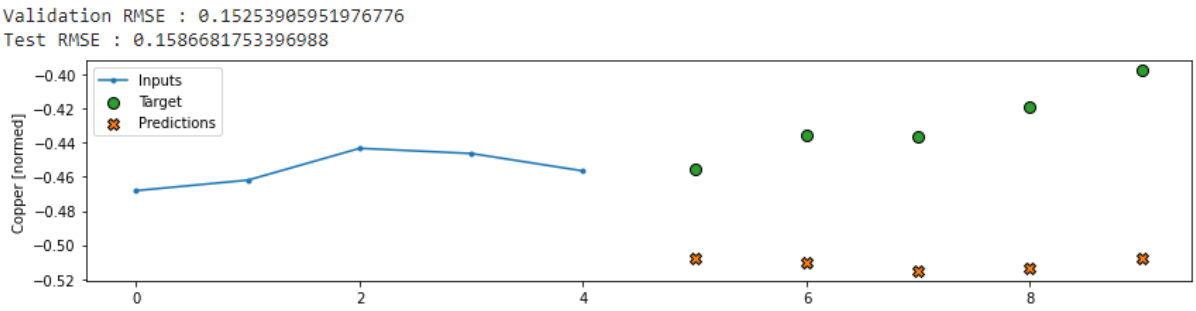


Figure 43 - Prediction of a single sequence (multi-layer GRU copper)

The example in figure 43 shows that the multi-layer GRU predicts this sequence similarly to the other models for copper. However, this is just one sample and in total the model performs worse than all the other models in validation but slightly better than the dense model and the single-layer LSTM model with a validation RMSE of 0,1525 and a test RMSE of 0,1587. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes well.

4.3 Zinc

The baseline model

The approach for the baseline model is described in 3.3 and is shown in figure 44 and uses the last known observation at timestep four (indexed from zero, meaning day five in practice) to predict the next five timesteps.

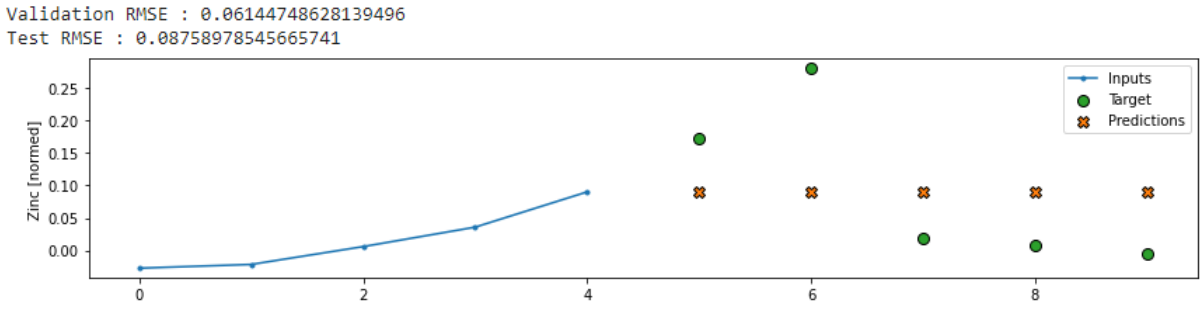


Figure 44 - Prediction of a single sequence (Baseline zinc)

From this example we can again see that this approach in general will produce decent results when the price changes from one timestep to another is not very drastic. However, the longer the time horizon and the more volatility in the commodity price will lead to worse results for this baseline. The first prediction is very close, and the baseline predicts well in general. The code for producing the baseline can be found in the Appendix and is the same for all metals. This approach yielded a RMSE of 0,06145 in the validation set and a RMSE of 0,0875 in the test set. The results from all the models are summarized and discussed in chapter 5.

The dense network model

The next model is a dense model with a single layer with one unit as well as a layer with the number of output steps (5) times the number of predicted features (1) as the number of units used to reshape the output. This can be considered one of the simpler models for a problem like this. The model does in total have 71 parameters and a more detailed summary of the model structure can be found in the appendix.

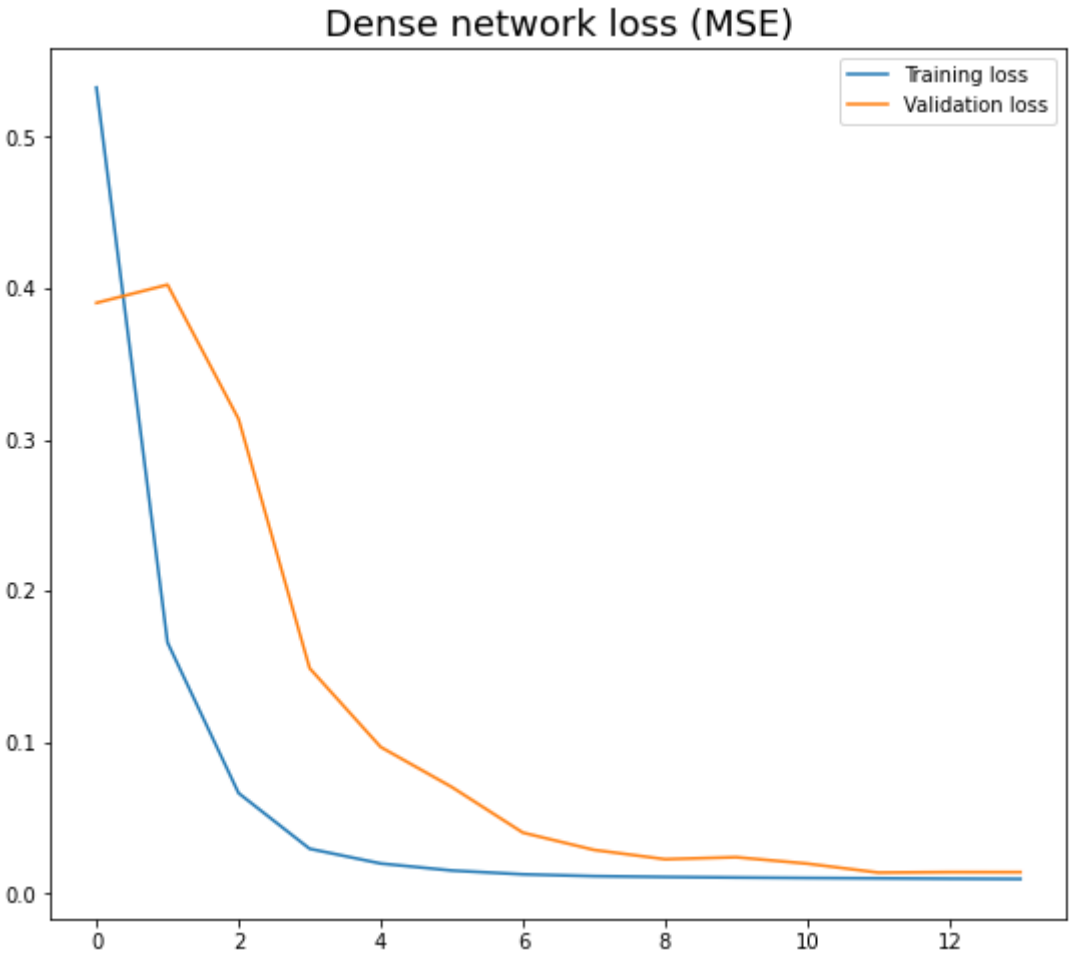


Figure 45 - Training- and validation loss (dense network zinc) . Y-axis displays the MSE and the x-axis shows the epochs during training

The loss measured by the MSE for the training set and validation set for the dense model predicting the zinc price is shown in figure 45. We can see the diminishing returns of training when the number of epochs is increased. This is especially apparent from around epoch three for the training data and around epoch six for the validation data. The model does not seem to overfit, as the difference between the training loss and validation loss decreases over time and is almost eliminated after training. The reason why the training stopped at 13 epochs is that the *Patience* parameter in Keras' EarlyStopping (Chollet & Others, 2015) function has been set to three which means the number of epochs with no improvement after which training will be stopped. This applies for all the other trainable models as well.

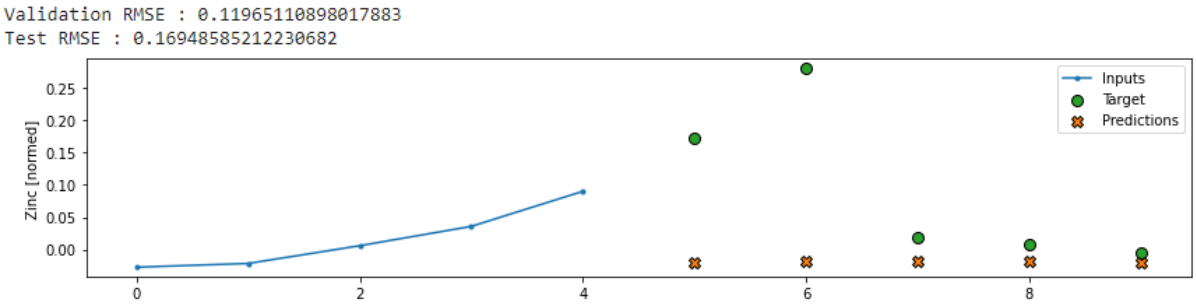


Figure 46 - Prediction of a single sequence (dense network zinc)

Figure 46 shows an example sequence where the dense model has predicted the output for five timesteps. The predictions from the dense model does not change much from one timestep to another but they are not equal to the baseline. Since we know that the prices does not change much from one timestep to another, the kernel_initializer parameter has been set to initialize to zeros using the TensorFlow zeros initializer (Chollet & Others, 2015).This is the case for all the other models as well and can be seen in the appendix. To be able to predict multiple timesteps ahead the final layer of the model reshapes to match the number of output steps times the number of targets to predict. In this case we only predict the price of one metal at the time, five timesteps ahead. This approach yields a higher RMSE than the baseline with a validation RMSE of 0,1197 and a test RMSE of 0,1695.

The single-layer LSTM model

The single-layer LSTM model for forecasting the zinc price consists of a single LSTM layer with 64 units, yielding a total of 20 037 parameters. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output at the final timestep. A detailed summary of the model can be found in the appendix.

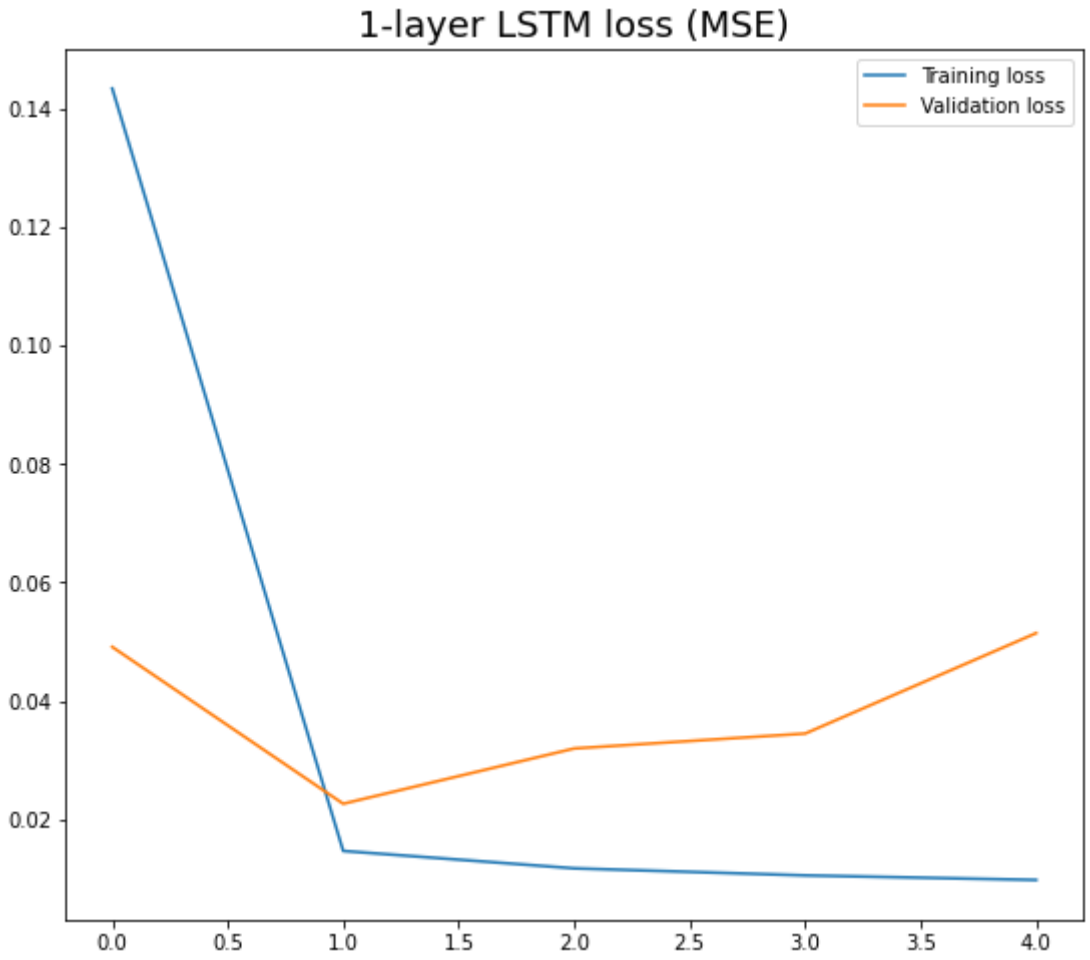


Figure 47 - Training- and validation loss (single-layer LSTM zinc) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer LSTM model is shown in figure 47 and we can see the training loss is not significantly reduced after the first epoch. The validation loss is slightly higher than the training loss and increasing after the first epoch which may indicate that the model is overfitting to the training data.

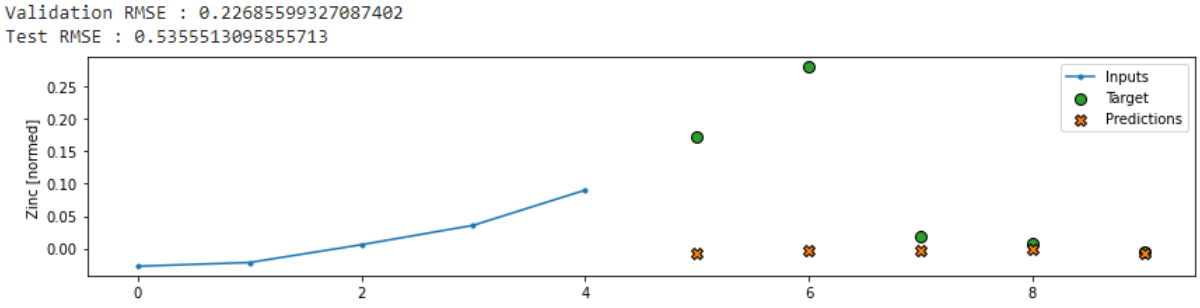


Figure 48 - Prediction of a single sequence (single-layer LSTM zinc)

The same example sequence as for the other zinc models is shown in figure 48 and like the two previous models, the LSTM model seem to predict values near the last known

observation in this specific sample, and in a very similar way as the dense model. As with the dense model, the weights are initialized to zero yielding small changes from the last known observation. This is however just one sample and would differ if another sample were picked. In total over the entire validation and test set, the single-layer LSTM model yielded a RMSE of 0,2269 and 0,5356, respectively. A difference between the validation and test RMSE as large as this indicates that the model has been overfit to the training data, as it does not perform particularly well on neither the validation nor test data compared to the other models.

The multi-layer LSTM model

The multi-layer LSTM model used for forecasting the copper price is built up with two LSTM layers consisting of 32 units and a dense layer with 32 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first layer and False on the second layer. This structure is required when stacking LSTM layers where all layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. In total this structure yields 15 301 parameters giving the model a smaller capacity than the single-layer LSTM. A detailed summary of the model can be found in the appendix.

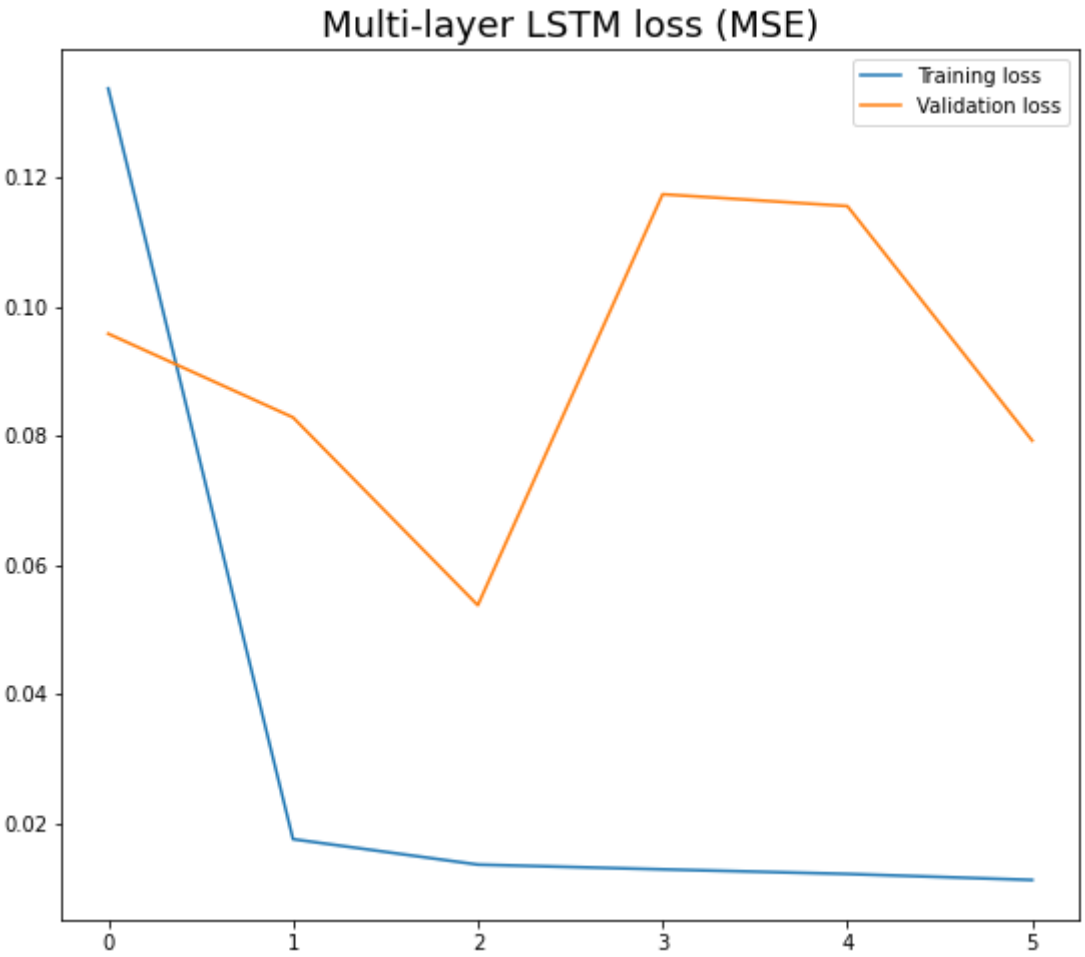


Figure 49 - Training- and validation loss (multi-layer LSTM zinc) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training process for the multi-layer LSTM model stops after five epochs and have most of the progress in the first epoch. The training loss have a steady decline over all the epochs starting to flatten more out after the first epoch. The validation loss is more volatile and starts to increase after the second epoch. As with the single layer LSTM, the model seems to overfit as it struggles to generalize on both the validation data and especially the test data.

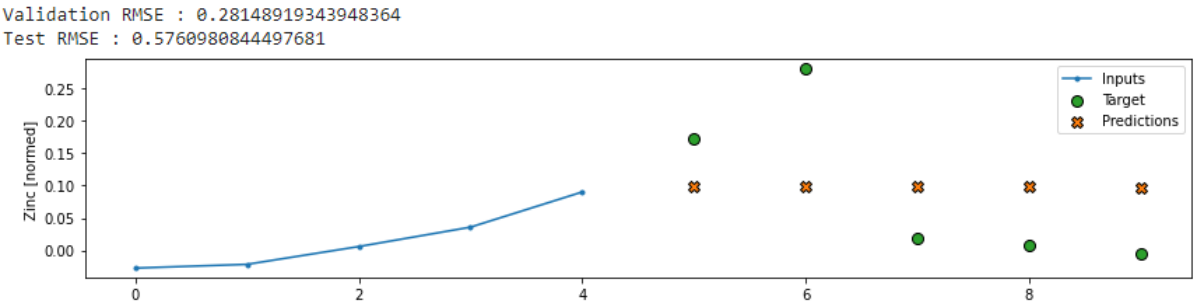


Figure 50 - Prediction of a single sequence (multi-layer LSTM zinc)

From the example shown in figure 50 we can see that the model seems to predict the sequence in a similar way to the previous models and the predictions does not change much from one timestep to another. All in all, the model performs worse than the baseline but better than the dense and single-layer LSTM models with a validation RMSE of 0,2815 and a test RMSE of 0,5761. As with the single-layer LSTM model, the test performance is poor as the test RMSE is more than double of the validation RMSE.

The single-layer GRU model

As an alternative to the LSTM layers, this model applies a single GRU layer where the number of units is set to 16, yielding a total of 1 525 parameters which is the around the average number of parameters thus far for single-layer LSTM and GRU models. The return_sequence parameter described in chapter 3.5.1 is set to false so that the layer only returns the output at the final timestep. A detailed summary of the model can be found in the appendix.

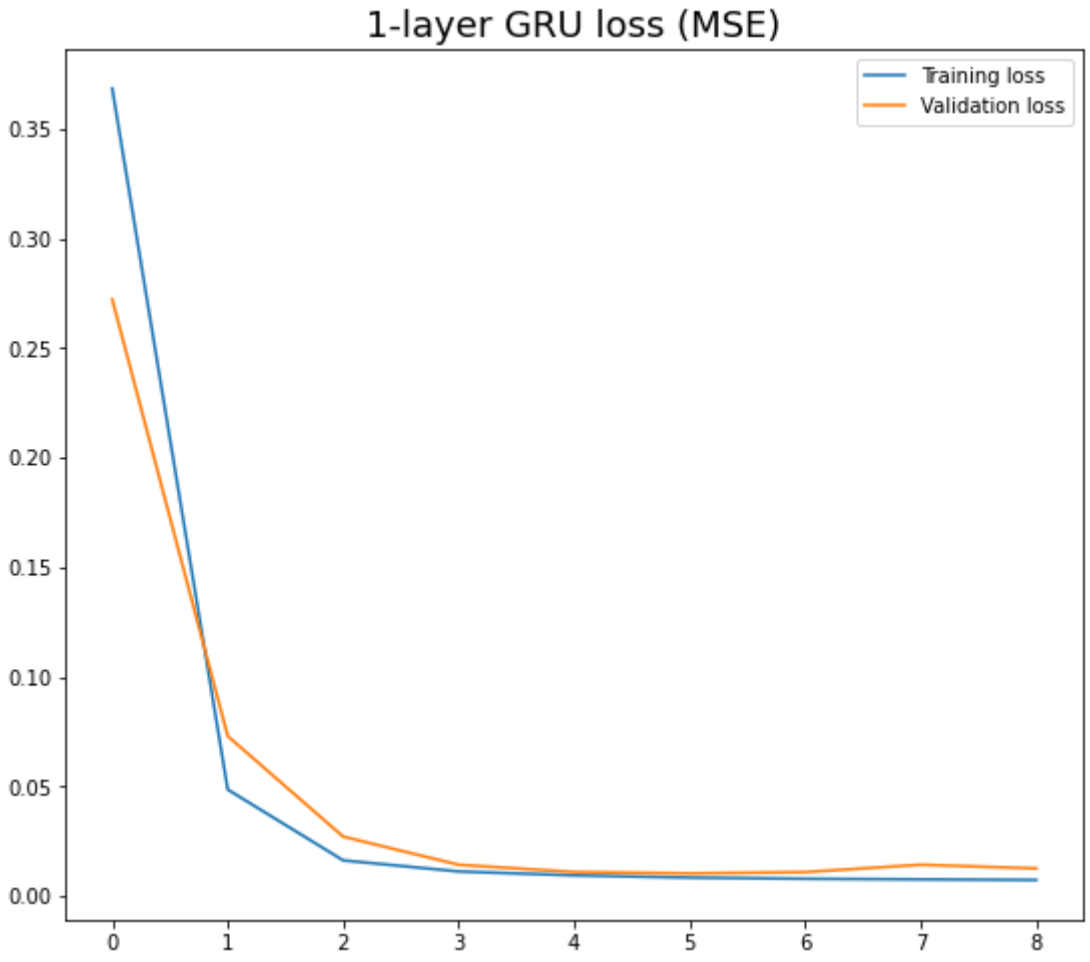


Figure 51 - Training- and validation (single-layer GRU zinc) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the single-layer GRU model shown in figure 51, and we can see that the majority of the reduction in the loss happens before the third epoch. The validation loss is slightly higher than the training loss after the first epoch, but as the number of epochs increases the difference between the training and validation loss does not increase except for the last two epochs where the validation loss has a slight increase. This indicates that the model does not overfit to the training data and generalizes well on the validation- and test data.

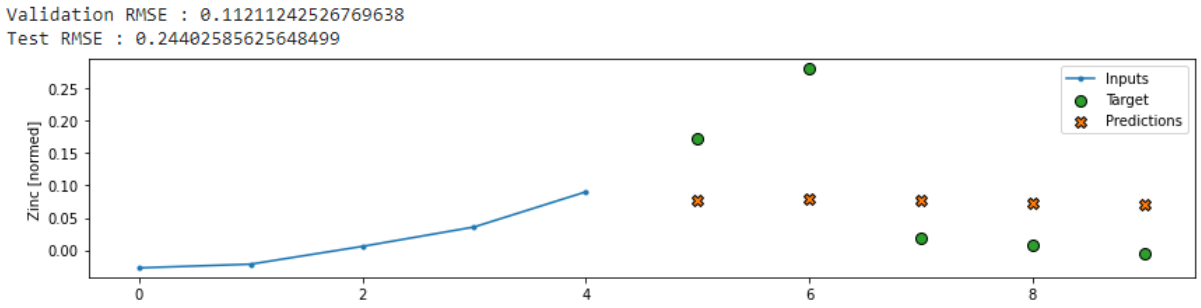


Figure 52 - Prediction of a single sequence (single-layer GRU zinc)

The example shown in figure 52 we can see that the model predicts in a similar fashion to the LSTM models where all the predictions comes in an almost straight line based on the observation on the fourth timestep. This however is just one sample and in total the model performs worse than the baseline but better than all the other models for zinc on the validation data with a RMSE of 0,1121 and a test RMSE better than all other zinc models besides the dense model with a RMSE of 0,2440. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes fairly well, which seems to be truer for the single-layer GRU model than the LSTM models.

The multi-layer GRU model

The multi-layer GRU model is built up in the same fashion as the multi-layer LSTM model for zinc with two GRU layers consisting of 32 units and a dense layer with 32 units. The return_sequence parameter described in chapter 3.5.1 is set to True for the first layer and False in the second layer. This is a required structure when stacking GRU layers where all layers need to return outputs for the full sequence except the last layer, which only returns its output from the final timestep. In total this structure yields 11 973 parameters giving the model a solid capacity but with fewer parameters than the multi-layer LSTM, even though the model architecture is the same. This is because of the way the GRU layer is built up compared

to the LSTM layer, which is described in detail in chapter 3.5.4 and 3.5.5. A detailed summary of the model can be found in the appendix.

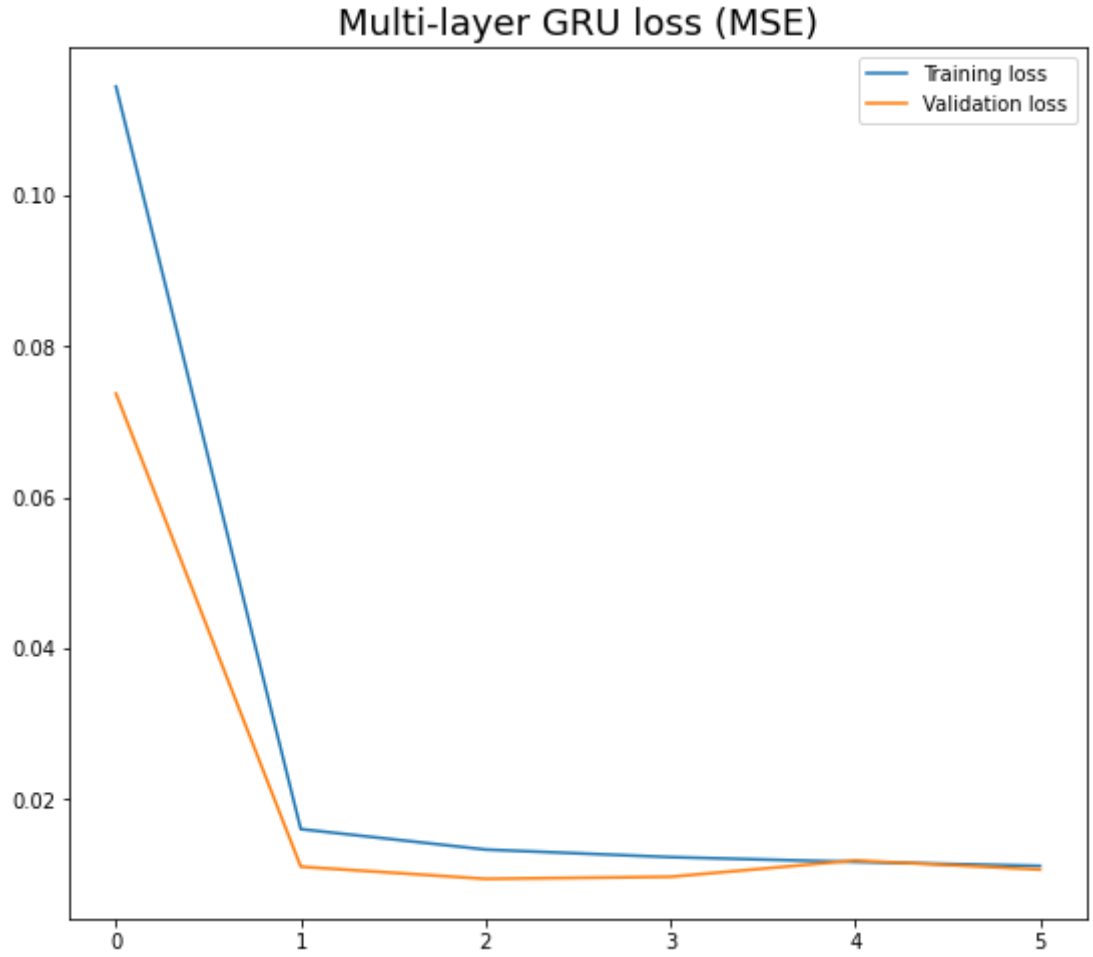


Figure 53 - Training- and validation loss (multi-layer GRU zinc) . Y-axis displays the MSE and the x-axis shows the epochs during training

The training of the multi-layer GRU model shown in figure 53. Training stops after five epochs where the training loss is reduced the most in the first epoch with a steady decrease until epoch five. The validation loss for this model is slightly lower than the training loss, indicating that the model is generalizing well. Results like these does not indicate overfitting.

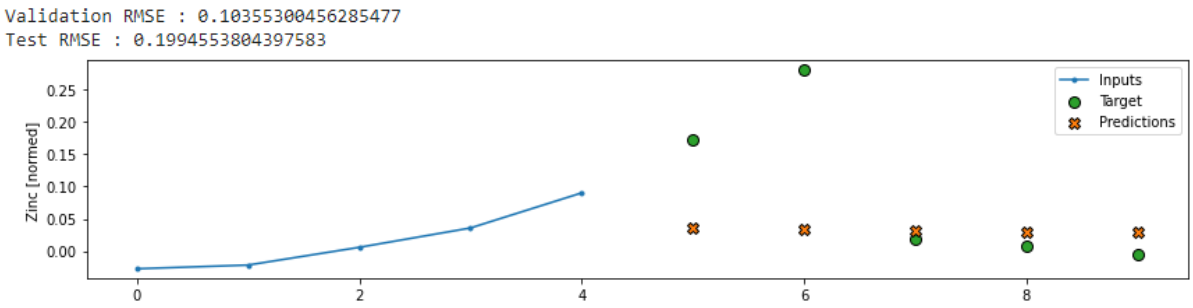


Figure 54 - Prediction of a single sequence (multi-layer GRU zinc)

The example in figure 54 shows that the multi-layer GRU predicts this sequence similarly to the other models for zinc. Keep in mind this is just one sample and in total the model performs worse than the baseline but has the best validation RMSE among all the other models at 0,1036. It also beats both the LSTM models and the single-layer GRU model on test RMSE only beaten by the dense model and the baseline with a test RMSE of 0,1995. A small difference between the validation and test RMSE indicates a model that does not overfit and generalizes well, which is somewhat the case for this model.

5. Discussion

In this chapter the results from chapter 4 will be discussed and analyzed. Since the metric RMSE measures the error in absolute terms one cannot compare the results directly between the different metals as they are not necessarily on the same scale. Even though they are normalized by subtracting the mean and dividing by the standard deviation of the training data, these means and standard deviations are not equal between the metals. If a metric such as MAPE was used that measures the error as a percentage it would be interpretable across different metals, but as discussed earlier in chapter 3.6 this method often leads to zero-division or extreme values due to very small values in the denominator. Therefore, the focus of the discussion will be on the suitability of RNN for short-term metal price forecasts for the specific metals. The first three subchapter will go over each metal individually and chapter 5.4 will discuss pros and cons of the approach in this paper and what may have affected these results.

5.1 Aluminum

Figure 55 summarizes the forecasting results for the validation- and test set for Aluminum presented in chapter 4.1.

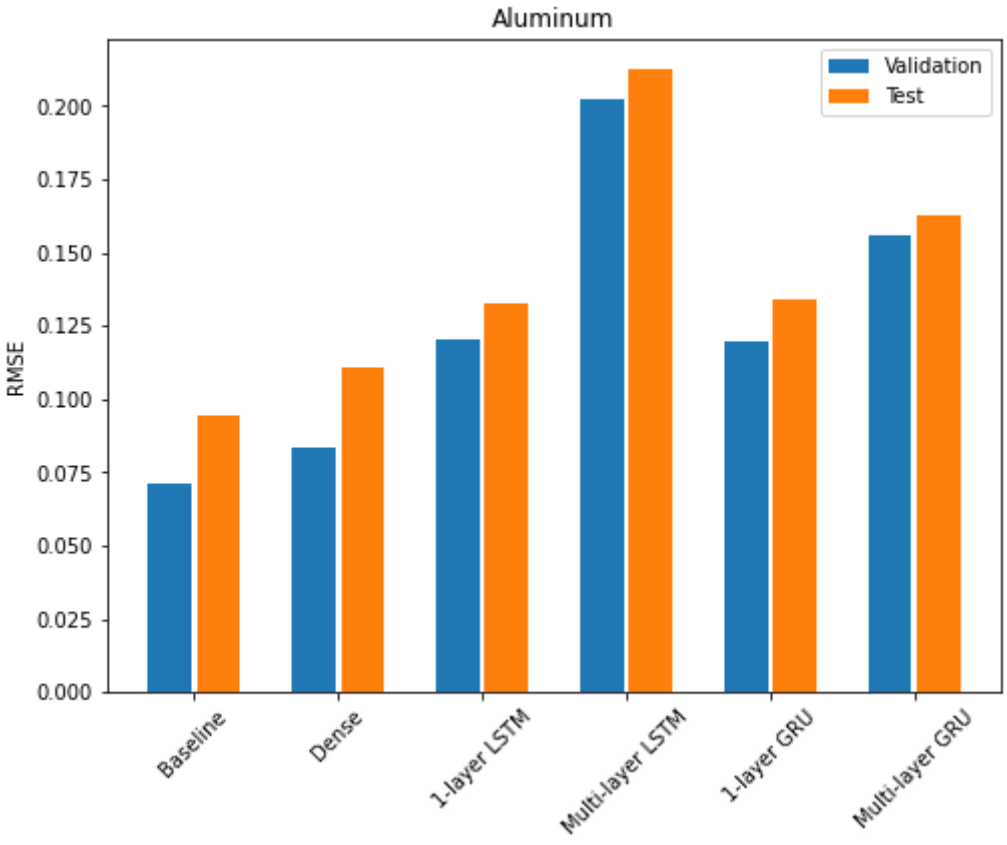


Figure 55 - Validation and test RMSE aluminum

The first and possibly most important finding when forecasting the aluminum price in the short-term is that the “last-known observation” baseline outperforms all the other models where the simple dense model comes closest. The multi-layer LSTM model had by-far the worst performance on the aluminum price with a RMSE of 0,2122 which is more than double of the baseline at 0,0942 seen in figure 56.

```
Baseline    : 0.0942
Dense       : 0.1106
1-layer LSTM: 0.1327
Multi-layer LSTM: 0.2122
1-layer GRU : 0.1337
Multi-layer GRU: 0.1630
```

Figure 56 - Test RMSE Aluminum

Another interesting finding is that the single-layer RNNs performed better than the multi-layer RNNs where the single-layer LSTM and GRU model had a comparable RMSE on the test set with 0,1327 and 0,1337 respectively. The multi-layer RNNs had problems in training as seen in figure 27 and 31 where the validation results were erratic and moving a lot in both directions between epochs. Out of the two multi-layer models, the GRU outperformed the LSTM model with a test RMSE of 0,1630 and 0,2122 respectively. In general, we had issues finding good multi-layer models that found a compromise between complexity and regularization, or over- and underfitting, which to some degree can be seen for copper and zinc as well in the next two sub-chapters. As described earlier, the choice of layers, parameters, regularization etc. is often more an art than a science, so more time could be spent testing out various combinations of weights and layers. However, this is very time consuming and at one point we must call it a day and accept the results even if it is not as good as desired.

5.2 Copper

Figure 57 summarizes the forecasting results for the validation- and test set for copper presented in chapter 4.2.

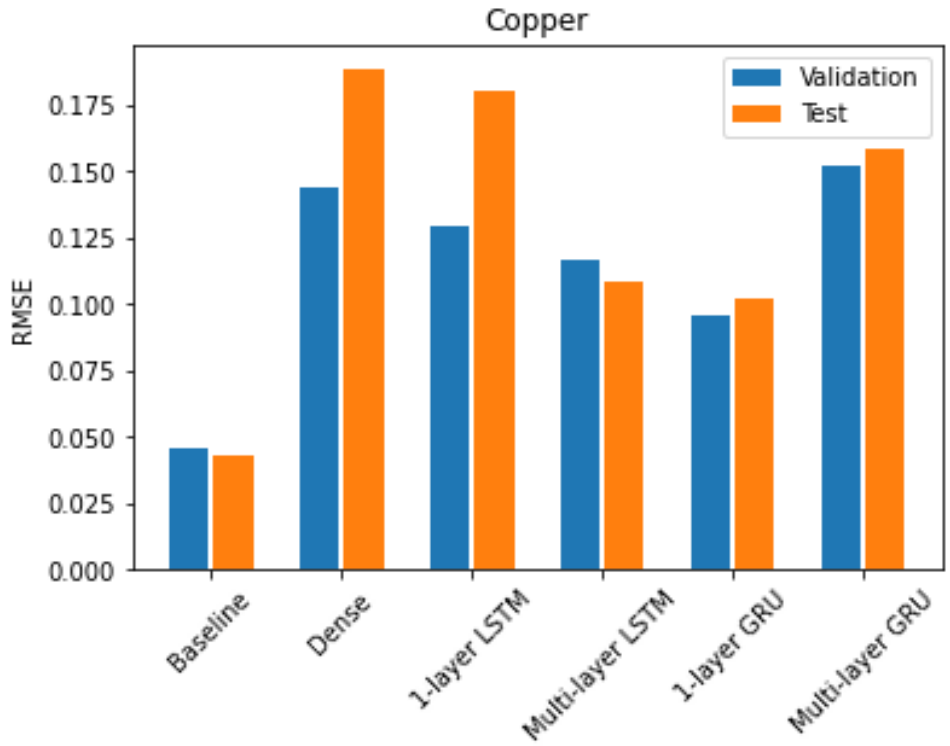


Figure 57 - Validation and test RMSE copper

As with aluminum, the "last-known observation" baseline outperforms all the other models but in an even more significant manner. The dense model performed worse on copper than for aluminum relatively to the other models. When it comes to the best performing models, the single-layer GRU and the multi-layer LSTM performs well with a RMSE of 0,1017 and 0,1083 respectively. In comparison the baseline had a RMSE of 0,0432 which is less than half of the second best performing models. For the dense model and the single-layer LSTM it appears that the performance on the test set is much worse than for the validation set compared to the other models. These models seem to struggle generalizing to new unseen data.

```
Baseline    : 0.0432
Dense       : 0.1886
1-layer LSTM: 0.1802
Multi-layer LSTM: 0.1083
1-layer GRU : 0.1017
Multi-layer GRU: 0.1587
```

Figure 58 - Test RMSE copper

Fewer problems in training appeared for copper than aluminum, and most of the models had naturally looking loss curves with slightly higher validation RMSE than training RMSE. An interesting point for the models forecasting copper is that the best performing model besides the baseline, the single-layer GRU, is the model with the fewest parameters besides the dense model with only 573 weights. When choosing a model one often wants to find the least complex models that solves the problem. The single-layer GRU with only eight units seem to be the best choice of model for predicting the copper price five days ahead based on the previous five days based on the various models tested in this paper besides the baseline. On the other hand, the baseline model predicting the same price for all five days outperforms all the other models by a lot, which leaves the question if making a model forecasting the price is necessary in the first place.

5.3 Zinc

The RMSE for the validation- and test set for the zinc price can be seen in figure 59

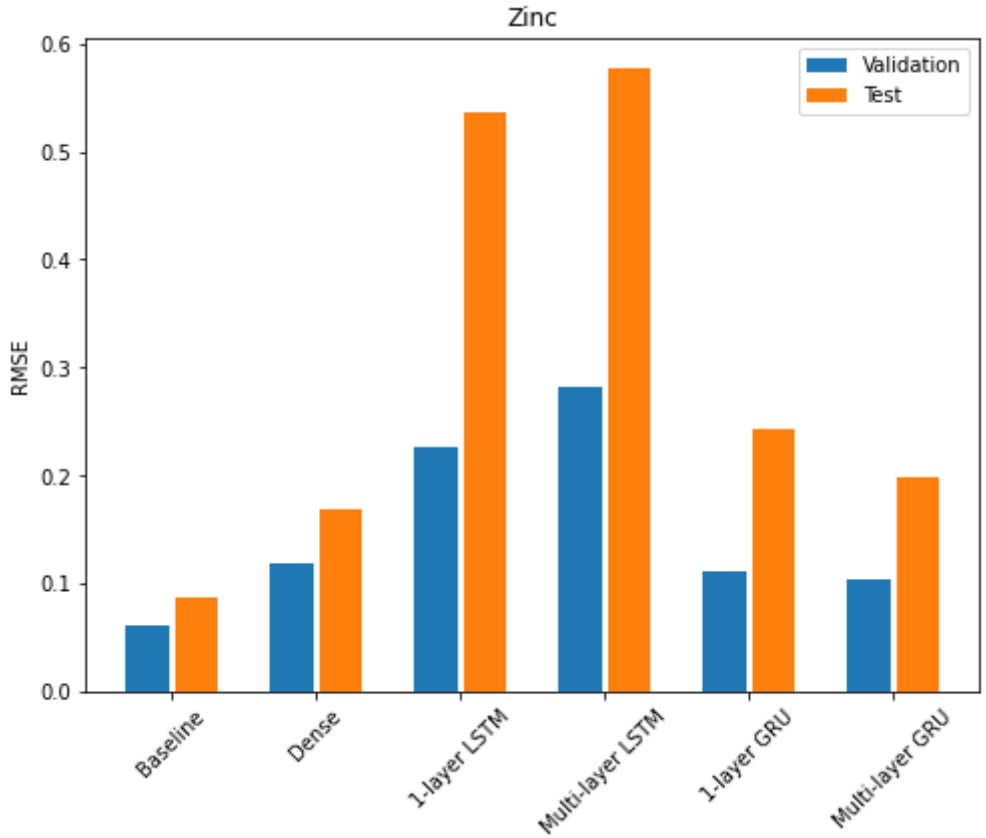


Figure 59 - Validation and test RMSE zinc

The first and possibly most important finding when forecasting the zinc price in the short-term is that the “last-known observation” baseline outperforms all the other models just like for aluminum and copper. The test RMSE for the baseline were 0,0876 where the dense model was the second best with at test RMSE of 0,1695.

```
Baseline    : 0.0876
Dense       : 0.1695
1-layer LSTM: 0.5356
Multi-layer LSTM: 0.5761
1-layer GRU : 0.2440
Multi-layer GRU: 0.1995
```

Figure 60 - Test RMSE zinc

Both the single-layer and multi-layer LSTM models have massive RMSE on the test set with 0,5356 and 0,5761 respectively, which is by far the worst of the models. Both these models have a high capacity where the single-layer model has 64 units in its layer yielding a total of 20 037 parameters and the multi-layer with a total of 15 301 parameters. This major difference between the training-, validation- and test performance indicates overfitting, which also was the case when other combinations of layers and units were tested. Finding a combination of good performance in-sample without overfitting and performing well out-of-sample can be hard and was not achieved for these two models.

In comparison with the LSTM models, the GRU models had a more similar performance in-sample and out-of-sample with validation RMSE around 0,1 and test RMSE of 0,2440 and 0,1995 for the single- and multi-layer models, respectively. These models with the single-layer GRU in particular performed well in training as well as validation and testing compared to the other models, where the loss in training decreased at a steady rate where the validation and training loss were very similar. This was also the case for the multi-layer GRU but over fewer training epochs.

5.4 General considerations

After reviewing the forecasting results it is important to note that none of the models outperformed the “last-known observation” baseline. This begs the question if RNN are suitable for this type of problem and based on the results in this paper the answer is probably no. Spending time and recourses developing and maintaining a complex recurrent neural network that performs worse than predicting the price at day two, three, four, five and six is equal to the price at day one does not make much sense. However, this does not mean that

RNN are not suitable for time series regression and commodity price forecasts in general but that for this specific dataset and timeframe it did not work well with the model specifications chosen. If other combinations of layers and units or a different timeframe were chosen, the results could have improved and is an area where further work can be done.

It is important to note that relative to for example sentiment analysis or other text modeling, the dataset used in this paper is small. 6524 observations for ten different features got split into 4566 observations for the training set, 1305 for the validation set and 653 for the training set. When this again got split into sequences of ten (5+5) observations and stored in batches with 32 sequences, yielding 143 training batches, 41 validation batches and 21 training batches. Other commodities or financial instruments that reports their prices more often such as for example Bitcoin could be another interesting task for RNN to tackle, since one would have access to a much larger dataset.

When ignoring the baseline, there seem to be no clear winner between the LSTM and GRU models on which models performed the best as the results differ between the different commodities. However, the single-layer GRU seem to be the model that has the most even performance on all three commodities and had good performance both in- and out-of-sample relative to the other trainable models. An advantage GRU has over LSTM is the complexity and reduced computation required when training the model. For larger networks and datasets this advantage gets more apparent than in this paper.

When weighing the pros and the cons when using an RNN approach to commodity price forecasting, the cons seem to outweigh the pros in this present work. Developing a RNN is time consuming, requires domain knowledge and require a good amount of computational power. It is also a black-box approach where it is hard for the researcher to figure out what the model learned or if it actually learned anything. From simpler models such as linear regression one can inspect the coefficients to see how much each feature contribute to the output of the model, which is not the case for RNN. On the other side RNN has some advantages when it comes to handling time information in the data which is important for timeseries data.

As mentioned previously, the future price of a commodity does not necessarily depend on previous prices and using historical data as input when forecasting commodity prices will always have its flaws. Models are only as good as the data it is given and could be a contributing factor to the results in this paper.

6. Conclusion and further work

The goal of this thesis was to determine if RNN could be used successfully to forecast the prices of non-ferrous metals in the short-term. After identifying other variables that could influence the price from a fundamental point of view, these variables were analyzed using descriptive statistics and plots and these results aligned with the analysis done on the fundamentals of these variables. After the dataset were split into training, validation and test sets it was then split into smaller sequences consisting of ten sequential observations including all ten variables and the "time-of-year signal" where the five first observations were input and the last five were output.

Four RNN networks were trained as well as a "last-known observation" baseline and a simple dense model. These RNNs failed to out-perform the baseline approach for both aluminum, copper and zinc, where the performance on aluminum were the best with a RMSE of 0,1327 and 0,1337 for the single-layer- LSTM and GRU, compared to 0,0942 for the baseline on the test data. For copper and zinc the RNNs struggled in general to make good predictions on the test data, which was seen by large test RMSE compared to validation RMSE. The performance in the training- and validation greatly differed from the performance on the test data for some of the models, especially LSTM models forecasting zinc but also too some extend the GRU models forecasting zinc. For copper, the GRU models performed best besides the baseline, but still had a test RMSE of more than double of the baseline.

Based on the results from this work, RNNs does not seem to be a great choice for forecasting the aluminum-, copper-, and zinc price in the short-term. However, this does not mean that RNNs are not suitable for similar problems, but the specific model specifications used in this work did not out-perform the baseline approach. Further work can be done on short-term metal price forecasting, where a different time horizon could be used, another combination of explanatory variables could be used as well as other forecasting techniques. When it comes to the forecasting horizon, it is a difficult compromise to make as with a shorter time horizon the forecast itself gives less value to the user. If a longer forecasting horizon is chosen, the number of sequences in the dataset will decrease and may negatively impact the results. If one is able to obtain intra-day data on a minute- or hourly timescale then one might have more wiggle room when it comes to the time horizon. As with other explanatory variables, this paper did not include prices of raw material such as iron ore and other energy commodities like coal which might have been a better proxy for the energy cost. The demand of these metals has a correlation with the general economic growth (Zhong et al., 2019) so an indicator

of the economic growth in various countries or regions could be used as an explanatory variable. When it comes to the models themselves, this paper implemented “one-shot” models where all the five predictions were made at the same time. Another approach to this is an autoregressive approach where the output at one timestep is used as input in the next timestep directly. Other model architectures such as the residual networks (ResNet) or the LSTNet described in the literature section could be used. ResNet could take advantage of the fact that the prices do not change much from one timestep to another and instead of predicting the price itself to predict the change in price. The ResNet is built in a way so that each layer adds to the accumulated result and by initializing the weights to zeros the model will predict small changes. The LSTNet has shown promising results combining convolutional layers and recurrent layers and would be interesting to utilize on metal price forecasting.

Citations

Ahti, V. (2009). *Forecasting Commodity Prices with Nonlinear Models*.

Anaconda. (2020). *The State of Data Science 2020 - Moving from hype toward maturity*.

Bandara, K., Bergmeir, C. & Hewamalage, H. (2021). Recurrent Neural Networks for Time Series Forecasting: Current status and future directions. *International Journal of Forecasting*, 37 (1): 388-427. doi: https://doi.org/10.1016/j.ijforecast.2020.06.008.

Box, G. E. P., Jenkins, G. M. & STATISTICS., W. U. M. D. o. (1970). *Time Series Analysis: Forecasting and Control*: Holden-Day.

Carter, C. A. (2012). *Futures and Options Markets: An Introduction*: RebelText.

Cho, K., Chung, J., Gulcehre, C. & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

Chollet, F. & Others. (2015). *Keras*: GitHub. Available at: https://github.com/fchollet/keras.

Chollet, F. (2017). *Deep Learning with Python*: Manning Publications Company.

Datastream. (2021). Eikon. Refinitiv online database (Datastream).

Figuerola-Ferretti, I. & Gilbert, C. (2008). Commonality in the LME aluminum and copper volatility processes through a FIGARCH lens. *Journal of Futures Markets*, 28: 935-962. doi: 10.1002/fut.20338.

Gal, Y. (2016). *Uncertainty in Deep Learning*.

Ganapathyraman, S., Sugumaran, S., Komatheswari, T., Surulivel, S. T., Selvabaskar, S., Anand, V. V. & Rengarajan, V. (2018). A study on relationship between price of us dollar and selected commodities. *International Journal of Pure and Applied Mathematics*, 119: 203-224.

Greff, K., Srivastava, R., Koutník, J., Steunebrink, B. & Schmidhuber, J. (2015). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28. doi: 10.1109/TNNLS.2016.2582924.

Harris, C. R., Millman, K. J., van der Walt, S. & J Gommers, R. a. V., Pauli and Cournapeau, David and Wieser, Eric and Taylor, Julian and Berg, Sebastian and Smith, Nathaniel J. and Kern, Robert and Picus, Matti and Hoyer, Stephan and van Kerkwijk, Marten H. and Brett, Matthew and Haldane, Allan and Fernández del Río, Jaime and Wiebe, Mark and Peterson, Pearu and Gérard-Marchant, Pierre and Sheppard, Kevin and Reddy, Tyler and Weckesser, Warren and Abbasi, Hameer and Gohlke, Christoph and Oliphant, Travis E. (2020). *Array programming with NumPy*. Nature. Available at: https://numpy.org/.

Harrison, D. & Rubinfeld, D. (1978). Hedonic housing prices and the demand for clean air. *Journal of Environmental Economics and Management*, 5: 81-102. doi: 10.1016/0095-0696(78)90006-2.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-term Memory. *Neural computation*, 9: 1735-80. doi: 10.1162/neco.1997.9.8.1735.

Hochreiter, S., Bengio, Y., Frasconi, P. & Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In Kremer, S. C. & Kolen, J. F. (eds) *A Field Guide to Dynamical Recurrent Neural Networks*: IEEE Press.

Hunter, J. D. (2007). *Matplotlib: A 2D graphics environment*. Available at: https://matplotlib.org/.

ICE. (2020). *US Dollar Index Futures*. Intercontinental Exchange: Intercontinental Exchange. Available at: https://www.theice.com/products/194/US-Dollar-Index-Futures (accessed: 02.12.2020).

ILZSG. (2021). *End Uses*: International Lead and Zinc Study Group. Available at: https://www.ilzsg.org/static/enduses.aspx?from=1.

Jue, W., Zhen, W., Xiang, L. & Hao, Z. (2019). Artificial bee colony-based combination approach to forecasting agricultural commodity prices. *International Journal of Forecasting*. doi: https://doi.org/10.1016/j.ijforecast.2019.08.006.

Labys, W. C. (2006). *Modeling and Forecasting Primary Commodity Prices*. 1 ed.: Routledge.

Lai, G., Chang, W.-C., Yang, Y. & Liu, H. (2018). *Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks*. The 41st International ACM SIGIR Conference on Research

& Development in Information Retrieval, Ann Arbor, MI, USA, pp. 95–104: Association for Computing Machinery.
LeCun, Y. & Cortes, C. (2010). MNIST handwritten digit database.
Lehfeldt, R. A. (1914). The Elasticity of Demand for Wheat. *The Economic Journal*, 24 (94): 212-217. doi: 10.2307/2222421.
Liu, C., Hu, Z., Li, Y. & Liu, S. (2017). Forecasting copper prices by decision tree learning. *Resources Policy*, 52: 427-434. doi: https://doi.org/10.1016/j.resourpol.2017.05.007.
LME. (2021). *Annual Trading Volumes*: London Metal Exchange. Available at: https://www.lme.com/Market-Data/Reports-and-data/Volumes/Annual-volumes.
Malliaris, A. & Malliaris, M. (2009). *Time series and neural networks comparison on gold, oil and the euro*.
Martín Abadi, A. A., Paul Barham, Eugene Brevdo,, Zhifeng Chen, C. C., Greg S. Corrado, Andy Davis,, Jeffrey Dean, M. D., Sanjay Ghemawat, Ian Goodfellow,, Andrew Harp, G. I., Michael Isard, Rafal Jozefowicz, Yangqing Jia,, Lukasz Kaiser, M. K., Josh Levenberg, Dan Mané, Mike Schuster,, Rajat Monga, S. M., Derek Murray, Chris Olah, Jonathon Shlens,, Benoit Steiner, I. S., Kunal Talwar, Paul Tucker,, Vincent Vanhoucke, V. V., Fernanda Viégas,, Oriol Vinyals, P. W., Martin Wattenberg, Martin Wicke, & Yuan Yu, a. X. Z. (2015). *Tensorflow: Large-Scale Machine Learning on Heterogeneous Systems*. Available at: https://www.tensorflow.org/.
McKinney, W. & Others. (2010). *Data structures for statistical computing in python*. Available at: https://pandas.pydata.org/docs/.
Olah, C. (2015). *Understanding LSTM Networks*. Github.
Ouyang, H., Wei, X. & Wu, Q. (2019). Agricultural commodity futures prices prediction via long- and short-term time series network. *Journal of Applied Economics*, 22 (1): 468-483. doi: 10.1080/15140326.2019.1668664.
Pontius, R., Thontteh, O. & Chen, H. (2008). Components of information for multiple resolution comparison between maps that share a real variable. *Environmental and Ecological Statistics*, 15: 111-142. doi: 10.1007/s10651-007-0043-y.
Raschka, S. & Mirjalili, V. (2017). *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow, 2nd Edition*: Packt Publishing.
S&P. (2021). *S&P GSCI Natural Gas*. Available at: https://www.spglobal.com/spdji/en/indices/commodities/sp-gsci-natural-gas/#overview.
Slutsky, E. E. (1927). Slozhenie sluchainykh prichin, kak istochnik tsiklicheskikh protsessov. *Voprosy kon''yunktury*, 3 (1): 34-64.
Tayefi, M. & Ramanathan, T. V. (2016). An Overview of FIGARCH and Related Time Series Models. *Austrian Journal of Statistics*, 41 (3): 175–196. doi: 10.17713/ajs.v41i3.172.
Tensorflow. (2021). *Time series forecasting*. Available at: https://www.tensorflow.org/tutorials/structured_data/time_series#data_windowing.
Theil, H. & Collection, K. M. R. (1971). *Principles of Econometrics*: Wiley.
Waskom, M., Botvinnik, O., O'Kane, D., and, P. H., and, S. L., and, D. C. G., and, T. A., and, Y. H., and, J. B. C., and, J. W., et al. (2017). *mwaskom/seaborn: v0.8.1 (September 2017)*: Zenodo. Available at: https://seaborn.pydata.org/.
Yule, G. U. (1927). *On a method of investigating periodicities in disturbed series, with special reference to Wolfer's sunspot numbers*: Harrison and Sons.
Zhong, M., He, R., Chen, J. & Huang, J. (2019). Time-varying effects of international nonferrous metal price shocks on China's industrial economy. *Physica A: Statistical Mechanics and its Applications*, 528: 121299. doi: https://doi.org/10.1016/j.physa.2019.121299.

Appendix

Baseline (same for all metals)

```
class MultiStepLastBaseline(tf.keras.Model):
  def call(self, inputs):
    return tf.tile(inputs[:, -1:, :1], [1, OUT_STEPS, 1])
```

Dense model (same for all metals)

```
multi_dense_model = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1),
    tf.keras.layers.Dense(OUT_STEPS*num_features
                          ,kernel_initializer=tf.initializers.zeros
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_5 (Flatten)          (None, 60)                0
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 61
_________________________________________________________________
dense_16 (Dense)             (None, 5)                 10
_________________________________________________________________
reshape_9 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 71
Trainable params: 71
Non-trainable params: 0
_________________________________________________________________
```

Single-layer LSTM Aluminum

```
multi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.LSTM(16, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_6 (LSTM)                (None, 16)                1856
_________________________________________________________________
dense_19 (Dense)             (None, 5)                 85
_________________________________________________________________
reshape_12 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 1,941
Trainable params: 1,941
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer LSTM Aluminum

```
multi_layer_lstm_model = tf.keras.Sequential([

    tf.keras.layers.LSTM(128, return_sequences=True,),
    tf.keras.layers.LSTM(128, return_sequences=True),
    tf.keras.layers.LSTM(128, return_sequences=False),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 5, 128)            72192
_________________________________________________________________
lstm_2 (LSTM)                (None, 5, 128)            131584
_________________________________________________________________
lstm_3 (LSTM)                (None, 128)               131584
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_4 (Dense)              (None, 5)                 645
_________________________________________________________________
reshape_2 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 352,517
Trainable params: 352,517
Non-trainable params: 0
_________________________________________________________________
```

Single-layer GRU Aluminum

```
multi_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(32, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_4 (GRU)                  (None, 32)                4416
_________________________________________________________________
dense_11 (Dense)             (None, 5)                 165
_________________________________________________________________
reshape_8 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 4,581
Trainable params: 4,581
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer GRU Aluminum

```
multi_layer_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.GRU(128, return_sequences=False),
    tf.keras.layers.Dense(128),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_5 (GRU)                  (None, 5, 128)            54528
_________________________________________________________________
gru_6 (GRU)                  (None, 5, 128)            99072
_________________________________________________________________
gru_7 (GRU)                  (None, 128)               99072
_________________________________________________________________
dense_12 (Dense)             (None, 128)               16512
_________________________________________________________________
dense_13 (Dense)             (None, 5)                 645
_________________________________________________________________
reshape_9 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 269,829
Trainable params: 269,829
Non-trainable params: 0
_________________________________________________________________
```

Single-layer LSTM Copper

```
multi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 32)                5760
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 165
_________________________________________________________________
reshape_1 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 5,925
Trainable params: 5,925
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer LSTM Copper

```
multi_layer_lstm_model = tf.keras.Sequential([

    tf.keras.layers.LSTM(32, return_sequences=True,),
    tf.keras.layers.LSTM(32, return_sequences=False),
    tf.keras.layers.Dense(32),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_11 (LSTM)               (None, 5, 32)             5760
_________________________________________________________________
lstm_12 (LSTM)               (None, 32)                8320
_________________________________________________________________
dense_17 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_18 (Dense)             (None, 5)                 165
_________________________________________________________________
reshape_10 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 15,301
Trainable params: 15,301
Non-trainable params: 0
_________________________________________________________________
```

Single-layer GRU Copper

```
multi_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(8, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 8)                 528
_________________________________________________________________
dense_5 (Dense)              (None, 5)                 45
_________________________________________________________________
reshape_3 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 573
Trainable params: 573
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer GRU Copper

```
multi_layer_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.GRU(32, return_sequences=False),
    tf.keras.layers.Dense(32),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_1 (GRU)                  (None, 5, 32)             4416
_________________________________________________________________
gru_2 (GRU)                  (None, 32)                6336
_________________________________________________________________
dense_6 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_7 (Dense)              (None, 5)                 165
_________________________________________________________________
reshape_4 (Reshape)          (None, 5, 1)              0
=================================================================
Total params: 11,973
Trainable params: 11,973
Non-trainable params: 0
_________________________________________________________________
```

Single-layer LSTM Zinc

```
multi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.LSTM(64, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_11 (LSTM)               (None, 64)                19712
_________________________________________________________________
dense_17 (Dense)             (None, 5)                 325
_________________________________________________________________
reshape_10 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 20,037
Trainable params: 20,037
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer LSTM Zinc

```
multi_layer_lstm_model = tf.keras.Sequential([

    tf.keras.layers.LSTM(32, return_sequences=True,),
    tf.keras.layers.LSTM(32, return_sequences=False),
    tf.keras.layers.Dense(32),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_12 (LSTM)               (None, 5, 32)             5760
_________________________________________________________________
lstm_13 (LSTM)               (None, 32)                8320
_________________________________________________________________
dense_18 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_19 (Dense)             (None, 5)                 165
_________________________________________________________________
reshape_11 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 15,301
Trainable params: 15,301
Non-trainable params: 0
_________________________________________________________________
```

Single-layer GRU Zinc

```
multi_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(16, return_sequences=False),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_7 (GRU)                  (None, 16)                1440
_________________________________________________________________
dense_23 (Dense)             (None, 5)                 85
_________________________________________________________________
reshape_14 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 1,525
Trainable params: 1,525
Non-trainable params: 0
_________________________________________________________________
```

Multi-layer GRU Zinc

```
multi_layer_gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.GRU(32, return_sequences=False),
    tf.keras.layers.Dense(32),
    tf.keras.layers.Dense(OUT_STEPS*num_features,
                          kernel_initializer=tf.initializers.zeros()
                          ),
    tf.keras.layers.Reshape([OUT_STEPS, num_features])
])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru_5 (GRU)                  (None, 5, 32)             4416
_________________________________________________________________
gru_6 (GRU)                  (None, 32)                6336
_________________________________________________________________
dense_21 (Dense)             (None, 32)                1056
_________________________________________________________________
dense_22 (Dense)             (None, 5)                 165
_________________________________________________________________
reshape_13 (Reshape)         (None, 5, 1)              0
=================================================================
Total params: 11,973
Trainable params: 11,973
Non-trainable params: 0
_________________________________________________________________
```

Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway