UNIVERSITY OF AMSTERDAM

MSc MATHEMATICS

MASTER THESIS

Application of Deep Learning for Simulating Time-Series of Oil Futures

---

Author:
Sebastian Ruiz

Supervisor:
prof. dr. P. Spreij
C. Schmidt

Examination date:
December 13, 2019

Korteweg-de Vries Institute for
Mathematics

Abstract

The aim of this master's thesis is to simulate oil futures products using deep learning methods rather than classical models. The master's thesis was written under the supervision of the University of Amsterdam and in co-operation with a large Dutch bank. The time-series data used for these simulations is provided by the bank. Dimension reduction methods including principal component analysis (PCA) and several autoencoders are compared in the compression of the number of tenors of futures curves from 56 to between one and four. The autoencoders analysed include variational and adversarial autoencoders. The performance of these techniques is shown to be coupled with how the data is processed, and multiple scaling techniques are examined. Various conditional generative adversarial networks (GANs) are used to make simulations of encoded log-return time-series of futures curves data. The results show that these neural network models perform three times better than the benchmark two-factor mean-reverting Andersen Markov model. Furthermore, trained autoencoder models are applied to detect unrealistic market scenarios with successful results.

Title: Application of Deep Learning for Simulating Time-Series of Oil Futures

Author: Sebastian Ruiz, sebastian.ruiz@student.uva.nl, 11823534
Supervisor: prof. dr. P. Spreij, C. Schmidt
Second Examiner: dr. Erik Winands

Examination date: December 13, 2019

Korteweg-de Vries Institute for Mathematics
University of Amsterdam
Science Park 105-107, 1098 XG Amsterdam
http://kdvi.uva.nl

Contents

1 Introduction **6**

1.1 A Short History of Oil Prices 7

1.2 Previous Work 7

2 Analysis of Oil Futures **9**

2.1 Oil Products 9

2.2 Futures Curve and Futures Contracts 10

2.3 Exploring the Data 12

2.4 Covariance of Oil Futures 13

3 Classical Methods for Simulating a Time-Series of Curves **15**

3.1 Symmetric Mean Absolute Percentage Error 15

3.2 Andersen Markov Model 15

3.3 Principal Component Analysis 17

4 Introduction to Neural Networks **19**

4.1 Training, Validation and Test Sets 19

4.2 Feedforward Networks 19

4.3 Forward Propagation 20

4.4 Back-Propagation 20

4.5 Parameter Optimisation 21

4.6 Activation Functions 21

4.7 Cost Functions 22

5 Investigation into Specialised Neural Networks **24**

5.1 Autoencoders 24

5.1.1 Standard Autoencoder 24

5.1.2 Variational Autoencoder 24

5.1.3 Adversarial Autoencoders 26

5.2 Generative Adversarial Networks 27

5.2.1 Standard GAN 27

5.2.2 Conditional GAN 28

5.2.3 Wasserstein GAN 29

5.2.4 Wasserstein GAN Improved 30

5.2.5 GAN for Anomaly Detection 31

5.3 Convolutional Networks 31

5.4 Recurrent Networks 31

5.4.1 Long Short-Term Memory 32

5.4.2 Gated Recurrent Unit 33

6 Data Preparation 34
6.1 Preprocessing Methods . . . 34
6.1.1 Standardisation . . . 34
6.1.2 Normalisation . . . 35
6.1.3 Log-returns . . . 35
6.2 Filtering Autoencoder Output . . . 35
6.3 Application to Time-Series of Futures Curves . . . 36
6.3.1 Normalisation over the Tenors . . . 37
6.3.2 Standardisation over the Tenors . . . 37
6.3.3 Log-returns over the Curves . . . 38
6.3.4 Normalisation and Standardisation over the Curves . . . 38

7 Dimension Reduction of Time-Series of Futures Curves 40
7.1 Normalisation over the Tenors . . . 41
7.1.1 Results . . . 44
7.2 Standardisation over the Tenors . . . 44
7.2.1 Results . . . 47
7.3 Log-Returns over the Tenors . . . 47
7.3.1 Results . . . 50
7.4 Dimension Reduction over Windows of Futures Curves . . . 50
7.5 Semi-Invertible Preprocessing Methods . . . 50
7.6 Conclusion . . . 51

8 Generative Adversarial Networks for Time-Series Simulation 52
8.1 Spot Price Time-Series Simulation . . . 52
8.2 Time-Series of Futures Curves Simulation . . . 53

9 Time-Series Simulation with RNNs 57

10 Simulation of Time Series Data with Missing Values 58

11 Detecting Unrealistic Scenarios 60

12 Conclusion and Remarks 63

A Mathematical Concepts 64
A.1 Notation . . . 64
A.2 Information Theory . . . 64
A.3 Bayesian Statistics . . . 65
A.4 Maximum Likelihood Estimation . . . 65
A.5 Variational Inference . . . 65
A.6 Reparametrisation Trick . . . 66

B Andersen Markov Model **68**

C Code **71**

Popular Summary **72**

1 Introduction

The development of new time-series simulations is an essential part of risk management. Neural networks offer the possibility to produce expertly tuned models at a much lower price. Instead of creating a stochastic model that encapsulates all the properties of the time-series data, a neural network is capable of learning these facts simply by training on historical data.

In financial risk management, the simulation of market scenarios can be used to price assets and to estimate the exposure of one's portfolio. In the simulation of market scenarios, the statistical properties of past observed market behaviour are reproduced. These properties, referred to as stylised facts, can include: the short end of an interest rate curve is more volatile than its long end, volatility smile is more pronounced for options with a shorter time to expiry, and futures curves have a certain amount of smoothness. Realistic market scenarios would also have to conform to no-arbitrage constraints. Typical simulations of realistic futures market evolutions are done using models calibrated to historical data. The model is constructed such that it displays the desired properties. There are many drawbacks with this approach: the stylised facts need to be known, important features can be missing if they are hard to detect, and the model may not be able to produce certain, possibly likely market configurations. Such a model should be frequently checked against new data to determine whether the model still reflects the properties of the data.

A new possibility to generate futures market scenarios is to use neural networks that can be trained to output realistic-looking scenarios. Neural networks can learn features existing in historical data and can be retrained on the most recent historical data without having to recreate the model to factor in new features. A neural network could capture the stylised facts of the data without them being directly incorporated in the model, thereby overcoming some of the difficulties of standard models. The primary focus of the project is to create a simulation engine for futures curves that produces realistic market scenarios. The secondary focus is to consider how to account for cases where some data points, or the entire underlying product except for some data points, are missing. In complex systems, historical prices of various financial products are aggregated from many sources, and it is often the case that there are pricing discrepancies between sources and incorrect prices in the data. The historical prices may be used in many models, and as such the correctness of the data is imperative. One of the fundamental properties of neural networks is that they are very capable in capturing important attributes of a dataset. A neural network trained on realistic data can be used to spot unrealistic patterns in historical data that it has not been trained on. This is useful to detect errors in given data and to test the performance of a model capable of simulating market scenarios. Automating this process would help an otherwise labour-intensive task.

Before formally introducing neural network models, an analysis is made about the history of oil prices in Section 1.1 along with previous work in this field in Section 1.2. In Section 2 oil products and futures contracts are examined, and data exploration is carried out. In Section 3 the SMAPE metric to measure the performance of simulations is inspected, the benchmark stochastic two-factor Andersen Markov model is introduced, and principal component analysis is explored for dimension reduction. Neural network concepts are introduced in Section 4, focussing on the configuration of training, validation and test sets, and the fundamental mathematics of a simple feedforward neural network. The concepts are built upon in Section 5 where specialised types of neural networks are investigated. The networks considered include autoencoders, generative adversarial networks, and recurrent networks. For each of these, a multitude of variations is explored. These are critical for dimension reduction of time-series and for simulating time-series. Necessary data preparation techniques are discussed in Section 6, these are required to optimise the performance of the neural networks. The main techniques studied concern standardisation, normalisation, and taking log-returns. Dimension reduction of time-series of futures is investigated in Section 7. Data preparation techniques are combined with autoencoders from Section 5 and principal component analysis (PCA) from Section 3.3 in order to achieve this. In Section 8, generative adversarial networks are used in combination with theory from Sections 5 and 7 to simulate time-series of futures curves. Experimentation using recurrent neural networks (RNNs) to try to simulate time-series data is done in Section 9. In Section 10 time-series data with missing values is examined, and a method based on the work in Section 5.2 is used to fill in the missing data. The usefulness in this model comes from that the model can also be trained on data containing missing values. Using autoencoder models, methods to detect unrealistic parts of a time-series are studied in Section 11. The final conclusion and remarks for future research are made in Section 12. In Appendix A mathematical concepts that are used in Section 4 and in Section 5 are

explained. The technical details for how the Andersen Markov model in Section 3.2 is calibrated, is given in Appendix B.

1.1 A Short History of Oil Prices

Oil prices are highly volatile. The reasons behind this are a combination of war, politics, limited availability and it being a highly sought after resource. The demand and supply are linked strongly to the political and economic behaviour of unstable countries and is correlated with events such as military conflicts and natural disasters [Barunik et al. 2015].

At the beginning of the 20th century, oil emerged as the preferred energy source over burning coal. This shift was lead by petroleum being more flexible than coal and its key use in vehicles, from aeroplanes to ships, to cars. The US was the dominant player in the market and the price of oil was fixed at the Gulf of Mexico. In the 1950s oil-producing nations, particularly in the middle east and in South America saw oil companies operating in their countries and began to exercise authority and control over their countries' oil resources. In 1960 the governments of Venezuela, Saudi Arabia, Kuwait, Iraq, and Iran founded the Organisation of the Petroleum Exporting Countries (OPEC) for negotiating with oil companies on oil production and prices. Saudi Arabia has the majority of OPEC reserves, followed by Iran and Venezuela [Interactive 2019].

During 2000-2008 the price of oil rose significantly from \$20 to \$140 per barrel and was explained by the rise in demand by countries such as China and India [Mouawad 2007]. The price rise could also be attributed to the US invasion of Iraq [Wiemer 2015]. During the financial crisis in 2008, the price dropped from \$140 to \$30, rebounding back to \$80 per barrel. Until 2014 the price stayed in the range of \$90 to \$120 per barrel, this could be attributed to the OPEC that acts as a cartel, fixing the price of oil and implementing quotas. In 2014 the price started declining due to an increase in production and a decline in demand. As well as this, OPEC countries exceeded their quotas and caused an oversupply driving down prices. Countries such as Saudi Arabia and Russia depend on their oil exports, causing them to increase production and further decrease prices. By 2016 the price was at \$30 per barrel. Between 2016 and 2018 the price of oil rose dramatically to \$70 per barrel due to OPEC quotas.

In the last 20 years, unconventional methods for oil and gas production have sprung up around the world in the forms of horizontal drilling, offshore drilling, and hydraulic fracturing. These techniques have created a low-cost alternative to compete against OPEC and have turned countries such as the US from an oil and gas importer to exporter.

In order to forecast futures oil prices, it is necessary to consider political and economic behaviour around the world, however modelling such aspects are out of the scope of this project. The aim of this paper is to simulate possible scenarios of futures oil prices based solely on historical data.

1.2 Previous Work

The work of Ahmed et al. [2010] presents a comparison study for some of the major machine learning models for time-series forecasting. The methods are tested on the monthly M3 time-series competition data [Forecasters 2000] consisting of around 1000 time-series. The neural networks considered are the multilayer perceptron (MLP) (explained in Section 4.2), the Bayesian neural network (BNN), the Radial Basis Function Neural Network (RBF), and the Generalised Regression Neural Network (GRNN). Further models considered were the K Nearest Neighbour Regression (KNN), Classification and Regression Trees (CART), Support Vector Regression (SVR), and Gaussian Processes (GP). The most computationally demanding models were BNN and MLP, however in most cases, the extra computation time required was not an issue. They conclude that the best models were MLP and GP. The MLP yielded good results because it can be reduced to a linear model and this is in agreement with the results from the M3 competition where simple models tend to outperform complex models.

In Fu et al. [2019] the conditional GAN (see Section 5.2.2) is examined for its ability to simulate time-series. Data is simulated according to the vector autoregressive model (VAR) and it is found that the CGAN is able to learn the dependent structure of the VAR time-series and the heavy tails of the underlying noise. In another analysis, 1-day stock returns are examined where the data is split into stressed and normal periods, where the stressed period concerns the time of the 2008 financial crisis. It is found that the CGAN is able to outperform the historical simulation method for the calculation of value-at-risk (VaR).

In Zhou et al. [2018] Long Short-Term Memory (LSTM) (see Section 5.4.1) and convolutional neural networks (CNN) (see Section 5.3) are used as the generator and discriminator network respectively in a GAN (see Section 5.2.1) to forecast the high-frequency stock market. The model created is called GAN-FD and it uses 13 technical indexers as input data and it forecasts the next day price. The intuition given behind this model is that a trader usually predicts the stock price through the available indicator data, which is what the generative model G does, and he judges the correctness of his own forecast using the previous stock price, which is what the discriminative model D does.

Uber has used LSTM neural networks for time-series forecasting [Laptev et al. 2017], specifically for extreme event simulation. LSTMs were chosen because they can model complex interactions in the data. They use the SMAPE metric to measure the model error. The same metric is applied to measure the performance of the models in this paper (see Definition 3.1).

Facebook has created a forecasting tool [Taylor et al. 2017] that is easy to use for analysts. It is a regression model with trend, seasonality, and holidays components. The model can easily accommodate new seasonality components. The model has the advantage over models such as ARIMA in that it does not require regularly spaced data measurements. Furthermore, fitting is fast and the model has easily interpretable parameters.

Machine learning techniques have been extremely successful at solving difficult real-world problems in the realms of autonomous vehicles, intelligent robots, image recognition, speech recognition, and language translations. Seeing these successes, machine learning techniques are attempted to be used for every problem where instead traditional models may perform better. In Makridakis et al. [2018] concerns with using machine learning methods for forecasting are examined. They note that in the literature, many published studies do not satisfactorily compare the machine learning model with simple statistical methods and that the data used in a study is in many cases not publicly available making it difficult to replicate the results. Furthermore, they remark that machine learning methods should be capable of specifying the uncertainty around them or provide confidence intervals. In Makridakis et al. [2018, Section 4] it is shown that statistical methods such as ARIMA and Theta obtain better forecasting performance compared to many popular machine learning methods such as GP, RNN, KNN, LSTM, and CART.

To conclude, it is important to compare neural network time-series simulation methods with a benchmark model. This is in order to show whether the examined neural network techniques give reasonable results, and possibly make improvements in the field of time-series simulation.

2 Analysis of Oil Futures

In this section, the historical oil and gas futures time-series data available is examined. The definition of a futures contract is given, it is shown how this relates to a futures curve, and what the standard properties of a futures curve are.

2.1 Oil Products

Crude oil extracted from the ground in its unrefined state varies in density, consistency and sulphur content. Density ranges from light to heavy, while sulphur content is characterised as sweet or sour. API gravity is a measure of how heavy or light oil is compared to water and it is an inverse measure to its density. The API gravity of water is $19°$. Crude oil is classified as light, medium, or heavy according to its measured API gravity. In general, the higher the API gravity, the more valuable the crude oil. The classification of crude oil is given by:

- Light crude oil has an API gravity higher than $31.1°$ and it has short hydrocarbon chains.
- Medium oil has an API gravity between $22.3°$ and $31.1°$.
- Heavy crude oil has an API gravity below $22.3°$ and it has long hydrocarbon chains.

Crude is considered sweet if it is low in sulphur content ($< 0.5\%$), or sour if high ($> 1.0\%$). Crude oils that are light and sweet are usually priced higher than heavy, sour crude oils because they can be processed with less sophisticated and less energy-intensive refineries, and they can easily and cheaply be used to make petrol. In Fig. 2.1 a graph shows the relation between sweet/sour and heavy/light oils and where they are extracted.

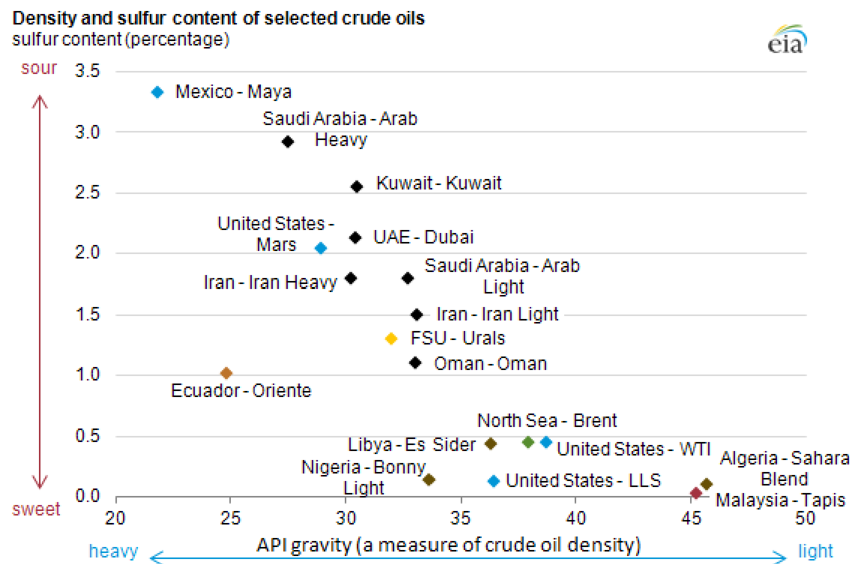


Figure 2.1: Source: Administration [2012]. Crude oils, sweet and sour vs. heavy and light. WTI = West Texas Intermediate; LLS = Louisiana Light Sweet; FSU = Former Soviet Union; UAE = United Arab Emirates.

A benchmark crude oil is a crude oil that acts as a reference price for buyers and sellers of crude oil. They are used because there are many varieties and grades of crude oil, so benchmarks allow for easy referencing for buyers and sellers. The main benchmark crude oils are

- West Texas Intermediate (WTI), United States, Oklahoma
- Brent Blend, United Kingdom, North Sea
- Dubai Crude, United Arab Emirates, Dubai.

The oil commodities for which futures time-series data is available are given in Table 2.1. The products consist of crude oils Brent and WTI, and a selection of refined oils.

Product	Description	API Gravity	Sulphur Content
Brent ICE	North Sea oil traded on the Intercontinental Exchange (ICE)	38.3°	0.37%
Brent Dated	Physical cargoes of crude oil in the North Sea that have been assigned specific delivery dates.	38.3°	0.37%
WTI NYMEX	West Texas Intermediate, listed on the New York Mercantile Exchange (NYMEX)	39.6°	0.24%
F35 ROT FB	Fuel Oil FOB Barges Rotterdam	-	3.5%
GO01 ROT FB	Gas oil FOB Barges Rotterdam	-	0.1%
HSFO380 SGP FC	Fuel Oil FOB Cargoes Singapore	-	3.8%
JK ROT FB	Jet Fuel FOB Rotterdam	$\approx 42°$	0.37%

Table 2.1: A list of the oil commodity data used in this paper. FOB stands for *free on board* and specifies at which points the costs involved with the delivery of the goods shift from the seller to the buyer.

2.2 Futures Curve and Futures Contracts

Futures contracts (Definition 2.1) are negotiated at futures exchanges that serve as a marketplace between buyers and sellers. They are standardised contracts and can only be traded on a futures exchange.

Definition 2.1 (Futures Contract)**.** *A futures contract is a standardised forward contract between two parties to buy or sell an asset at a specified date in the future at a price agreed upon today.*

The buyer of the contract is called the long position holder, while the seller is called the short position holder. The Futures exchange requires both parties to put up initial cash, known as the margin, usually a percentage of the value of the futures contract. The margin must be maintained throughout the life of the contract. The product is marked to market daily, which means that the difference between the agreed upon price and the daily futures price is re-evaluated daily. The exchange will take money out of the losing party's margin account and put it in the other party's account. On the delivery date, only the spot value is exchanged.

The first futures market appeared in the 17th and 18th centuries in Holland [Goetzmann et al. 2008] and it helped lay the foundations of the modern financial system. The original use for futures contracts was to mitigate the risk of price movements by allowing parties to fix future prices today. This allows a party to be protected against unfavourable price movements for a product that they require in the future.

The **maturity** is the date of expiry of a contract, and a **tenor** is the duration left until expiry of a contract. The price of a futures contract at time t with tenor T is written as the function $F(t, T)$. For simplification of the Andersen Markov Model (Section 3.2), $\hat{F}(t, t+T) := F(t, T)$ is defined, where the $t + T$ can be treated as a date. In Table 2.2 there is an example of a historical futures contract. For the first entry, the historical date is $t =$ 2-1-2018 and tenor $T = 29$, the duration in days between the t and the expiry date 31-1-2018. From a list of historical futures contracts such as in Table 2.2 a **futures curve** as defined in Definition 2.2 can be created.

Definition 2.2 (Futures curve)**.** *Fix a historical date t. Consider a sequence of tenors $(T_1, ..., T_n) \in \mathbf{N}^n$ given in days. Usually $T_1 = 0$ and $T_{i+1} - T_i = 30$, representing 30 day intervals. The futures curve at time t is given by $(F(t, T_j))_{j=1,...,n}$. If there is no futures contract giving a value for $F(t, T_j)$, then $F(t, T_j)$ is estimated by the linear interpolation of nearest existing tenors $R, S \in \mathbf{N}$ such that $R < T_j < S$.*

A futures curve shows the current price for the underlying to be delivered at maturity. It is not a forecast of futures spot prices, but the reflection of the view of the market at that time. Futures curves have typical forms that they take with a few examples given in Fig. 2.2. A downward sloping curve, where the forward price is lower than the spot price is called **backwardation**. An upward sloping curve, where the forward price is higher than the spot price is called **contango**. Usually, futures

Commodity Name	Historic Date	Price	Contract Name	Expiry
CO1 Comdty	2-1-2018	66.57	COH8	31-1-2018
CO1 Comdty	3-1-2018	67.84	COH8	31-1-2018
CO1 Comdty	4-1-2018	68.07	COH8	31-1-2018
⋮	⋮	⋮	⋮	⋮
CO1 Comdty	29-1-2018	69.46	COH8	31-1-2018
CO1 Comdty	30-1-2018	69.02	COH8	31-1-2018
CO1 Comdty	31-1-2018	69.05	COH8	31-1-2018

Table 2.2: Some prices of a futures contract for Brent Crude (ICE) with commodity name 'CO1 Comdty' and contract name 'COH8' in Bloomberg.

curves are simply in contango or backwardation like in the 2014-08-01 and 2015-10-07 examples. Due to some uncertainty in the prices for a small time frame, some futures curves e.g. 2016-12-02 and 2017-04-21 examples have a local maximum.

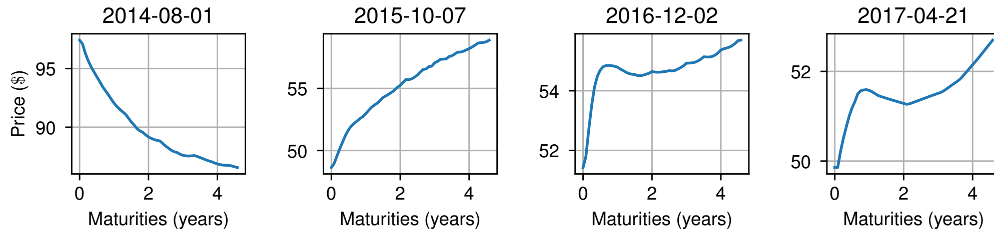
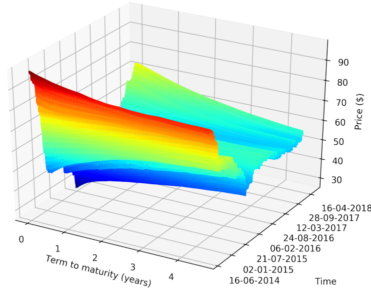


Figure 2.2: Types of futures curves. In this figure, a selection of different shaped futures curves from historical data are shown. The curves come from the WTI NYMEX dataset.

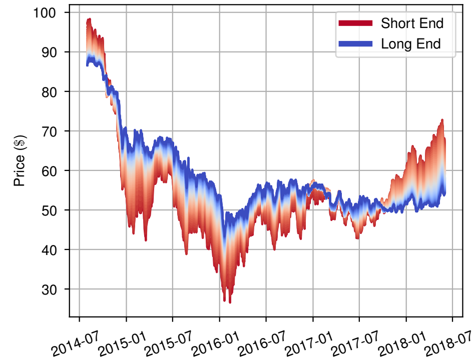
Consider a time-series given by a sequence of dates $(t_1, ..., t_m)$, with futures curves as given in Definition 2.3.

Definition 2.3 (Time-series of futures curves)**.** *A time-series of futures curves is given by an* $m \times n$ *matrix* X *with the columns representing tenors* $(T_1, ..., T_n)$ *and the rows representing dates* $(t_1, ..., t_m)$*. Then each row* i *is given by the futures curve at time* t_i*. In other words,* $X_{ij} = F(t_i, T_j)$ *or the linear interpolation of the nearest existing tenors as in Definition 2.2.*

It is possible to plot a time-series of futures curves as a 3-dimensional graph as shown in Fig. 2.3a. Following Definition 2.3 the Maturities axis refers to the tenors T_j, and the Time axis refers to times t_i. In Fig. 2.3b the same 3d plot is shown but represented in 2d. The **short end** is the time-series with maturity $T_1 = 0$. The **long end** is the time-series with maturity $T_{56} = 56 \cdot 30 = 1680$ days, approximately 4.5 years. The futures time-series behaves fairly smoothly over the tenors as seen in Fig. 2.2, while day-to-day prices behave irregularly, as seen in Fig. 2.3b.



(a) 3d plot of WTI NYMEX Futures time-series.



(b) 2d plot of WTI NYMEX Futures time-series.

Figure 2.3: Time series of WTI NYMEX (North American oil benchmark) futures curves from 01/08/14 to 01/06/18.

2.3 Exploring the Data

An exploration of the datasets is made, outline which datasets will be used for training and which for testing. Furthermore, an explanation of some of the limitations imposed by the data is made.

The data available are futures time-series for the oil commodities given in Table 2.1. Refer to each futures time-series of a commodity as a dataset. The available data runs from 2014-08-01 to 2018-06-01, that is 988 days of data - the markets are open on working days only. Each futures curve consists of 56 tenors; therefore, a dataset can be represented as a matrix of size 988×56, sometimes this is notated as $(988, 56)$. In Table 2.3 a list of the products with minimum and maximum prices in U.S. dollars ($) per barrel is shown. The range of each dataset varies considerably. The data for *F35 ROT FB*, *GO01 ROT FB*, *HSFO380 SGP FC*, *JK ROT FB* falls in the range $[100\$, 1000\$]$ while the data for *Brent ICE*, *Brent Dated*, and *WTI NYMEX* falls in the range $[20\$, 110\$]$.

Product	Minimum Price ($)	Maximum Price ($)
Brent ICE	27.76	107.02
Brent Dated	27.21	79.89
WTI NYMEX	26.53	98.36
F35 ROT FB	98.94	569.27
GO01 ROT FB	237.87	896.92
HSFO380 SGP FC	122.83	591.67
JK ROT FB	263.14	970.37

Table 2.3: Minimum and maximum prices of the commodities from the available data in the period 2014-08-01 to 2018-06-01.

At the beginning of the project, the choice was made to choose one product as the test set, one product as the validation set, and the others as the training sets. *WTI NYMEX* was assigned as the test set, *Brent ICE* was assigned as the validation set, and the other 5 sets as training sets. The choice of this split was mainly due to *WTI NYMEX* being one of the largest benchmark crude oils, and thus a good choice to have as the test set.

It is not possible to treat all the training datasets as a single dataset by joining the time-series' at the beginning and ends. This will create price jumps where the datasets are joined. Furthermore, by joining the training sets into a single large time-series is that the price range of the set is approximately $[20\$, 1000\$]$. This is larger than the price range of any individual dataset. Scaling of the data is carried out as outlined in Section 6, so that the data can be used by a neural network. This will scale the data to a smaller range such as to the interval $[0, 1]$ in the case of normalisation. In doing so, some datasets will be scaled to the range $[0, 0.1]$, while others to the range $[0.1, 1]$. The discrepancy between the intervals $[20\$, 110\$]$ and $[100\$, 1000\$]$ is hereby maintained. This is a problem when

training neural networks because it is needed that each futures curve is approximately in the same range. This method is therefore discarded and another approach is taken.

Each training set is seen separately, and scaling is carried out per dataset. That means that when scaling is applied, each dataset is scaled to fit in the interval $[0, 1]$ or similar with other scaling techniques. The problem with this approach is that prices between datasets are in this method no longer preserved and futures curves from different training sets may be scaled to the same scaled curve. Testing this method produced much better results than the first, so although this method also has its disadvantages, it does allow for the successful training of neural networks. This is the method that is used throughout this paper.

As later discussed in Section 12 other training/validation/test distributions (see Section 4.1) could have been considered. The first $3/4$ of each dataset could have been treated as training data, the next $1/8$ as validation data and the last $1/8$ as test data. This would have been useful to learn interrelationships between the different products. This method was not chosen initially because it would have resulted in less training data when training autoencoders to learn to compress futures curves (see Section 7).

2.4 Covariance of Oil Futures

The covariance of the log-returns over the tenors is often calculated for historical oil futures time-series data. The covariance of the historical data is compared against the covariance of the simulated data provided by the models. The definition of covariance is as follows.

Definition 2.4 (Covariance)**.** *For random variables X, Y over $\mathbb{R}$,*

$$\text{cov}(X, Y) = \mathbf{E}\left[(X - \mathbf{E}[X])(Y - \mathbf{E}[Y])\right].$$

For n realisations $\mathbf{x} = (x_1, ..., x_n)$ and $\mathbf{y} = (y_1, ..., y_n)$ of X and Y, the sample covariance is given by

$$\text{cov}(\mathbf{x}, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}),$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.

Consider a random vector $\mathbf{X} = (\mathbf{X}_1, ..., \mathbf{X}_m)$ in $\mathbb{R}^m$. The covariance is defined as

$$\text{Cov}(\mathbf{X}) = \begin{pmatrix} \text{cov}(\mathbf{X}_1, \mathbf{X}_1) & \dots & \text{cov}(\mathbf{X}_1, \mathbf{X}_m) \\ \text{cov}(\mathbf{X}_2, \mathbf{X}_1) & \dots & \text{cov}(\mathbf{X}_2, \mathbf{X}_m) \\ \vdots & & \vdots \\ \text{cov}(\mathbf{X}_m, \mathbf{X}_1) & \dots & \text{cov}(\mathbf{X}_m, \mathbf{X}_m) \end{pmatrix}.$$

Let $\mathbf{x} \in \mathbb{R}^{m \times n}$ matrix, where the n columns of $\mathbf{x}$ represent n realisations of the random vector $\mathbf{Z}$ over $\mathbb{R}^m$. Let the sample mean be given by $\bar{\mathbf{x}} = (\bar{\mathbf{x}}_1, ..., \bar{\mathbf{x}}_m)^\top$ where $\bar{\mathbf{x}}_i = \frac{1}{n}\sum_{j=1}^{n}\mathbf{x}_{ij}$. The covariance is then given by

$$\text{Cov}(\mathbf{x}) = \frac{1}{n}(\mathbf{x} - \bar{\mathbf{x}}1_{(n)}^\top)(\mathbf{x} - \bar{\mathbf{x}}1_{(n)}^\top)^\top$$

where $\mathbf{1}_{(n)} = (1, ..., 1)^\top$ a column vector of n 1's. Note $\text{Cov}(\mathbf{x})$ is a $(m \times m)$ matrix. Let $k, l \in \{1, ..., m\}$. Then

$$\begin{aligned} \text{Cov}(\mathbf{x})_{kl} &= \frac{1}{n}\sum_{p=1}^{n}(\mathbf{x} - \bar{\mathbf{x}}1_{(n)}^\top)_{kp}((\mathbf{x} - \bar{\mathbf{x}}1_{(n)}^\top)^\top)_{pl} \\ &= \frac{1}{n}\sum_{p=1}^{n}(\mathbf{x}_{kp} - \bar{\mathbf{x}}_k)(\mathbf{x}_{lp} - \bar{\mathbf{x}}_l). \end{aligned}$$

Let $\mathbf{x}_{k\cdot} = (\mathbf{x}_{k1}, ..., \mathbf{x}_{kn})$ and $\mathbf{x}_{l\cdot} = (\mathbf{x}_{l1}, ..., \mathbf{x}_{ln})$. It follows that $\text{Cov}(\mathbf{x})_{kl} = \text{cov}(\mathbf{x}_{k\cdot}, \mathbf{x}_{l\cdot})$.

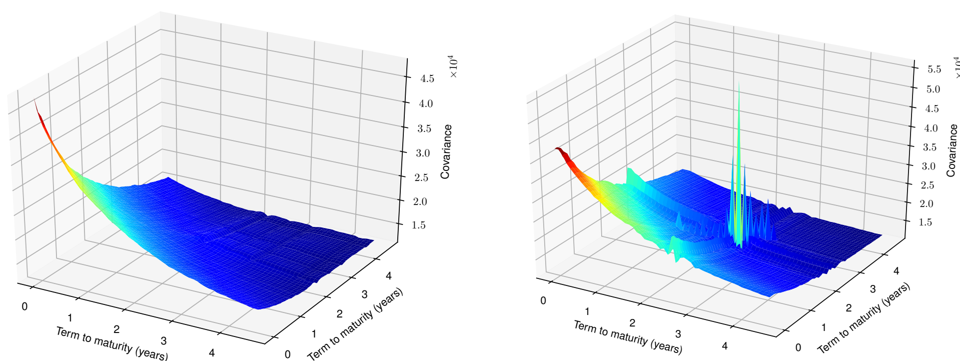
Definition 2.5 (Log-returns)**.** *Let* $\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$*. Consider constants* $c, k \in \mathbb{R}$*. Let* $\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_n) = (x_1 + c, x_2 + c, ..., x_n + c)$*, where* c *is a constant such that* $x_i + c > 0$ *for all* $i = 1, ..., n$*. Define*

$$y_{i-1} := k \log\left(\frac{x_i + c}{x_{i-1} + c}\right) = k \log\left(\frac{\tilde{x}_i}{\tilde{x}_{i-1}}\right) \quad \text{for } i = 2, ..., n.$$

Here $\mathbf{y} := (y_1, ..., y_{n-1})$ *are the corresponding log-returns. The constant* c *is chosen to avoid division by* 0*. The default choice of the constant* k *is* 1*, however it can be used to adjust the range of the* y_i *values, which may be useful when using log-returns as input data for a neural network.*

The log-returns operation can be undone as shown in Section 6.1.3.

Let X be the $m \times n$ matrix of WTI NYMEX futures time-series with $n = 56$ tenors and $m = 988$ days. Let Y be the $((m-1) \times n)$ matrix given by taking *log-returns over the tenors* of X (as explained more clearly in Section 6.3). The covariance matrix $\text{Cov}(Y^\top)$ is an $(n \times n)$ matrix and it is given as a surface plot in Fig. 2.4.



(a) Covariance of log-returns of WTI NYMEX. **(b)** Covariance of log-returns of BR DATED.

Figure 2.4: Covariance of log-returns for some of the datasets from Table 2.1.

In Fig. 2.4a along the diagonal a downward sloping curve can be seen, showing that the variance decreases as the tenor increases. This agrees with that the short end of the curve is more volatile than the long end. When making simulations of the data it is expected that the covariance of the log-returns should be similar in shape to this. Unfortunately not all the data has similar looking covariance log-returns. In Fig. 2.4 the covariance of log-returns from 4 datasets of Table 2.1 are examined. While the covariance of log-returns in Fig. 2.4b is approximately decreasing similar to Fig. 2.4a, it contains some spikes around the 2.5-year mark, showing a high variance of the log-returns at this point in time. The reason behind this high variance is not clear and it could be due to some errors in the datasets.

3 Classical Methods for Simulating a Time-Series of Curves

In this section, classical methods for commodity time-series simulation are examined. In Section 3.1 a measure is introduced to compare the results of different simulation methods. In Section 3.2 a stochastic model is examined, which will act as the benchmark model and is used to compare generative neural network models (see Section 8) against a baseline. In Section 3.3 a standard method for dimension reduction is considered and this is used to set a baseline for the autoencoder models (see Section 7).

3.1 Symmetric Mean Absolute Percentage Error

In order to determine the quality of a particular simulation, the **symmetric mean absolute percentage error** (SMAPE) which determines the relative error on a sequence of values is introduced. In the literature SMAPE is often used to measure the accuracy of time-series forecasting [Laptev et al. 2017; Ahmed et al. 2010]. SMAPE is less sensitive to outliers compared to **root mean square error** (RMSE) [Chen et al. 2017]. SMAPE is scale-independent, so the scale is independent of the scale of the data, while RMSE and MSE are scale-dependent. The SMAPE function is given in Definition 3.1.

Definition 3.1 (SMAPE)**.** *Consider the sequences* $A = (A_1, ..., A_n) \in \mathbb{R}^n$ *and* $B = (B_1, ..., B_n) \in \mathbb{R}^n$*. The symmetric mean absolute percentage error (SMAPE) is given by*

$$\mathrm{SMAPE}(A, B) := \frac{1}{n} \sum_{t=1}^{n} \frac{|A_t - B_t|}{(|A_t| + |B_t|)/2},$$

where it is assumed that for all t, $|A_t|+|B_t| > 0$*. Then the SMAPE lies in the range* $\mathrm{SMAPE}(A, B) \in (0, 2)$*.*

The SMAPE is not a metric since sub-additivity does not hold. The MSE is also not a metric for the same reason. In the literature, they are called metrics to mean performance indicators.

3.2 Andersen Markov Model

The Andersen Markov model (AMM) is a two-factor mean-reverting model from Andersen [2008, Section 7.1]. The performance to that of the generative adversarial neural networks is compared in Section 8. The model is calibrated by choosing the parameters such that the model covariance log-returns matches that of historical data. The AMM model is referred to as the benchmark model.

Let W_1, W_2 be independent Brownian motions under the risk-neutral measure Q. The forward prices are determined by the following SDE

$$\frac{d\hat{F}(t,T)}{\hat{F}(t,T)} = \mu(t,T)dt + \sigma_1(t,T)dW_1(t) + \sigma_2(t,T)dW_2(t), \tag{1}$$

where

$$\sigma_1(t,T) = e^{a(T)}\eta_1 e^{-\kappa(T-t)} + \eta_\infty e^{a(T)}, \quad \sigma_2(t,T) = e^{a(T)}\eta_2 e^{-\kappa(T-t)}.$$

with parameters $\kappa \geq 0$ the mean reversion speed, volatility parameters η_1 and η_2, long term volatility parameter η_∞, and deterministic seasonality adjustment $a(T)$. The σ_2 affects the short-end of the futures curve while σ_1 also has the extra term $\eta_\infty e^{a(T)}$ without a decay term that persists for long futures maturities. The $\mu(t,T)$ is the deterministic time-dependent drift term. In the notation of the forward price $\hat{F}(t,T)$, both the t and T are dates referring to the futures contract at time t with maturity date T.

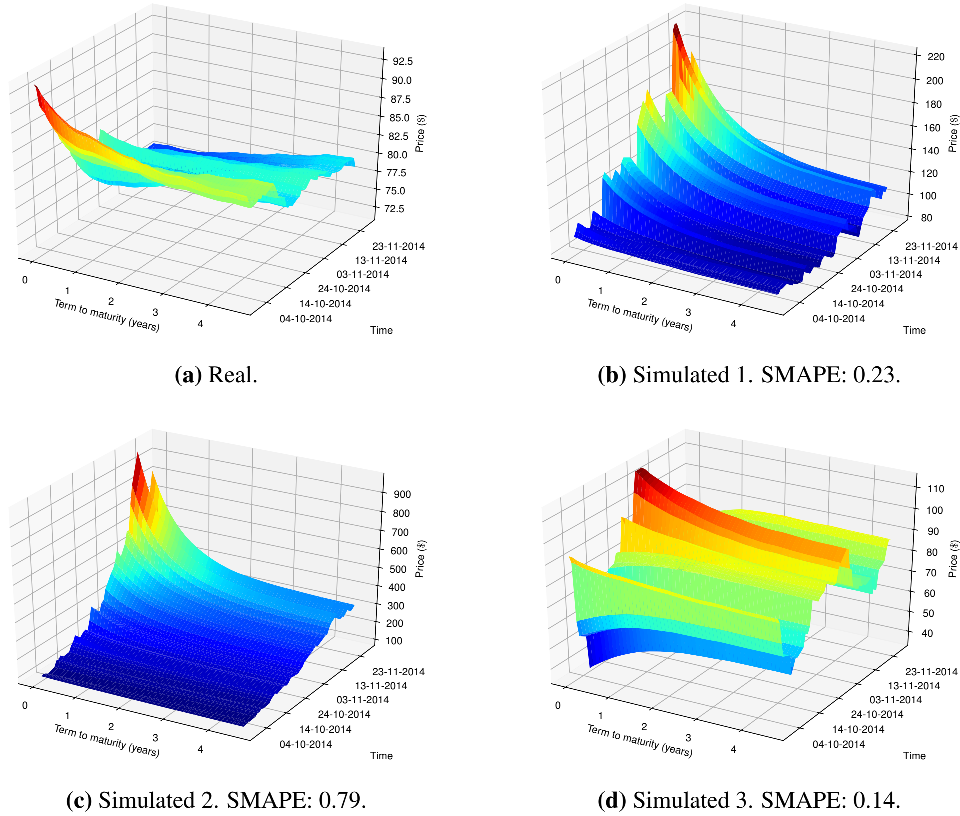
The two-factors in the model allows for modelling of both contango and backwardation, and the up and down movement of the curves as a whole. The model is calibrated by fitting the covariance of the log-returns of time-series futures data. With a two-factor model it is not possible to fit more than the first two moments. A constraint hereby is that the curves must be strictly increasing or strictly decreasing, which is not always the case in the data. The attractiveness of the Andersen Markov model is that it can reasonably model simple curves without too much complexity.

Furthermore, the AMM has support for seasonality, which can be useful to simulate commodities such as natural gases that have a seasonality component. The analysis on historical data of crude oils

by Wiemer [2015], shows that there is no seasonality component in this data, therefore the seasonality component $a(T)$ is set to zero. There is also an extension of the AMM with support for stochastic volatility which adds to its popularity in the industry. These features are mentioned because they add to the attractiveness of the model in the banking world, however in this paper the focus is put on the base model.

The parameters $\kappa, \eta_1, \eta_2, \eta_\infty$ of the model are calibrated by minimising the distance between the covariance of the log-returns of historical data to the covariance of the log-returns given by the model. Once the model is calibrated, discrete simulation of the time-series of futures curves is carried out, given the futures curve at $t = 0$. The technical details for the calibration of the model and further explanation can be found in Appendix B.

In Fig. 3.1 simulations of the model are carried out. The AMM is calibrated on the first 42 curves of the test set, to simulate the next 42 curves. Calibrating on a larger number of days does not seem to affect the results. In Fig. 3.1a the true data is shown. Comparing Figs. 3.1b and 3.1d to Fig. 3.1a it appears that the simulated time-series are more volatile. The individual curves simulated are also perfectly smooth, whereas in the real data they are not so. Repeating the simulation 100 times, a SMAPE mean of 0.26 and a standard deviation of 0.15 is found. This is shown explicitly in Table 3.1 and this is compared to the final results of the deep learning methods in Table 8.2.



(a) Real.

(b) Simulated 1. SMAPE: 0.23.

(c) Simulated 2. SMAPE: 0.79.

(d) Simulated 3. SMAPE: 0.14.

Figure 3.1: Comparison of the real data compared to 3 simulations using the Andersen Markov Model.

Method	calibrated on	SMAPE, mean	SMAPE, std
AMM	42 days	0.26	0.15
AMM	840 days	0.27	0.15

Table 3.1: Result of the AMM model from 100 simulations.

3.3 Principal Component Analysis

Principal component analysis (PCA) is a tool for data analysis that reveals the internal structure of the data by transforming the data into sets of linearly uncorrelated values called principal components. Applications of PCA are dimension reduction, data compression, feature extraction, and data visualisation. PCA is primarily used for dimension reduction in Section 7 and its performance is compared to non-linear autoencoders (Section 5.1).

Definition 3.2 (Principal Component Analysis)**.** *PCA is the Orthogonal projection of the data onto a lower-dimensional linear space, such that the variance of the projected data is maximised [Hotelling 1933]. Equivalently, it is the linear projection that minimises the average projection cost (the mean squared distance between the data points and their projections) [Pearson 1901].*

PCA can be used for dimension reduction, by omitting the axes with the smallest variances, and thereby only losing a small amount of information. PCA can also be used in predictive models, such as principal component regression (PCR). PCR is based on a standard linear regression model, but uses PCA for estimating the regression coefficients. PCR is not looked into further in this paper.

The procedure outlined in Filipovic [2009] is followed. Consider a random vector $X \in \mathbb{R}^n$. Let $\mu = \mathbf{E}[X]$ (with dimension n) and $Q = \mathrm{cov}(X)$ (with dimension $n \times n$). Since Q is symmetric and positive-definite, $Q = ALA^\top$ where $A^\top A = I$ and $L = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ for $\lambda_1 \geq \cdots \geq \lambda_n$ and $a_1, ..., a_n$ normalised eigenvectors such that $Qa_i = \lambda_i a_i$. Here $a_1, ..., a_n$ are the columns of A. Define $Y := A^\top(X - \mu)$, then $Y_i = a_i^\top(X - \mu)$ for $1 \leq i \leq n$ are the rows of Y. Call Y_i the ith principal component, and a_i the ith vector of loadings, of X. It follows that $X = \mu + AY$. Note that $\mathbf{E}[Y] = 0$ and $\mathrm{Cov}(Y) = L$, so principal components of X are uncorrelated and $\mathrm{Var}(Y_i) = \lambda_i$. Observe that

$$\sum_{i=1}^{n} \mathrm{Var}(X_i) = \mathrm{trace}(Q) = \sum_{i=1}^{n} \lambda_i = \sum_{i=1}^{n} \mathrm{Var}(Y_i).$$

Hence $\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{n} \lambda_i$ represents the variability in X from the first k principal components $Y_1, ..., Y_k$. Approximate X with $X \approx \mu + A_k Y_k$, where

$$A_k := \begin{pmatrix} | & & | \\ a_1 & \dots & a_k \\ | & & | \end{pmatrix} \quad \text{and} \quad Y_k = \begin{pmatrix} — & Y_1 & — \\ & \vdots & \\ — & Y_k & — \end{pmatrix}.$$

Using WTI NYMEX data, the variability $\sum_{i=1}^{k} \lambda_i / \sum_{i=1}^{n} \lambda_i$ for $k = 1, 2, 3$ is $0.9346, 0.9991, 0.9998$ respectively. By taking only the first three components, the dimension of the data can be reduced while still retaining most of the important information.

Let $X \in \mathbb{R}^{n \times m_1}$ be the training data with m_1 samples, and let $\hat{X} \in \mathbb{R}^{n \times m_2}$ be the test data with m_2 samples. The samples are represented as columns in the matrix. Let $k \in \mathbf{N}$ with $k < n$. Compute A_k and Y_k as above using X. Fix A_k. The dimension of the test data $\hat{X}$ is reduced by $\hat{Y} := A_k^\top(\hat{X} - \hat{\mu})$ where $\hat{\mu} = \mathbf{E}[\hat{X}]$. Now $\hat{Y}$ has dimension $k \times m_2$. Each sample has dimension reduction applied from n to k tenors. It follows that X can be reconstructed by the approximation $\hat{X} \approx \hat{\mu} + A_k \hat{Y}$. For the reconstruction, $\hat{\mu}$ can be approximated with μ, giving the approximation $\hat{X} \approx \mu + A_k \hat{Y}$.

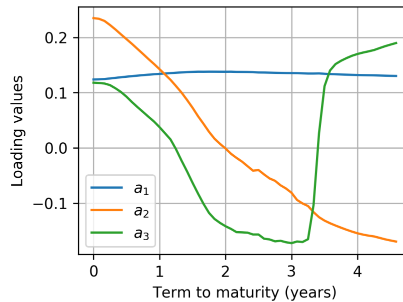


Figure 3.2: Applying PCA to WTI NYMEX, with standardisation over the tenors (see Section 7.2), the PCA loadings a_1, a_2, a_3 are obtained.

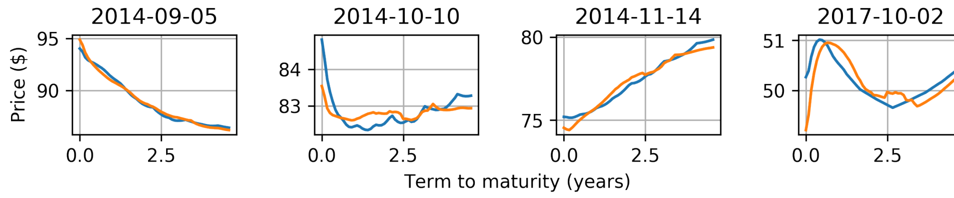


Figure 3.3: A selection of futures curves from WTI NYMEX, reconstructed using PCA with pre-processing method standardisation over the tenors (see Section 7.2).

In Section 7 the performance of PCA to various autoencoder models is compared. An example of encoding futures curves with PCA is shown in Fig. 3.3 with the associated loadings shown in Fig. 3.2. Autoencoders are non-linear and should in theory perform better than PCA, however from later results it clearly follows that this is not always the case.

4 Introduction to Neural Networks

Neural networks are a branch of machine learning. Machine learning is the ability for machines to gain their own knowledge by extracting patterns from raw data. The performance of machine learning algorithms depends on the representation of the data given and almost always the data has to be processed before being used successfully.

Machine learning still has its limitations. The **data-generating distribution** is the probability distribution that the data is assumed to be sampled from. The no free lunch theorem [Wolpert et al. 1997] states that an algorithm that performs well on a certain data-generating distribution necessarily performs badly on another distribution. Averaged over all possible data-generating distributions, every algorithm performs equally well when classifying unobserved points. Restricted to a family distributions, it is possible to design algorithms that perform well on these distributions. The hypothesis space is the set of functions that are possible for an algorithm to learn, and an algorithm is affected by how large this space is made.

A dataset can be represented as a large matrix, where each row is called a sample or an example and is a single observation. Each sample has a number of properties and these are given by the columns. These properties are often called **features**. Given a dataset containing a number of features, **unsupervised learning algorithms** learn useful properties of the structure of the dataset. For example, to learn the probability distribution that generated a dataset, or to divide the dataset into clusters. Given a dataset containing a number of features, **supervised learning algorithms** learn to associate each example with a label. The algorithm learns to predict the label based on the features.

4.1 Training, Validation and Test Sets

In supervised learning, each sample can be represented as a pair $(\mathbf{x}, \mathbf{y})$ where $\mathbf{x}$ and $\mathbf{y}$ are both a set of features. Given $\mathbf{x}$ the neural network should learn to output $\mathbf{y}$. Given a list of pairs $(\mathbf{x}, \mathbf{y})$, the data is split into training, validation and test sets. The **training set** are examples that the algorithm is trained on and used to fit the model. The **validation set** are examples that the training algorithm does not observe. It is used to estimate the **generalisation error** during or after training. The generalisation error is the error of the algorithm on previously unseen data. After seeing the generalisation error, the hyper-parameters of the model, for instance the number of layers or the number of nodes per layer, are tuned. The **test set** are examples used to estimate the generalisation error after learning has completed. No examples from this set should be present in the training or validation set. The test set is used to validate the actual real generalisation power of the algorithm. The test set should be used once all the data analysis has been done. The model **underfits** when it is obtains a large error on the training set and on the validation set. The model **overfits** when the gap between the error on the training set and the error on the validation set is large.

Some assumptions about the data are made. It is assumed that there is a data-generating process that is the true, underlying phenomenon that is creating the data. The data-generating distribution is called p_{true}. It is assumed that examples from the data are independent and identically distributed. The distribution of the available data is called p_d.

4.2 Feedforward Networks

Supervised neural networks can be summarised as, given pairs $(\mathbf{x}, \mathbf{y})$, they find a function f such that $f(\mathbf{x})$ approximates $\mathbf{y}$. In a **feedforward neural network**, also known as a **multilayer perceptron** (MLP), information flows in one direction, the value $\mathbf{x}$ is passed through the network to output a value $\hat{\mathbf{y}}$, the approximation for $\mathbf{y}$. There are no loops in the network.

Feedforward networks can be considered as a chain of functions $f^{(1)}, f^{(2)}..., f^{(l)}$ with parameters $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, ..., \boldsymbol{\theta}^{(l)}$, in the form

$$f(\mathbf{x}) = f^{(l)}(f^{(l-1)}(...f^{(1)}(\mathbf{x}, \boldsymbol{\theta}^{(1)}), ...), \boldsymbol{\theta}^{(l-1)}), \boldsymbol{\theta}^{(l)}) = \hat{\mathbf{y}}.$$

In training the parameters $\boldsymbol{\theta}$ are updated such that $\hat{\mathbf{y}}$ is as close as possible to $\mathbf{y}$ by considering a loss function $L(\mathbf{y}, \hat{\mathbf{y}})$.

4.3 Forward Propagation

The input layer is referred to as the 0 layer. The layers between the input and output layers are called **hidden layers**. The parameters $\boldsymbol{\theta}^{(k)}$ can be written as $\boldsymbol{\theta}^{(k)} = (\mathbf{W}^{(k)}, \mathbf{b}^{(k)})$ where $\mathbf{W}^{(k)}$ refer to the weights between layer $k-1$ and layer k, and where $\mathbf{b}^{(k)}$ refers to the biases of layer k. The dimension of $\mathbf{W}^{(k)}$ is $(d^{(k-1)} \times d^{(k)})$ and the dimension of $\mathbf{b}^{(k)}$ is $d^{(k)}$. The possibly non-linear functions $g^{(k)}$ retain the dimension of the input. Thus the output of each layer has dimension $d^{(k)}$. The input layer is $\mathbf{h}^{(0)} = \mathbf{x}$ of length $d^{(0)}$. The output of the first layer is given by

$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{a}^{(1)}) \quad \text{where} \quad \mathbf{a}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}.$$

The second layer is given by

$$\mathbf{h}^{(2)} = g^{(2)}(\mathbf{a}^{(2)}) \quad \text{where} \quad \mathbf{a}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}$$

repeating until $\mathbf{h}^{(l)}$ is found. This process is called **forward propagation** and is shown in the form of a diagram in Fig. 4.1 and in Algorithm 1. When a $\mathbf{W}^{(k)}$ contains only, or mostly non-zero entries, the layer is called **dense**. If all the layers are dense then the entire network is called dense.

The output of the last layer $\hat{\mathbf{y}} := \mathbf{h}^{(l)}$ is used to compute a loss function $L(\hat{\mathbf{y}}, \mathbf{y})$ representing the similarity between the output of the network $\hat{\mathbf{y}}$ and the true value $\mathbf{y}$.

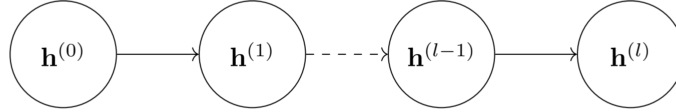


Figure 4.1: Forward Propagation: Let $\mathbf{h}^{(0)} = \mathbf{x}$ and $\mathbf{h}^{(k)} = g^{(k)}(\mathbf{a}^{(k)})$ where $\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$ for $k = 1, ..., l$.

Algorithm 1 Forward Propagation

1: $\mathbf{h}^{(0)} = \mathbf{x}$
2: **for all** k in $1, 2, ..., l$ **do**:
3: $\quad \mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}$
4: $\quad \mathbf{h}^{(k)} = g^{(k)}(\mathbf{a}^{(k)})$
5: $\hat{\mathbf{y}} = \mathbf{h}^{(l)}$
6: $J = L(\hat{\mathbf{y}}, \mathbf{y})$

In a feedforward network, the input $\mathbf{x}$ is passed into the network, and moves through the network to obtain output $\hat{\mathbf{y}}$. During training forward propagation is used to produce cost J in Line 6 of Algorithm 1. The cost J is often written as $J(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ represents the parameters of the model. In practical applications, forward propagation is not applied individually to examples $(\mathbf{x}, \mathbf{y})$, but in batches, where the loss is computed over a batch of examples. This has the advantage of faster computation times.

4.4 Back-Propagation

The **back-propagation** algorithm computes the gradients of the loss function with respect to the parameters of the network. Let the function f be as in Section 4.2 and let $J := L(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$ be the loss function. Then back-propagation computes $\nabla_{\mathbf{W}^{(k)}} J$ and $\nabla_{\mathbf{b}^{(k)}} J$ for $k = l, l-1, ..., 1$. This can be done iteratively using the chain rule as shown in Algorithm 2, computing backwards from the last layer. The gradients are required in order optimise the weights to reduce the loss in forward propagation.

In Algorithm 2, the same subexpressions may be recomputed several times depending on the neural network. The algorithm could have exponential runtime due to these repetitions. In practice, a more generalised form of back-propagation is used and for further details refer to Goodfellow et al. [2016, Section 6.5.6]. Intermediate results can be saved avoiding recomputation.

Algorithm 2 Back-Propagation

1: $\mathbf{u} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$
2: **for all** k in $l, l-1, ..., 1$ **do**:
3: $\nabla_{\mathbf{b}^{(k)}} J = \mathbf{u} g^{(k)'}(\mathbf{a}^{(k)})$
4: $\nabla_{\mathbf{W}^{(k)}} J = \mathbf{u} g^{(k)'}(\mathbf{a}^{(k)})\mathbf{h}^{(k-1)}$
5: $\mathbf{u} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)} \mathbf{u} g^{(k)'}(\mathbf{a}^{(k)})$

4.5 Parameter Optimisation

Let p_d be the empirical distribution given by the training set. Let p_{true} be the data-generating distribution, which is unknown, unlike p_d. Let the function f be as in Section 4.2. The empirical loss is given by the average loss over the training set

$$J(\boldsymbol{\theta}) = \mathbf{E}_{\mathbf{x},\mathbf{y}\sim p_d} L(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}) = \frac{1}{m} \sum_{k=1}^{m} L(f(\mathbf{x}^{(k)}, \boldsymbol{\theta}), \mathbf{y}^{(k)}).$$

The empirical loss is minimised in the hope that this decreases the loss over p_{true}. This has the possibility for overfitting, since if the model has a high enough capacity it can memorise the training set.

In convex optimisation problems, any local minimum is a global minimum. Unfortunately the objective functions in neural networks are non-convex due to the non-linear activation functions (see Section 4.6). In Goodfellow et al. [2014b] it is argued that local minima are not a big problem for training neural networks. In sufficiently large neural networks, most local minima have a small cost and finding the global minimum is unimportant [Goodfellow et al. 2016, Section 8.2.2]. In high dimensional spaces, local minima are rare and saddle points are more common. Stochastic gradient descent has shown to be successful on many different neural network tasks.

Definition 4.1 (Gradient descent)**.** *Let $f : \mathbb{R}^n \to \mathbb{R}$ be a C^1 smooth function. Gradient descent is the method of decreasing f by moving in the direction of the negative gradient. Gradient descent gives the new point*

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

*where $\epsilon \in \mathbf{R}_{>0}$ is the **learning rate**, determining the size of each step.*

For stochastic gradient descent, gradient descent is applied to a **minibatch** of examples from the training set on each iteration. A minibatch is a subset of the data. The minibatch is chosen randomly. The weights are updated in the negative direction of the gradient. See Algorithm 3.

Algorithm 3 Stochastic gradient descent (SGD)

1: **while** stopping condition not met **do**
2: Sample minibatch $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), ..., (\mathbf{x}^{(m)}, \mathbf{y}^{(m)})\}$ from p_d.
3: $\hat{\mathbf{u}} \leftarrow -\epsilon \frac{1}{k} \nabla_{\boldsymbol{\theta}} \sum_k L(f(\mathbf{x}^{(k)}, \boldsymbol{\theta}), \mathbf{y}^{(k)})$.
4: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \hat{\mathbf{u}}$.

Stochastic gradient descent is sensitive to the initial weight values and for feedforward neural networks the weights are initialised to small random values. To improve the SGD algorithm the learning rate is not kept fixed and momentum is introduced. In practice, the learning rate is decreased linearly during learning until some fixed iteration where after ϵ is kept constant. With momentum, the previous gradient is not completely discarded. Line 3 in Algorithm 3 is adjusted with

$$\hat{\mathbf{u}} \leftarrow \hat{\mathbf{u}}\alpha - \epsilon \frac{1}{k} \nabla_{\boldsymbol{\theta}} \sum_k L(f(\mathbf{x}^{(k)}, \boldsymbol{\theta}), \mathbf{y}^{(k)})$$

where α controls the momentum.

4.6 Activation Functions

Activation functions (or units) are the non-linear component of each layer. The non-linearity allows the neural network to learn a large family of functions. Consider a two layer neural network, $\mathbf{h}^{(0)} = \mathbf{x}$

and $\mathbf{h}^{(1)} = g(a)$ where $\mathbf{a} = \mathbf{W}\mathbf{x} + \mathbf{b}$. Then g is called the activation function. Some activation functions are differentiable almost everywhere and the probability of evaluating the function exactly in a non-differentiable point is very small. In practice, the one-sided derivative is taken instead at these points.

Name	Function	Derivative	Image
Identity	$g(x) = x$	$g'(x) = 1$	$(-\infty, \infty)$
Sigmoid	$g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$g'(x) = g(x)(1 - g(x))$	$(0, 1)$
TanH	$g(x) = \tanh(x)$	$g'(x) = 1 - g(x)^2$	$(-1, 1)$
ReLU	$g(x) = \max(0, x)$	$g'(x) = \mathbf{1}_{x \geq 0}$	$[0, \infty)$
Leaky ReLU	$g(x) = \max(0, x) + \alpha \min(0, x)$	$g'(x) = \mathbf{1}_{x \geq 0} + \alpha \mathbf{1}_{x<0}$	$[0, \infty)$
Softmax	$g_i(\mathbf{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$	$\frac{\partial g_i(\mathbf{x})}{\partial x_j} = g_i(\mathbf{x})(\delta_{ij} - g_j(\mathbf{x}))$	(0, 1)

Table 4.1: A list of common activation functions.

Layers with certain activation functions can lead to the gradients of the loss function to approach zero, making the network hard to train. This is called the **vanishing gradient problem**. The derivative of the sigmoid function is close to 0 for large or small values. With many layers using the sigmoid function, this effect is compounded and leads to the weights of the initial layers not being updated effectively during training. It can be partially avoided by using activation functions that are more robust against the vanishing gradient problem such as (leaky) ReLU.

ReLU stands for **rectified linear unit**; they are equivalent to linear functions for half their domain, hence the derivatives remain large when the unit is active. Upon initialisation of the weights, small random positive values will ensure that the unit is active.

Leaky ReLU is a generalisation of ReLU where α is usually chosen to be quite small with $\alpha = 0.01$ a commonly chosen value. With ReLU the gradient is 0 for negative values and can no longer learn when the gradient becomes 0. The leaky ReLU avoids this and guarantees that they receive positive gradient everywhere. In Parametric ReLU, α is a parameter of the activation function that can be learnt.

The tanh and sigmoid functions are related by $\tanh(x) = 2\sigma(2x) - 1$. Sigmoid functions saturate to close to 1 when the input is large, and saturate to close to 0 when the input is small. This saturation means that the derivative is close to 0 making it difficult to continue learning. The sigmoid function is often used only in the last layer with an appropriate cost function that can undo the saturation.

4.7 Cost Functions

Similar to activation functions, the cost function should have a large gradient and not saturate easily in order to help gradient-based learning.

Usually, neural networks are trained using the maximum likelihood. The cost function is given by

$$J(\theta) = -\mathbf{E}_{\mathbf{x},\mathbf{y}p_d} \log p_{\text{model}}(\mathbf{y}|\mathbf{x}).$$

If $p_{\text{model}}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), I)$, where the function f represents a neural network with parameters $\boldsymbol{\theta}$, then ignoring terms that do not depend on the model parameters,

$$J(\theta) = \frac{1}{2}\mathbf{E}_{\mathbf{x},\mathbf{y}\sim p_d}||\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})||^2,$$

where $\mathbf{E}_{\mathbf{x},\mathbf{y}\sim p_d}||\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})||^2$ is known as the mean squared error (MSE).

In the case of a binary classification problem, where $y \in \{0, 1\}$, take the sigmoid function as the activation function of the last layer. In the multi-class classification problem, if the labels are mutually exclusive softmax should be used, and if not sigmoid can be used for each output. There are many activation functions that map the input to the range $(0, 1)$, however the reasoning behind the sigmoid

function is that it arises naturally in the posterior probability distribution in binary classification [Bishop 2009, Section 4.2].

If it is assumed that $p_{\text{model}}(y|\mathbf{x};\boldsymbol{\theta})$ is Bernoulli, in the case of a binary classification problem, the sigmoid function arises as a consequence of generalised linear models (GLMs) [Ng 2018, Section 9.2]. Let $p_{\text{model}}(y|\mathbf{x};\boldsymbol{\theta}) = \text{Bernoulli}(y;\hat{y}(\mathbf{x};\boldsymbol{\theta}))$ then

$$\begin{aligned}\log p(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}) &= \log\left((\hat{y}^{(i)})^{y^{(i)}}(1-\hat{y}^{(i)})^{(1-y^{(i)})}\right)\\ &= y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)}).\end{aligned}$$

From Theorem A.2 maximising the likelihood is equivalent to minimising the cross-entropy. Hence the loss function is chosen as

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

where the function is minimised using a method such as SGD. This loss function is called **binary cross-entropy loss**. The sigmoid as activation function in the last layer is usually coupled with binary cross-entropy loss since the logarithm in the binary cross-entropy loss undoes some of the effects of the exponential in the sigmoid and counteracting the saturating property.

5 Investigation into Specialised Neural Networks

In this section neural network structures are investigated such as to give an overview of the current state of neural network development. Some neural networks will be very useful for solving the problems later on in Sections 7 and 8. Other neural networks seem like they could be useful, but perform poorly when trained on time-series futures data, for example in Section 9.

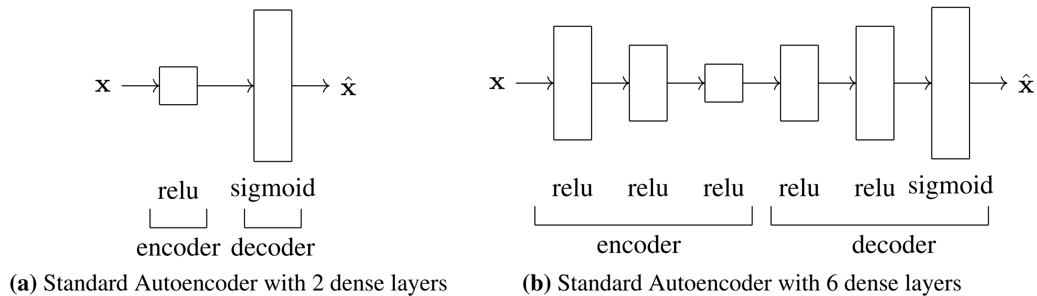
5.1 Autoencoders

Autoencoders are a type of unsupervised feed-forward neural network that learn how to encode data. In a supervised case the data is represented in pairs $(\mathbf{x}, \mathbf{y})$; the same is done now except on pairs $(\mathbf{x}, \mathbf{x})$, that is to say, an autoencoder finds a function f such that $f(\mathbf{x})$ approximates $\mathbf{x}$. They are used to learn a representation of the dataset for dimension reduction. Recently, autoencoders have been extended to learn generative models of the data.

5.1.1 Standard Autoencoder

Autoencoders are based on two parts, an encoder and a decoder. The encoder reduces the dimension of the data and the dimension of the output of the encoder will be referred to as the **latent dimension** and the space is called the **latent space**. The encoder learns a representation of the data in a smaller dimension. The latent dimension should be less than the dimension of the data, otherwise the autoencoder will not need to carry out any dimension reduction and will not be particularly useful. The decoder reconstructs the original data from the latent representation as best as possible. The encoder and decoder can be seen as functions $g(f(\mathbf{x})) = \hat{\mathbf{x}}$ where f represents the encoder, and g represents the decoder. More precisely, write $f(\cdot)$ as $f(\cdot; \boldsymbol{\theta}_1)$ and similarly $g(\cdot)$ as $g(\cdot; \boldsymbol{\theta}_2)$ where $\boldsymbol{\theta}_i$ are the parameters of the model. In learning, the loss function $L(\mathbf{x}, g(f(\mathbf{x}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_2))$ is minimised over $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. In the case that the encoder and decoder are linear, the model is similar to PCA in some respects, both are used for dimension reduction and are linear. They differ in that PCA takes the components that have maximal variance, while a linear autoencoder is trained via SGD or a similar algorithm to reduce the loss.

In diagram form, a standard autoencoder with one layer acting as an encoder, and one layer acting as the decoder, is shown in Fig. 5.1a. More layers can be added to both the encoder and the decoder, giving a representation shown in Fig. 5.1b. Usually, a funnel style of stacked layers is chosen, working from the input to the encoded value and the reverse in the decoder to give the output, such as in Fig. 5.1b.



(a) Standard Autoencoder with 2 dense layers

(b) Standard Autoencoder with 6 dense layers

Figure 5.1: Standard autoencoders with 2 and 6 dense layers respectively. The number of nodes of each layer is illustrated by the height of the rectangle in the diagram.

5.1.2 Variational Autoencoder

One of the drawbacks of the standard autoencoder is that there is no structure in the latent space and the encoded points may be sparsely mapped over the space. The standard autoencoder is very capable of replicating the input but from the sparseness it is not possible to randomly sample from the latent space [Shafkat 2018]. The variational autoencoder [Kingma et al. 2013] gives some structure to the latent space. It forces similar points from the data to be grouped closely in the latent space, and it forces the points to be multivariate normally distributed. In this way, a sense of continuousness is achieved. Points in the latent space that are close together produce similar points in the space the

data lives in. This allows the variational autoencoder to learn encodings of the data that it has not seen before but are variations of encodings that it has seen. Since the points are multivariate normally distributed, it is possible to sample from the latent space.

Consider a dataset $\mathbf{X} = \{\mathbf{x}^{(n)}\}_{n=1,\dots,N}$ of i.i.d. samples of an unknown distribution where $\mathbf{x}^{(n)} \in \mathbb{R}^D$. Assume that $\mathbf{x}^{(n)} \sim p_{\boldsymbol{\theta}^*}(\mathbf{x}|\mathbf{z})$ where $\mathbf{z}^{(n)} \sim p_{\boldsymbol{\theta}^*}(\mathbf{z})$, however these distributions and the parameters $\boldsymbol{\theta}^*$ are unknown. The $\mathbf{z}$ plays the role of noise. These distributions are estimated by parametric distributions $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ and $p_{\boldsymbol{\theta}}(\mathbf{z})$ depending on parameter $\boldsymbol{\theta}$. Let the prior distribution be standard multivariate normal $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{z}|\mathbf{0}, \mathbb{I})$. Let the likelihood be multivariate normal $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{z}|\boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbb{I})$ where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ are dependent on $\mathbf{z}$. The marginal likelihood $p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{z}) p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\, d\mathbf{z}$ and the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) p_{\boldsymbol{\theta}}(\mathbf{z}) / p_{\boldsymbol{\theta}}(\mathbf{x})$ are often intractable because $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ may be dependent on $\boldsymbol{\theta}$ in a complex way. Let $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2\mathbf{I}$ are dependent on $\mathbf{x}$, be an approximation of $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, and assume thereby that $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ is approximately normal. The distribution $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ acts as a probabilistic encoder and $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ acts as a probabilistic decoder. The data is represented by $\mathbf{x}$ with distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ and the encoded data is represented by $\mathbf{z}$ with distribution $p_{\boldsymbol{\theta}}(\mathbf{z})$.

A good approximation for $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ of $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ needs to be found. The variational inference technique (see Appendix A.5) is applied to find $\arg\min_{\phi} \mathbf{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right)$. The evidence lower bound $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})$ is defined as

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}) := \underbrace{-\mathbf{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z})\right)}_{(1)} + \underbrace{\mathbf{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right]}_{(2)},$$

where

(1) is called the regularisation term, which is a measure of how close $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is to $p_{\boldsymbol{\theta}}(\mathbf{z})$,

(2) gives the expected negative reconstruction error.

By Appendix A.5, maximising $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})$ over $\boldsymbol{\phi}$, that is minimising $-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})$ over $\boldsymbol{\phi}$, is equivalent to minimising $\mathbf{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right)$ over $\boldsymbol{\phi}$. By applying the reparametrisation trick to $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, Lemma A.3, rewrite $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon}) = N(\mathbf{0}, \mathbb{I})$. Computing the Kullback-Leibler from Lemma A.1 gives

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}) &= -\mathbf{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z})\right) + \mathbf{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] \\
&= -\mathbf{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) || p_{\boldsymbol{\theta}}(\mathbf{z})\right) + \mathbf{E}_{p(\boldsymbol{\epsilon})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \mathbf{x}))\right] \\
&= \frac{1}{2}\sum_{j=1}^{n}\left(1 + \log\left(\sigma_j^2\right) - \mu_j^2 - \sigma_j^2\right) + \mathbf{E}_{p(\boldsymbol{\epsilon})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}|g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}, \mathbf{x}))\right].
\end{aligned}$$

This is used to define the variational autoencoder. The encoder is given by $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ with parameters $\boldsymbol{\phi}$ and the decoder is given by $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ with parameters $\boldsymbol{\theta}$. Let the loss function be $-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})$. The probabilistic encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ can be written in terms of neural networks $\text{NN}_{\text{enc}}^{\boldsymbol{\mu}}(\mathbf{x})$ and $\text{NN}_{\text{enc}}^{\log \boldsymbol{\sigma}^2}(\mathbf{x})$ representing networks that map from $\mathbf{x} \mapsto \boldsymbol{\mu}$ and $\mathbf{x} \mapsto \log \boldsymbol{\sigma}^2$ respectively. These functions are feedforward networks like the function f in Section 4.2. The 'enc' refers to them being part of the encoder, and the '$\boldsymbol{\mu}$' and '$\log \boldsymbol{\sigma}^2$' respectively refer to what the output of the network should represent. Let

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\text{NN}_{\text{enc}}^{\boldsymbol{\mu}}(\mathbf{x}), \exp(\text{NN}_{\text{enc}}^{\log \boldsymbol{\sigma}^2}(\mathbf{x})/2)\mathbf{I}).$$

Then $\mathbf{z}$ can be written as

$$\mathbf{z} = \text{NN}_{\text{enc}}^{\boldsymbol{\mu}}(\mathbf{x}) + \exp(\text{NN}_{\text{enc}}^{\log \boldsymbol{\sigma}^2}(\mathbf{x})/2) \cdot \boldsymbol{\epsilon}, \quad \text{with} \quad \boldsymbol{\epsilon} \sim N(0, \mathbb{I}).$$

The likelihood $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ is assumed to be multivariate normal, and it is represented as

$$p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}|\text{NN}_{\text{dec}}^{\boldsymbol{\mu}}(\mathbf{z}), I)$$

where $\text{NN}_{\text{dec}}^{\boldsymbol{\mu}} : \mathbf{z} \mapsto \boldsymbol{\mu}$ represents the decoder neural network. For a particular $\mathbf{x}_i$ and $\mathbf{z}_i$ it follows that

$$\begin{aligned}
-\log(p(\mathbf{x}^{(i)}|\mathbf{z}^{(i)})) &= -\log\left(\frac{1}{\sqrt{(2\pi)^n |I|}} \exp\left(-\frac{1}{2}(\mathbf{x}^{(i)} - \text{NN}_{\text{dec}}^{\boldsymbol{\mu}}(\mathbf{z}^{(i)}))^\top I (\mathbf{x}^{(i)} - \text{NN}_{\text{dec}}^{\boldsymbol{\mu}}(\mathbf{z}^{(i)}))\right)\right) \\
&= \frac{1}{2}||\mathbf{x}^{(i)} - \text{NN}_{\text{dec}}^{\boldsymbol{\mu}}(\mathbf{z}^{(i)})||^2 + \text{constant}. \qquad (2)
\end{aligned}$$

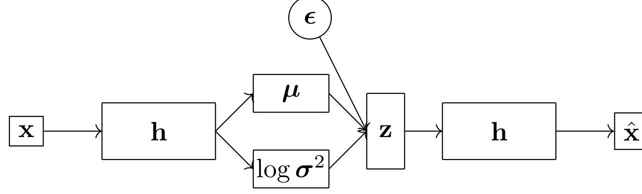


Figure 5.2: Variational Autoencoder diagram. Here $\mathbf{x}$ is the input and $\hat{\mathbf{x}}$ is the output. The $\mathbf{h}$ represents a dense layer. The $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$ refer to dense layers where the output of the layer should be interpreted as $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$ respectively. The ϵ is a multivariate standard normal random variable and it is shown in a circle instead of a rectangle to show that this is a source of stochasticity. The arrows show the direction of forward propagation through the network.

The setup is shown in Fig. 5.2. The encoder is $\mathrm{NN}^{\boldsymbol{\mu}}_{\text{enc}}(\mathbf{x})$ and the decoder is $\mathrm{NN}^{\boldsymbol{\mu}}_{\text{dec}}(\mathbf{z})$.

In carrying out back-propagation it is needed to compute the gradients $\nabla_{\boldsymbol{\theta}} \left(-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})\right)$ and $\nabla_{\boldsymbol{\phi}} \left(-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})\right)$ in each step. Calculating the gradient $\nabla_{\boldsymbol{\theta}} \left(-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})\right)$ is the same as usual. The reason for applying the reparametrisation trick is such that the gradient $\nabla_{\boldsymbol{\phi}} \left(-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x})\right)$ can be calculated because it allows back-propagation through $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$. The encoder and decoder can both be given by a number of ReLU layers. Instead of $\boldsymbol{\sigma}^2$, $\log \boldsymbol{\sigma}^2$ is used because it is more stable for the neural network. It is needed that $\boldsymbol{\sigma}^2$ is always positive, however the output of a neural network can be negative or positive. By using the $\log$, $\boldsymbol{\sigma}^2$ is guaranteed to be positive. Usually $\boldsymbol{\sigma}^2$ is quite small, $0 < \boldsymbol{\sigma}^2 \ll 1$, so applying the $\log$ gives a larger range to the numbers close to 0. The neural network does not have problems with large negative numbers and calculating the exponential of the logarithm is numerically stable.

The method is carried out numerically as follows. Let $\mathbf{x}^{(i)}$ be the ith sample from $\mathbf{X}$. Let $-\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}^{(i)})$ be an empirical estimate of $-\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}^{(i)})$.

$$-\tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}^{(i)}) = -\frac{1}{2} \sum_{j=1}^{n} \left(1 + \log\left((\sigma_j^{(i)})^2\right) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) - \frac{1}{L} \sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} | \mathbf{z}^{(i,l)})$$

where in the last line $\mathbf{z}^{(i,l)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(l)}$ for $\boldsymbol{\epsilon}^{(l)} \sim p(\boldsymbol{\epsilon})$, and take L large enough. The last term can be replaced with the MSE by Eq. (2).

Over a minibatch $\mathbf{X}^M = \{\mathbf{x}^{(i)}\}_{i=1,\ldots,M}$ of $\mathbf{X}$, consider

$$\tilde{\mathcal{L}}^M(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{X}^M) = \frac{1}{M} \sum_{i=1}^{M} \tilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{x}^{(i)}).$$

Now the gradient $\nabla_{\boldsymbol{\theta}, \boldsymbol{\phi}} \left(-\tilde{\mathcal{L}}^M(\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{X}^M)\right)$ can be computed and stochastic gradient descent can be applied.

5.1.3 Adversarial Autoencoders

The adversarial autoencoder [Makhzani et al. 2015] uses a combination of an autoencoder and a GAN (Section 5.2). The GAN is used in the regularisation phase for the latent space.

Let $p_d(\mathbf{x})$ be the data distribution. Let $\mathbf{x} \sim p_d(\mathbf{x})$ be the input and $\mathbf{z}$ the latent code. Let $p(\mathbf{z})$ be the prior distribution of the model, $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ the posterior distribution acting as the probabilistic encoder, and $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ the likelihood acting as the probabilistic decoder. The encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ defines the posterior distribution $q(\mathbf{z})$, which is the distribution on the latent space, given by

$$q(\mathbf{z}) = \int_{\mathbf{x}} q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) p_d(\mathbf{x}) \, d\mathbf{x}$$

The autoencoder consists of an encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and a decoder $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. The encoder may function like in the standard autoencoder where $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is deterministic with the only stochasticity in $q(\mathbf{z})$ is from $p_d(\mathbf{x})$. Alternatively the encoder can be a Gaussian posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2 I)$ where

$\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$ depend on $\mathbf{x}$ like in the variational autoencoder model, where the reparametrisation trick is applied to allow for back-propagation. The prior $p(\mathbf{z})$ may be any distribution in the case of the deterministic posterior, while when using the Gaussian posterior it is set to $p(\mathbf{z}) = N(\mathbf{z}; \mathbf{0}, I)$.

The GAN acts as a regulariser on top of the autoencoder and guides the distribution $q(\mathbf{z})$ to be close to $p(\mathbf{z})$. Samples $\mathbf{z} \sim q(\mathbf{z})$ are labelled as fake and samples $\mathbf{z} \sim p(\mathbf{z})$ are labelled as real. The model works as follows, and is shown in Fig. 5.3.

1. **Train the autoencoder:** Real data $\mathbf{x} \sim p(\mathbf{x})$ is passed through the encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and decoder $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ and then the encoder and decoder are updated based on the reconstruction loss.
2. **Train the discriminator:** Input $\mathbf{z} \sim q(\mathbf{z})$ is labelled as fake and $\mathbf{z} \sim p(\mathbf{z})$ is labelled as real. The discriminator is trained to learn this categorisation.
3. **Train the generator:** Real data $\mathbf{x} \sim p(\mathbf{x})$ is fed into the encoder $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, and the output $\mathbf{z} \sim q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ is fed into the discriminator. The encoder (or generator) is trained to fool the discriminator into believing $\mathbf{z}$ is distributed according to $p(\mathbf{z})$.

Refer to Section 5.2.1 for the precise training of the discriminator and generator forming the generative adversarial part of the model.

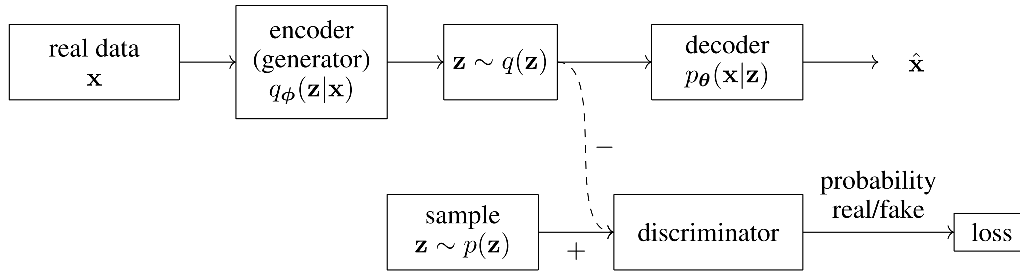


Figure 5.3: Adversarial Autoencoder diagram. The $\mathbf{x}$ is the input and the $\hat{\mathbf{x}}$ is the output. For the autoencoder, only the top row is used. For the discriminator, the $-$ represents $\mathbf{z}$ that are labelled as fake and the $+$ represents the $\mathbf{z}$ that are labelled as real.

The adversarial autoencoder and the variational autoencoder are both generative autoencoder models, however there are some clear differences. For the variational autoencoder, the Kullback-Leibler divergence is used to guide $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ to be similar to the prior distribution $p\boldsymbol{\theta}(\mathbf{z})$. For the adversarial autoencoder the aggregated posterior distribution $q(\mathbf{z})$ is guided to be similar to the prior $p(\mathbf{z})$ using adversarial training. Advantages of the adversarial autoencoder is that it only needs to be able to sample from the prior distribution $p(\mathbf{z})$ while the variational autoencoder needs the exact form. This allows for complex choices for the prior distribution. Experiments for a comparison of the Adversarial Autoencoder against the Variational Autoencoder on labelled data (MNIST) has been done in Makhzani et al. [2015, Section 2.1]. The clustering in the Adversarial case is denser than in the Variational case for supervised learning.

5.2 Generative Adversarial Networks

Generative models are models which learn an approximation $p_{\boldsymbol{\theta}}(\mathbf{x})$ to the probability distribution $p_d(\mathbf{x})$ over the input space. In the following sections the standard generative adversarial (GAN) model is introduced (see Section 5.2.1). An extension of this is the conditional GAN, described in Section 5.2.2. A more robust version of the GAN is the Wasserstein GAN, explained in Section 5.2.3. An improved version of it is examined in Section 5.2.4. A discussion about using GAN methods for anomaly detection can be found in Section 5.2.5.

5.2.1 Standard GAN

Generative adversarial networks consist of two networks, a generator and a discriminator, and they are played against each other in a game theory type scenario [Goodfellow et al. 2014a]. Let $p_d(\mathbf{x})$ be the probability distribution over the data, defined on space $\mathcal{X}$. Let $p(\mathbf{z})$ be a prior noise distribution

defined on space $\mathcal{Z}$, usually $p(\mathbf{z})$ is chosen to be Gaussian. Define the generator as a neural network $G : \mathcal{Z} \times \mathbb{R}^d \to \mathcal{X}$, written as $G_{\boldsymbol{\theta}}(\mathbf{z})$ where $\mathbf{z} \in \mathcal{Z}$ and $\boldsymbol{\theta} \in \mathbb{R}^d$ are the parameters of the network. For random variable $\mathbf{z} \sim p(\mathbf{z})$, let $\mathbf{x} = G_{\boldsymbol{\theta}}(\mathbf{z})$ and call $p_{\boldsymbol{\theta}}(\mathbf{x})$ distribution of $\mathbf{x}$. Let the discriminator be given by the neural network $D : \mathcal{X} \times \mathbb{R}^d \to (0, 1)$, written as $D_{\boldsymbol{\theta}}(\mathbf{x})$. The discriminator gives a value corresponding to the likeliness that the $\mathbf{x}$ was sampled from $p_d(\mathbf{x})$ instead of $p_{\boldsymbol{\theta}}(\mathbf{x})$. The model can be described as a two-player zero-sum game, where the discriminator receives payoff $v(G_{\boldsymbol{\theta}}, D_{\boldsymbol{\theta}})$ and the generator receives payoff $-v(G_{\boldsymbol{\theta}}, D_{\boldsymbol{\theta}})$ with

$$v(G_{\boldsymbol{\theta}}, D_{\boldsymbol{\theta}}) = \mathbf{E}_{\mathbf{x} \sim p_d(\mathbf{x})} \log D_{\boldsymbol{\theta}}(\mathbf{x}) + \mathbf{E}_{\mathbf{z} \sim p(\mathbf{z})} \log \left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z}))\right).$$

Each player tries to maximise its payoff. The optimal solution for G is

$$G^* = \arg\min_G \max_D v(G, D).$$

In the payoff function, note $D_{\boldsymbol{\theta}}(\mathbf{x}), 1 - D_{\boldsymbol{\theta}}(G(\mathbf{z})) \in (0, 1)$ hence $\log D_{\boldsymbol{\theta}}(\mathbf{x}), \log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z}))\right) \in (-\infty, 0)$. The first term describes how well the discriminator can assign high probability to samples from $p_d(\mathbf{x})$. The second term describes how well the generator can fool the discriminator into assigning samples $\mathbf{x}$ from $p(\mathbf{x})$ with high probability of coming from $p_d(\mathbf{x})$.

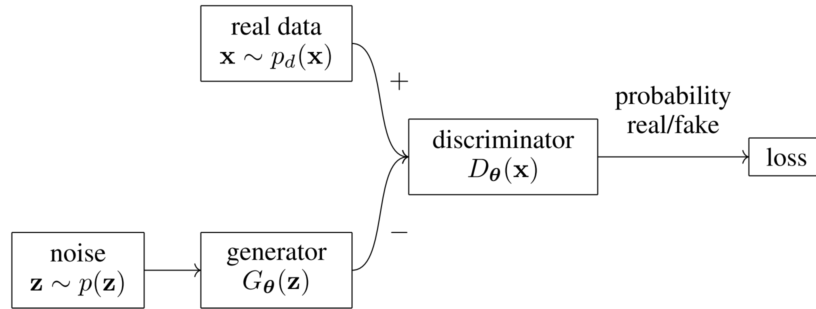


Figure 5.4: GAN diagram. For training the discriminator, the samples $\mathbf{x}$ labelled as true are represented by the $+$ and the samples labelled as false are represented by the $-$. The arrows show the direction of forward propagation through the network.

The generator network can no longer improve when the discriminator cannot differentiate between the two distributions, i.e. $D_{\boldsymbol{\theta}}(\mathbf{x}) = 1/2$. In practice, instead of training $G_{\boldsymbol{\theta}}$ to minimise $\log\left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z}))\right)$, $G_{\boldsymbol{\theta}}$ can be trained to maximise $\log D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z}))$. The discriminator and the generator are trained separately. The training procedure is given by

1. Train discriminator on $(\mathbf{x}, 1)$ where $\mathbf{x} \sim p_d(\mathbf{x})$, with loss $L(D_{\boldsymbol{\theta}}(\mathbf{x}), 1)$.
2. Train discriminator on $(\mathbf{x}, 0)$ where $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x})$, with loss $L(D_{\boldsymbol{\theta}}(\mathbf{x}), 0)$.
3. Train generator (to have the discriminator label samples as valid) on $(\mathbf{z}, 1)$ where $\mathbf{z} \sim p(\mathbf{z})$, with loss $L(D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z})), 1)$.

Learning in the generative adversarial framework can have its difficulties. When $\max_D v(G_{\boldsymbol{\theta}}, D_{\boldsymbol{\theta}})$ is convex in $G_{\boldsymbol{\theta}}$, there is asymptotic consistency to the true G^* [Goodfellow et al. 2016, Section 20.10.4]. In practice, this is not the case and it may lead to the model underfitting, or being unstable. It can be the case that the discriminator learns much faster than the generator. If the discriminator is very good, compared to the generator, then it becomes hard to train the generator since the generator is unable to fool the discriminator. To remedy this it is possible to train the generator more than the discriminator in each round. In practice, it is found that as the discriminator gets better the rate at which the generator improves becomes smaller [Arjovsky et al. 2017a; Goodfellow et al. 2016]. In Arjovsky et al. [2017a] the author goes into details explaining the instability of standard GANs and shows that if the supports of $p_d(\mathbf{x})$ and $p(\mathbf{x})$ are disjoint or if the supports lie in a low dimensional space, then there exists a perfect discriminator, that is to say the discriminator is constant on both spaces and therefore nothing is learnt with back-propagation. In Section 5.2.3 the Wasserstein GAN is examined, which promises better performance and better stability compared to the standard GAN.

5.2.2 Conditional GAN

The conditional GAN is an extension of the standard GAN with the addition of conditional data being fed as input to the generator and the real data. By conditioning the model on additional

information it is possible to direct the data generation process. Let the generator be the feed-forward neural network $G : \mathcal{Z} \times \mathcal{Y} \times \mathbb{R}^d \to \mathcal{X}$, written as $G_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{y})$. For $\mathbf{z} \sim p(\mathbf{z})$ and some $\mathbf{y} \in \mathcal{Y}$, call $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{y})$ the distribution of $G_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{y})$. Let the discriminator be the feed-forward neural network $D : \mathcal{X} \times \mathcal{Y} \times \mathbb{R}^d \to \mathbb{R}$, written as $D_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{y})$. Fig. 5.5 shows the GAN from Fig. 5.4 with the addition of conditional data $\mathbf{y}$ fed into the generator, the discriminator and the real data sampling.

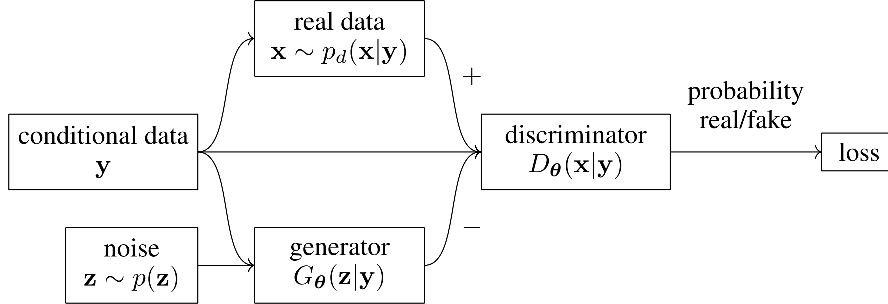


Figure 5.5: Conditional GAN diagram. This is an extension of the GAN diagram shown in Fig. 5.4 where the conditional data $\mathbf{y}$ has been added.

Conditioned on $\mathbf{y}$ the payoff is now written as

$$v(G_{\boldsymbol{\theta}}, D_{\boldsymbol{\theta}}) = \mathbf{E}_{\mathbf{x} \sim p_d(\mathbf{x})} \log D_{\boldsymbol{\theta}}(\mathbf{x} \,|\, \mathbf{y}) + \mathbf{E}_{\mathbf{z} \sim p(\mathbf{z})} \log \left(1 - D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z} \,|\, \mathbf{y}) \,|\, \mathbf{y})\right).$$

The main application for conditional GANs in this project is for the use of time-series data.

Consider data in a time-series setting, where samples from $p_d(\mathbf{x})$ are windows of time-series data. A **window** is an interval of time-series data of a fixed length. The next window is the directly following interval of the same fixed length. Let $\mathbf{y} \sim p_d(\mathbf{x})$, so $\mathbf{y}$ is a window of time-series data. Let $\mathbf{x} \sim p_d(\mathbf{x}|\mathbf{y})$ be the next window of time-series data that directly follows from $\mathbf{y}$. The generator is also conditioned on $\mathbf{y}$ and therefore the generator should learn the next window of time-series data given the window $\mathbf{y}$ of time-series data. The training procedure is given by

1. Train discriminator on $(\mathbf{x}, 1)$ where $\mathbf{x} \sim p_d(\mathbf{x}|\mathbf{y})$, with loss $L(D_{\boldsymbol{\theta}}(\mathbf{x} \,|\, \mathbf{y}), 1)$.
2. Train discriminator on $(\mathbf{x}, 0)$ where $\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{y})$, with loss $L(D_{\boldsymbol{\theta}}(\mathbf{x} \,|\, \mathbf{y}), 0)$.
3. Train generator (to have the discriminator label samples as valid) on $(\mathbf{z}, 1)$ where $\mathbf{z} \sim p(\mathbf{z})$, with loss $L(D_{\boldsymbol{\theta}}(G_{\boldsymbol{\theta}}(\mathbf{z} \,|\, \mathbf{y}) \,|\, \mathbf{y}), 1)$.

5.2.3 Wasserstein GAN

Wasserstein GANs (WGANs) [Arjovsky et al. 2017b] do not require a careful balance in the training of the discriminator and the generator such as is needed for the standard GAN model. The loss of the WGAN shows properties of convergence and one does not need to examine generated samples to find whether one model is better than another. It is enough to look at the loss value since it is proportional to the quality of the model. In practice, WGANs are more robust than standard GANs and have proven to perform well on generator architectures where GANs have performed poorly on.

Assume the data follows the unknown distribution $p_d(\mathbf{x})$, where $p_{\boldsymbol{\theta}}(\mathbf{x})$ acts as an approximation to $p_d(\mathbf{x})$. One method for this approximation is to minimise the Kullback-Leibler divergence $\mathbf{KL}(p_d || p_{\boldsymbol{\theta}})$; this is used in the Variational autoencoder and the standard GAN models. The disadvantage of this method is that the support of one distribution is not contained in the support of the other, so the Kullback-Leibler divergence is not defined [Arjovsky et al. 2017a]. To fix this, a noise term can be added to the model distribution. Consider the Wasserstein-1 in Definition 5.1.

Definition 5.1 (Earth-Mover distance or Wasserstein-1)**.** *Let $\mathcal{X}$ be a compact metric space and let Σ the set of all Borel subsets of $\mathcal{X}$. Let Prob$(\mathcal{X})$ be the space of all probability measures defined on $\mathcal{X}$. Let $p_d, p_g \in$ Prob$(\mathcal{X})$. The Earth-Mover distance is given by*

$$W(p_d, p_g) = \inf_{\gamma \in \Pi(p_d, p_g)} \mathbf{E}_{(x,y) \sim \gamma} \left[||x - y||_2 \right],$$

where $\Pi(p_d, p_g)$ is the set of all joint distributions $\gamma(x, y)$ whose marginals are p_d and p_g respectively.

Intuitively, each distribution can be seen as a unit amount of dirt on $\mathcal{X}$ and the distance is the minimum cost of of reshaping the dirt from one distribution such that it takes the shape of the other distribution. The cost to move the dirt is the amount of dirt to be moved multiplied by the distance to be moved. The Wasserstein-1 is a metric [Villani 2009, Chapter 6]. In Arjovsky et al. [2017b] it is shown that the Wasserstein-1 metric is more sensible than the Kullback-Leibler divergence as a cost function when learning distributions with support on low dimensional manifolds because it does not matter if the supports are disjoint.

Definition 5.2 (K-Lipschitz function)**.** *Given two metric spaces (X, d_X) and (Y, d_Y), a function $f : X \to Y$ is called K-Lipschitz, for $K \in \mathbf{N}$, if for all $x_1, x_2 \in X$*

$$d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2).$$

Theorem 5.1 (Kantorovich-Rubinstein duality)**.** *The Kantorovich-Rubinstein duality shows that [Edwards 2011]*

$$W(p_d, p_{\boldsymbol{\theta}}) = \sup_{||f||_L \leq 1} \mathbf{E}_{\mathbf{x} \sim p_d} [f(\mathbf{x})] - \mathbf{E}_{\mathbf{x} \sim p_{\boldsymbol{\theta}}} [f(\mathbf{x})] \quad (3)$$

where the supremum is taken over all 1-Lipschitz functions $f : \mathcal{X} \to \mathbb{R}$.

Let $\mathcal{X}$ be a compact metric space. Let $\mathbf{z} \sim p(\mathbf{z})$ be a Gaussian random variable over another space $\mathcal{Z}$. Let $g : \mathcal{Z} \times \mathbb{R}^d \to \mathcal{X}$ be a deterministic function denoted by $g_{\boldsymbol{\theta}}(z)$ where $\boldsymbol{\theta} \in \mathbb{R}^d$ and $z \in \mathcal{Z}$. Let $p_{\boldsymbol{\theta}}$ denote the distribution of $g_{\boldsymbol{\theta}}(\mathbf{z})$. More precisely, let $g_{\boldsymbol{\theta}}$ be a feed-forward neural network parameterised by $\boldsymbol{\theta}$. By Arjovsky et al. [2017b, Corollary 1], $g_{\boldsymbol{\theta}}$ is locally Lipschitz and there are local Lipschitz constants $L(\boldsymbol{\theta}, \mathbf{z})$ such that $\mathbf{E}_{\mathbf{z} \sim p(\mathbf{z})} [L(\boldsymbol{\theta}, \mathbf{z})] < \infty$. It follows by Arjovsky et al. [2017b, Theorem 3] that there is a solution $f : \mathcal{X} \to \mathbb{R}$ to the problem

$$\max_{||f||_L \leq 1} \mathbf{E}_{\mathbf{x} \sim p_d} [f(\mathbf{x})] - \mathbf{E}_{\mathbf{x} \sim p_{\boldsymbol{\theta}}} [f(\mathbf{x})] \quad (4)$$

and

$$\nabla_{\boldsymbol{\theta}} W(p_d, p_{\boldsymbol{\theta}}) = -\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [\nabla_{\boldsymbol{\theta}} f(g_{\boldsymbol{\theta}}(\mathbf{z}))]$$

where both terms are well-defined.

To find a function f that is a good estimator for the f in Eq. (4) a neural network can be used. Let $\{f_w\}_{w \in \mathcal{W}}$ where $\mathcal{W}$ is a compact space, $w \in \mathcal{W}$ represents a set of weights and f_w a feed-forward neural network parameterised by w. In the proof of Arjovsky et al. [2017b, Corollary 1] it is shown that the derivative of a feed-forward neural network is bounded when the weights are bounded, hence compactness of $\mathcal{W}$ implies that f_w are K-Lipschitz for some K that only depends on $\mathcal{W}$. Instead of Eq. (4) consider the estimate of $W(p_d, p_{\boldsymbol{\theta}})$ given by

$$W(p_d, p_{\boldsymbol{\theta}}) \approx \max_{w \in \mathcal{W}} \mathbf{E}_{\mathbf{x} \sim p_d(\mathbf{x})} [f_w(\mathbf{x})] - \mathbf{E}_{\mathbf{x} \sim p_{\boldsymbol{\theta}}(\mathbf{x})} [f_w(\mathbf{x})] \,. \quad (5)$$

The neural network should optimise

$$\min_{\boldsymbol{\theta}} \max_{w \in \mathcal{W}} \mathbf{E}_{\mathbf{x} \sim p_d(\mathbf{x})} [f_w(\mathbf{x})] - \mathbf{E}_{\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})} [f_w(g_{\boldsymbol{\theta}}(\mathbf{z}))] \,. \quad (6)$$

Let $n_{\text{critic}} \in \mathbf{N}$. Carry out stochastic gradient descent over w on $W(p_d, p_{\boldsymbol{\theta}})$ (Eq. (5)), and repeat this n_{critic} times. The critic is the name of the function f_w, and by applying this method a good critic f_w is found that can be used as the maximum f_w in Eq. (6). Note that sampling from $g_{\boldsymbol{\theta}}(\mathbf{z})$ with $\mathbf{z} \sim p(\mathbf{z})$, gives samples from $p_{\boldsymbol{\theta}}(\mathbf{x})$. The new parameters w may not lie in the space $\mathcal{W}$. A simple way to enforce the compact constraint is to clip the weights such that they fit in $\mathcal{W}$. Let c be some small constant, for example $c = 0.01$ and let $\mathcal{W} = [-c, c]^l$ then after after each weight update, the new w is forced to be in $\mathcal{W}$ by replacing values outside of $[-c, c]$ with the boundary value. After training the critic, the generative model $g_{\boldsymbol{\theta}}$ is trained. Carry out stochastic gradient descent over $\boldsymbol{\theta}$ on $W(p_d, p_{\boldsymbol{\theta}})$ (Eq. (5)) by back-propagating over $-\mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [\nabla_{\boldsymbol{\theta}} f_w(g_{\boldsymbol{\theta}}(\mathbf{z}))]$. Weight clipping is not a good way to enforce the Lipschitz constraint and it leads to optimisation difficulties [Arjovsky et al. 2017a]. In Section 5.2.4 a different method is covered to improve this.

5.2.4 Wasserstein GAN Improved

In Gulrajani et al. [2017] an alternative to weight clipping for the Wasserstein GAN is found. The motivation behind this is that the weight clipping in WGAN has lead to optimisation problems and that it biases the critic towards simpler functions.

A differentiable function is 1-Lipschitz if and only if the derivative in absolute value is less than or equal to 1 everywhere. This is used as a soft constraint on the critic loss instead of weight clipping. Let the critic loss be given by

$$L = \mathbf{E}_{\mathbf{x}\sim p_d(\mathbf{x})}\left[f(\mathbf{x})\right] - \mathbf{E}_{\mathbf{x}\sim p_{\boldsymbol{\theta}}(\mathbf{x})}\left[f(\mathbf{x})\right] + \lambda \mathbf{E}_{\tilde{\mathbf{x}}\sim\tilde{p}}\left[(||\nabla_{\tilde{\mathbf{x}}} f(\tilde{\mathbf{x}})||_2 - 1)^2\right],$$

where $\tilde{\mathbf{x}} \sim \tilde{p}$ is distributed as $\tilde{\mathbf{x}} = u\mathbf{x} + (1-u)\mathbf{y}$ where $\mathbf{x} \sim p_d(\mathbf{x})$, $\mathbf{y} \sim p_{\boldsymbol{\theta}}(\mathbf{x})$, $\mathbf{z} \sim p_{\boldsymbol{\theta}}(\mathbf{z})$, $u \sim U[0,1]$. It is intractable to enforce the gradient constraint everywhere, so instead the distribution $\tilde{p}$ is chosen such that sampling from $\tilde{p}$ is equivalent to sampling uniformly along the line between samples from $p_d(\mathbf{x})$ and from $p_{\boldsymbol{\theta}}(\mathbf{x})$. The coefficient $\lambda \in \mathbf{R}_{>0}$ is chosen by experimentation, and by Gulrajani et al. [2017], the value of $\lambda = 10$ works well in general. In this paper the value $\lambda = 10$ is used and provides good results, further tuning of λ has not been carried out.

The constraint penalises the critic if the derivative is larger than 1, which is the required constraint, but it also penalises if the derivative is less than 1. In Gulrajani et al. [2017, Appendix C] a one-sided constraint is considered, however empirically the two-sided constraint performs slightly better.

5.2.5 GAN for Anomaly Detection

The discriminator is learning to discriminate whether a sample came from the real data $p_d(\mathbf{x})$ or the generator $p(\mathbf{x})$. As the GAN is trained, the probability distribution $p(\mathbf{x})$ approaches the probability distribution $p_d(\mathbf{x})$ and the discriminator finds it harder to differentiate between $p(\mathbf{x})$ and $p_d(\mathbf{x})$ and outputs values around $\frac{1}{2}$. The discriminator is not a generalised detector of anomalous data and it cannot be used to detect whether a sample resembles $p_d(\mathbf{x})$ or not. It is trained to maximise the probability of assigning real training samples a positive label and generated samples a negative label.

Research in anomaly detection GANs has been made, for example with the method anoGAN [Schlegl et al. 2017]. With anoGAN a standard GAN is trained on healthy data. A new loss function called the residual loss is introduced, where the idea is for a sample $\mathbf{x}$ in the test set to find a $\mathbf{z}$ such that $||\mathbf{x} - G(\mathbf{z})||_1$ is minimised. This method for anomaly detection is not examined closer, however it is a good starting point for further research.

5.3 Convolutional Networks

In the previous neural networks the layers are all fully connected (dense), which means that each node is connected to all the nodes in the next layer. Instead of dense layers, convolutional layers and pooling layers can be used, especially when dealing with very large inputs such as for data in the form of images.

Convolutional networks can capture spatial and temporal dependencies [Saha 2018], usually in images through the use of filters. Assume the input to the convolutional network is an image with 3 dimensions, width, height, and colour values. In a **convolutional layer**, a number of filters are passed over the width and the height of the input by convolution and dot products are computed between the entries of the filter and the values of the image that the filter is on. Each filter detects different details in an image. With a convolutional layer, each node is connected to only a local region of the nodes. In a **pooling layer**, downsampling is performed along the width and height to reduce the image size. For a complete explanation of convolutional neural networks, refer to Karpathy [2015].

Instead of applying convolutional networks to images, they will be applied to time-series data. Some research into this has been tried [Brownlee 2018c; Eddy 2018] however convincing results could not be found.

5.4 Recurrent Networks

Recurrent neural networks (RNNs) are unlike the feedforward networks from earlier sections, in that they have loops where the output is fed back into the input and they can have internal memory. They can use their internal memory to process sequences. They have successfully been used for handwriting and speech recognition.

The basic form of a RNN is shown in Fig. 5.6. The output $v^{(t)}$ of the unit at time t is fed into the input of the unit at time $t+1$. In this way a chain of RNN units can be formed. Two types of RNNs in Section 5.4.1 and Section 5.4.2 are examined.

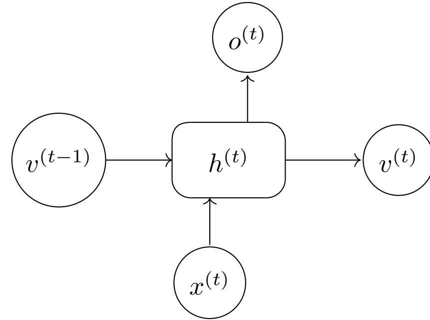


Figure 5.6: Recurrent neural network unit. The output $v^{(t)}$ of one unit is fed into the input of the next. The $x^{(t)}$ represents an additional input to the unit and $o^{(t)}$ represents the output of the unit.

5.4.1 Long Short-Term Memory

In simple recurrent neural networks the output is fed back into the input. They work to an extent, however they suffered from the vanishing gradient problem (see Section 4.6). Long short-term memory networks (LSTMs) address this problem and one of their strengths is remembering information for a long time.

An LSTM unit consists of an input gate, an output gate and a forget gate [Hochreiter et al. 1997]. The structure of an LSTM unit is given in Fig. 5.7. The output of an LSTM unit is the input of the next unit.

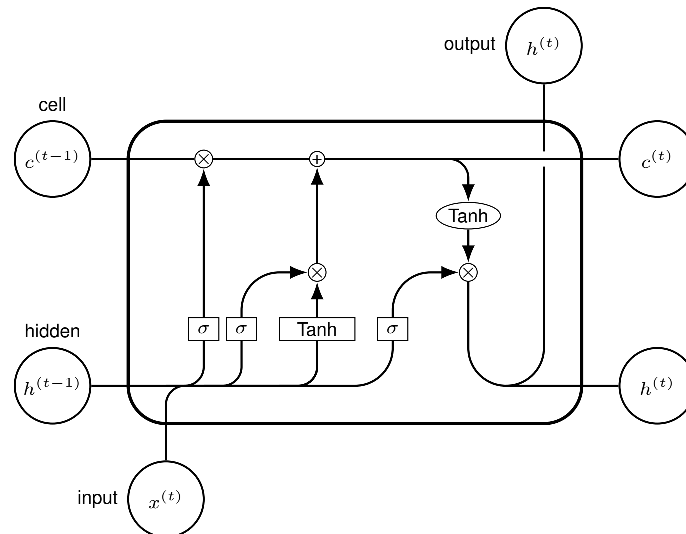


Figure 5.7: LSTM unit. Source: Leon [2018]. Merging arrows mean concatenation operation and splitting arrows mean copying operation. The outputs $c^{(t)}$ and $h^{(t)}$ are used as inputs of the next unit in the chain. Each unit can take an input $x^{(t)}$ and can return an output $h^{(t)}$.

In Fig. 5.7 the top horizontal line running through the cell represents the cell state. The LSTM can add information to the cell state via the gates. There are three vertical sections in the LSTM unit connecting the hidden state with the cell state. From left to right they are called the forget gate, the input gate, and the output gate.

The forget gate looks at $h^{(t-1)}$ and $x^{(t)}$ and using the sigmoid layer to multiply each element of $c^{(t-1)}$ by a number in the interval $[0, 1]$ where 0 allows previous value to be forgotten, and 1 allows the value to be kept.

In the input gate, the sigmoid layer determines which values should be updated, and the Tanh layer creates a new candidate vector $\tilde{c}^{(t)}$ which is added to the cell state.

The output gate is used to give the output of the LSTM unit and it is based on the cell state. The sigmoid layer determines which parts of the cell state will be outputted and the Tanh function scales the values to the range $[-1, 1]$. For a more detailed explanation, see Olah [2015].

5.4.2 Gated Recurrent Unit

The gated recurrent unit (GRU) [Cho et al. 2014] is a type of recurrent neural network that is similar to the LSTM but with fewer parameters. In comparison with an LSTM unit, the GRU unit has a hidden state but not a cell state, furthermore it only has two gates, a relevance (or reset) gate and an update gate. In Fig. 5.8 the reset gate is represented by r_t and the update gate by z_t. The update gate is similar to the forget and input gate of an LSTM, and it decides what information from the previous state should be retained and what new information to add. The difference is that the LSTM controls the amount of new memory being added with the input gate and this is independent from the forget gate.

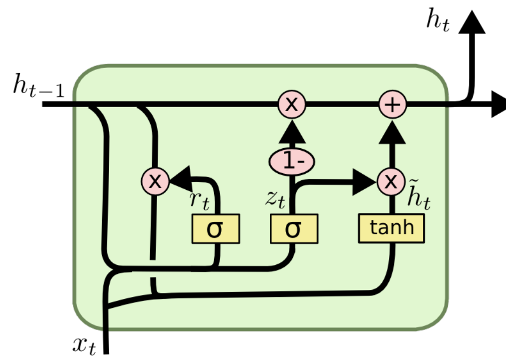


Figure 5.8: GRU unit. Source: Olah [2015]. The GRU unit is similar to Fig. 5.7 but simplified since the output of the unit $h^{(t)}$ is both the input to the next unit in the chain but also the output of the unit itself. Each unit takes an input $x^{(t)}$.

In Fig. 5.8 the drawn gates have the following form

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\
\tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\
h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t.
\end{aligned}
$$

At time t the GRU unit updates to hidden state h_t based on the hidden state h_{t-1} from time $t-1$ and the new information x_t. Note that the output of the GRU unit is equal to its hidden state, that is h_t. The relevance gate r_t decides how relevant the previous information is. r_t has a sigmoid activation function so the values lie in the range $[0, 1]$. This is multiplied with h_{t-1} and the value is used to create the new candidate state $\tilde{h}_t$. The update gate is denoted by z_t and it decides how much of the previous cell state should be retained and how much of the candidate state $\tilde{h}_t$ should be used. This is more clearly shown in the formula above for h_t.

In Chung et al. [2014] an empirical comparison of GRU and LSTM is made in the case of music data and speech signal data. No concrete conclusion was made on which of the two units was better, however both were shown to be clearly better than a traditional tanh unit.

6 Data Preparation

Neural networks learn a mapping from input variables to output variables. The scale of each feature in the input may be different. Unscaled variables may lead to unstable, slow, or failed learning. This can lead to exploding gradients and large weight values [Brownlee 2019]. To remedy this, all the inputs are required to be in a comparable range. The preparation of the data before using it to train a neural network is called **data preprocessing**, similarly the output of a neural network requires post-processing. One of the most common types of preprocessing consists of linear rescaling of the input variables. Furthermore, in this section methods such as standardisation and power transforms are discussed.

The time-series data that is used for training is of a number of oil commodities with different pricing ranges. The preprocessing methods in Section 6.1 are applied to the data such that the ranges of the data is converted to a comparable range.

6.1 Preprocessing Methods

In this section some of the most well-known methods for data preparation are considered. In general it may not be necessary to take the inverse operation of a data preparation method, for example in the setting of a classification problem. However, it is important to be able to take the inverse operation when dealing with regression problems where it is necessary to reconstruct the input. In that case a preprocessing model should have the structure as in Definition 6.1.

Definition 6.1 (Invertible Preprocessing Method)**.** *Given a data set $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1,...,M}$, which is split into training and validation, $\mathbf{X}_{train} = \{\mathbf{x}^{(i)}\}_{i=1,...,k}$ and $\mathbf{X}_{val} = \{\mathbf{x}^{(i)}\}_{i=k+1,...,M}$. The preprocessing method, denoted by a function f can be fitted to the training set. $f_{fit}(\mathbf{X}_{train}) = \mathbf{r} = (r_1, ..., r_n)$ a set of learned parameters. The functions f and f_{inv} have access to the parameters $\mathbf{r}$. The training and validation data can be transformed by f with the inverse function given by f_{inv}, more precisely $f_{inv}(f(\mathbf{x}^{(i)})) = \mathbf{x}^{(i)}$ on the training set, and $f_{inv}(f(\mathbf{x}^{(i)})) \simeq \mathbf{x}^{(i)}$ on the validation set. The size of the learned parameters n, should not be dependent on the size of the dataset M.*

The importance here is that the method is only allowed to calibrate parameters on a section of the data while still expected to perform well on the unseen data. A relaxed version of Definition 6.1 is considered in Definition 6.2 where the inverse function is permitted to know the first value of the vector that it is reconstructing.

Definition 6.2 (Semi-invertible Preprocessing Method)**.** *Consider a function f as in Definition 6.1, where fit to the training set $f_{fit}(\mathbf{X}_{train})$. The method is called semi-invertible if to reconstruct $\mathbf{x}^{(i)}$ from $f(\mathbf{x}^{(i)})$ additional information $x_1^{(i)}$ needs to be given. Moreover, $f_{inv}(f(\mathbf{x}^{(i)}), x_1^{(i)}) = \mathbf{x}^{(i)}$ on the training set, and $f_{inv}(f(\mathbf{x}^{(i)}), x_1^{(i)}) \simeq \mathbf{x}^{(i)}$ on the validation set.*

In Section 6.3 the methods in Definitions 6.1 and 6.2 are applied to standardisation (Section 6.1.1), normalisation (Section 6.1.2) and log-returns (Section 6.1.3) techniques.

6.1.1 Standardisation

With standardisation it is assumed that the data is approximately normally distributed. The data is scaled such that each feature has mean 0 and standard deviation 1. Transform the data to centre it by removing the mean value of each feature, and scaling each feature by dividing by their standard deviation.

To check if the data is approximately normal, a QQ-plot or a histogram can be made. The **Shapiro-Wilk test** quantifies how likely it is that the data was drawn from a standard normal distribution. The **D'Agostino's K^2 test** measures skewness and kurtosis and can be used to measure asymmetry and normality respectively [Brownlee 2018a]. Measuring the normality of the data is not examined in this paper.

Often it is assumed that the data is normally distributed, for example in the setup of variational autoencoders [Kingma et al. 2013, Section 2.1], however this is not always the case. Power transforms aim to map data from any distribution to as close to a Gaussian distribution as possible. They are non-linear functions and the most well-known transforms are the Yeo-Johnson transform and the

Box-Cox transform. The **Box-Cox transform** [Box et al. 1964] transforms y to $y^{(\lambda)}$ given by

$$y^{(\lambda)} = \begin{cases} \frac{(y+\lambda_2)^{\lambda_1}-1}{\lambda_1} & \text{for } \lambda_1 \neq 0, \\ \log(y+\lambda_2) & \text{for } \lambda_1 = 0. \end{cases}$$

where λ_2 is chosen such that $y > -\lambda_2$. The parameter λ_1 is estimated using the likelihood function. The Box-Cox transform is not further used, but it could be interesting to see its effect on the performance of the neural networks shown in Section 7 and Section 8.

6.1.2 Normalisation

In data normalisation, the data is linearly scaled such that it fits in the range $[a, b]$ for $a, b \in \mathbb{R}$. Usually the range $[0, 1]$ or $[-1, 1]$ is taken. In the case of binary cross-entropy loss function the range $[0, 1]$ must be used. Normalisation is usually carried out over the features.

Consider a dataset $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1,\dots,m}$ where $\mathbf{x}^{(i)} = (x_1^{(i)}, \dots, x_n^{(i)}) \in \mathbb{R}^n$. The dataset $\mathbf{X} \in \mathbb{R}^{m\times n}$ can be regarded as a matrix of m rows and n columns. Let $\mathbf{X}_{\text{min}} = (\min_i\{x_1^{(i)}\}, \dots, \min_i\{x_n^{(i)}\})$ and similarly for $\mathbf{X}_{\text{max}}$. Then the normalised data is given by

$$\mathbf{X}' := (b-a)\frac{\mathbf{X} - \mathbf{X}_{\text{min}}}{\mathbf{X}_{\text{max}} - \mathbf{X}_{\text{min}}} + a$$

where the subtraction in the numerator, and the division is carried out row-wise. That is, $(\mathbf{X}-\mathbf{X}_{\text{min}}) \in \mathbb{R}^{m\times n}$ with the subtraction applied to each row vector $\mathbf{x}^{(i)}$ of $\mathbf{X}$. The subtraction $(\mathbf{X}_{\text{max}}-\mathbf{X}_{\text{min}}) \in \mathbb{R}^n$ is done element-wise.

In training, normalisation should be fitted on the training set, that is to say, the vectors $\mathbf{X}_{\text{min}}$ and $\mathbf{X}_{\text{max}}$ should be found using the data in the training set only. If it is also fitted on the basis of data in the validation set, then the network is given information about the validation set. By saving the values $\mathbf{X}_{\text{min}}$ and $\mathbf{X}_{\text{max}}$ the output of a neural network can be re-scaled, in the case of an autoencoder for example.

6.1.3 Log-returns

Log-returns were defined in Definition 2.5. Getting back to the original data, given x_1 is as follows. Let

$$z_1 := \tilde{x}_1, \quad z_i := \exp\left(\frac{y_{i-1}}{k}\right) = \frac{\tilde{x}_i}{\tilde{x}_{i-1}} \quad \text{for } i = 2, \dots, n.$$

Then $z_i\tilde{x}_{i-1} = \tilde{x}_i$ for $i = 2, \dots, n$. The $\tilde{x}_i$ are now recoverable and in turn so are x_i for $i = 2, \dots, n$, assuming that x_1 is known.

6.2 Filtering Autoencoder Output

The output of the autoencoder is in general not as smooth as the input. Examples of autoencoder outputs without smoothing applied can be found in Fig. 7.1. This leads to an examination of methods whereby a smoother curve can be obtained. The easiest solution is to manually apply smoothing to the result of the autoencoder. An extension to this would be to set the smoothing function as the final layer of the decoder in the autoencoder network, where the parameters of the smoothing function are learnable.

Applicable smoothing functions are: the Savitzky-Golay filter, simple moving-average and exponential smoothing. The simple moving average and exponential smoothing models introduce lag relative to the input data. The Savitzky-Golay filter is interesting because it is capable of preserving the minimum and maximum of the noisy data. The filter works by fitting neighbouring points with a polynomial using the linear least squares approximation [Schafer 2011].

Definition 6.3 (Savitzky-Golay)**.** *The Savitzky-Golay takes two parameters: $N \in \mathbf{N}$ where N is the degree of the polynomial to be fitted, and fix $M \in \mathbf{N}$ odd where M is the number of neighbouring points, or window size, that the polynomial is fit to. Let $x_1, x_2, \dots, x_t \in \mathbb{R}$ be a sequence representing the original noisy curve. Give a smooth version of the curve by $y_1, y_2, \dots, y_t \in \mathbb{R}$ as follows. For*

each $j \in \{1, 2, ..., t\}$ consider the error

$$\mathcal{E}_j = \sum_{i=\frac{1-M}{2}}^{\frac{M-1}{2}} (p_j(i) - x_{j+i})^2$$

where $p_j(n)$ is the N-degree polynomial with coefficients $a_{j,0}, ..., a_{j,N} \in \mathbb{R}$ given by

$$p_j(n) = \sum_{k=0}^{N} a_{j,k} n^k$$

The coefficients $a_{j,0}, ..., a_{j,N}$ that minimise $\mathcal{E}_j$ are found. Let $y_j = a_{j,0}$ for all $j \in \{1, 2, ..., t\}$.

By experimentation on futures curves outputted by the autoencoder, taking a polynomial order of 5 and a window size of 23 already leads to good results. The difference between input and output curves of the autoencoder is also reduced by application of the Savitzky-Golay filter, under the MSE and SMAPE performance metrics, although not considerably. What is key is that the filter does not adversely affect the performance of the autoencoders from empirical results.

6.3 Application to Time-Series of Futures Curves

Each futures curve is made up of a number of points called tenors (see Section 2.2); for oil curves from the datasets, each curve is composed of $N = 56$ tenors. Represent the data by $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1,...,M}$ where $\mathbf{x}^{(i)} = (x_1^{(i)}, ..., x_N^{(i)}) \in \mathbb{R}^N$. The data $\mathbf{X}$ can be viewed as an $M \times N$ matrix. Each row $\mathbf{x}^{(i)}$ represents a futures curve and the columns are the tenors, or features as seen in Section 4 and Section 5. Call the number of tenors (or features) the dimension of the time-series data. The data has 56 tenors, so the data is said to have dimension 56. The dimension of the futures curves can be reduced to a dimension of 1, 2, 3, or 4 and this is what will be investigated.

In order to apply compression methods, data preparation needs to be applied to fit the data in the range $[-1, 1]$ or $[0, 1]$. This is required because the original data has different unit sizes depending on the dataset and because the neural network models require data in this range to be able to learn adequately.

The following invertible preprocessing methods (Definition 6.1) over the data are considered:

- Normalisation over the tenors
- Standardisation over the tenors
- Log-returns over the tenors

In normalisation over the tenors, for each feature column, the data is linearly scaled such that it fits in the given range. In standardisation over the tenors, for each feature column, the data is scaled such that it has mean 0 and standard deviation 1. In both normalisation and standardisation methods, the shape of the individual futures curves may be changed and it may result in the smoothness of the curve being lost. The shape of the time-series is preserved for each individual tenor. In log-returns over the tenors, the difference is taken between each curve and the previous one. One can see this as per column in the dataset represented, take log-returns as described in Definition 2.5. To inverse taking log-returns over the tenors, it is necessary to preserve the first curve in the series.

Consider the following semi-invertible preprocessing methods (Definition 6.2) over the futures curves, where the begin price and/or end price value per futures curve is required to restore the curve:

- Normalisation over the curves
- Standardisation over the curves
- Log-returns over the curves

In normalisation over the curves, each curve is individually linearly scaled such that it fits in the given range. In standardisation over the curves, each curve is individually scaled such that it has mean 0 and standard deviation 1. In this case, the shape of the individual futures curves is preserved, however the price changes over time are lost, since each curve is fitted in a certain range. In log-returns over

the curves, for each individual curve take the log-returns as in Definition 2.5. To inverse taking log-returns over the curves, it is necessary to preserve the first tenor of each curve.

The dimension of the curve needs to be reduced from 56 to 1, 2, 3 or 4. By using a semi-invertible method, the begin value needs to be preserved, occupying one dimension, and giving one less dimension whereby to describe the futures curve shape. This makes these methods less than ideal compared to the invertible preprocessing methods. Further analysis of these semi-invertible methods is not taken in this paper and restrict ourselves to the invertible methods.

6.3.1 Normalisation over the Tenors

In Fig. 6.1 normalisation over the tenors of WTI NYMEX is applied and the normalised curves are compared to the original curves. It is important to note that the normalisation does not happen over the curves, if that were the case each curve would be scaled to fit exactly in the interval $[0, 1]$. The curves are normalised over the tenors and this is the reason why the shape of each curve is not necessarily preserved. For curves 1, 2, and 4, the normalised curve has a completely different shape. The shape of the normalised curve is determined by the shape of the original curve and by the distribution of each tenor. The normalised curves do preserve the prices, more precisely, the time-series of scaled prices by looking at a specific tenor has the same shape as original prices. This can be seen by comparing the price ranges of the curves in the first row compared to the second row. However, as can be seen in Fig. 6.1 the shape of the curve is not always preserved.

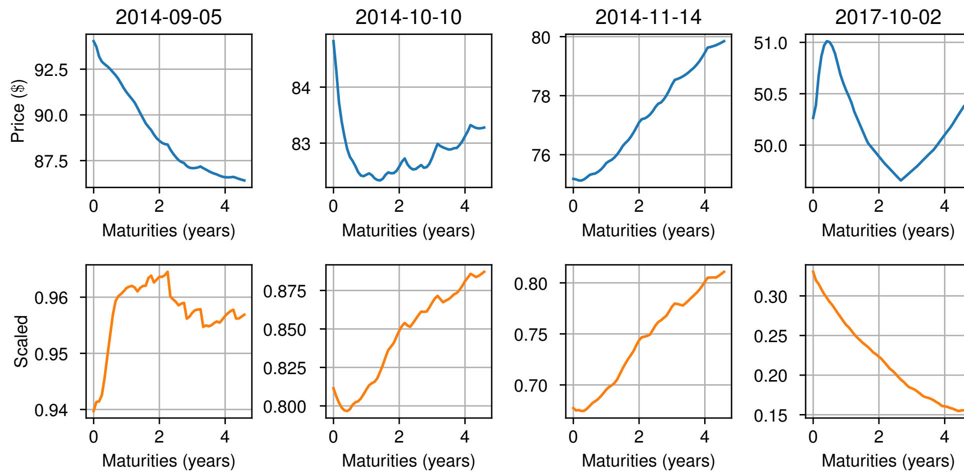


Figure 6.1: A selection of futures curves (first row) from the WTI NYMEX time series compared with their normalised curve (second row). The normalisation is taken over the tenors with the range set to $[0, 1]$.

6.3.2 Standardisation over the Tenors

From Fig. 6.2 it can be seen that standardisation over the tenors is similar to normalisation over the tenors, with the differences being that the normalised curves lie in a larger range than $[0, 1]$ and that the curves seem smoother in some instances.

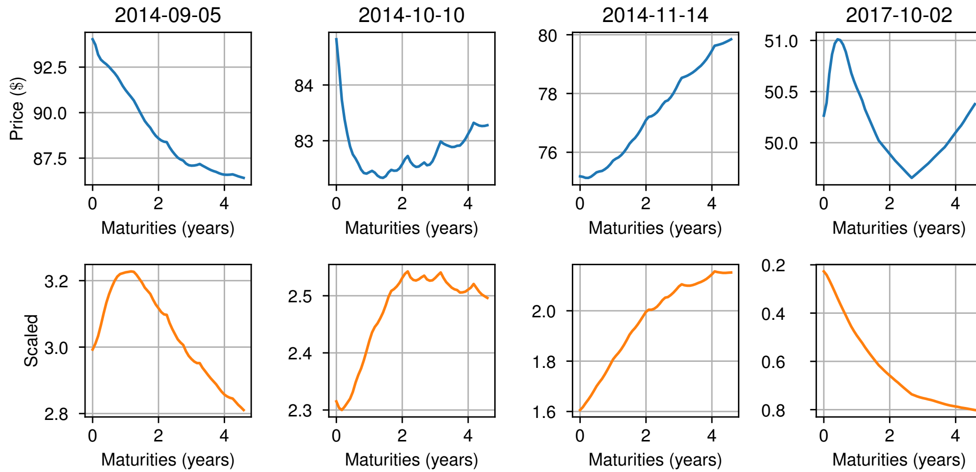


Figure 6.2: A selection of futures curves (first row) from the WTI NYMEX time series compared with their standardised curve (second row). The standardisation is taken over the tenors.

6.3.3 Log-returns over the Curves

The log-returns over the time-series of futures curves for WTI NYMEX is taken, as shown in Fig. 6.3. The log-returns are similar to normalisation and standardisation over the tenors in that the shape of the curves are similar, however the values of the log-returns are now relative prices instead of scaled prices.

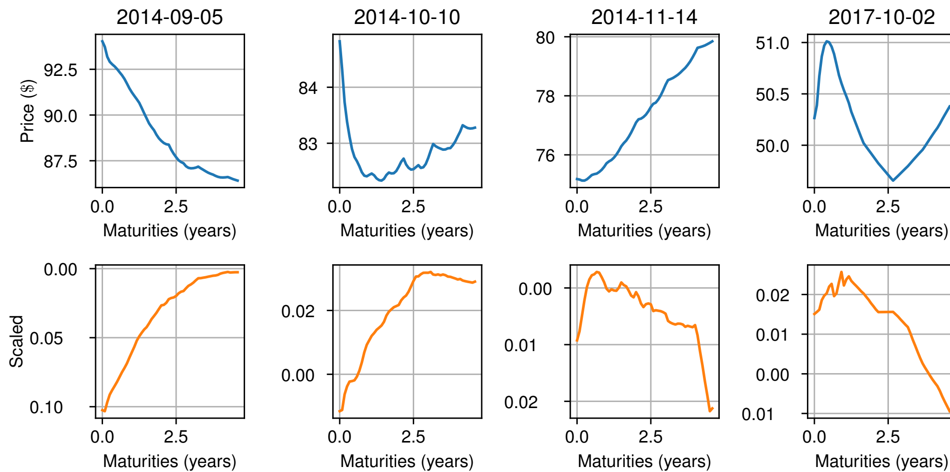


Figure 6.3: A selection of futures curves (first row) from the WTI NYMEX time series compared with their log-returns (second row). The log-returns are taken over the curves.

6.3.4 Normalisation and Standardisation over the Curves

In normalisation over the curves each curve is fitted such that the maximum value of the curve is scaled to take the value 1 and the minimum value of the curve is scaled to take the value 0. Therefore each curve retains its original shape. Only by retaining the original maximum and minimum values for each curve can the original time-series be recovered. Similar to normalisation over the curves, in standardisation over the curves each curve is fitted such that the mean of the points on the curve is 0 and the standard deviation is 1. Only by retaining the original mean and standard deviation of each curve can the original time-series be recovered. With both normalisation over the curves

and standardisation over the curves, in order to reconstruct the original time-series, for each curve two values need to saved. Therefore, applying these methods one has the scaled time-series, and a time-series of the minimum and maximum values, or a time-series of the mean and standard deviation of the original curve. This second time-series is required to reconstruct the scaled time-series. The values in the second time-series need to also be scaled such that it can be passed into a neural network. This again requires a scaling method, hence normalisation and standardisation over the curves are not viable options for scaling the data.

7 Dimension Reduction of Time-Series of Futures Curves

The standard, variational and adversarial autoencoder as well as PCA are dimension reduction methods that preserve the important information to the latent space. In this section, the performance of the different techniques is compared, and a comparison is made for how preprocessing methods (Section 6) affect the compression techniques.

The data from Table 2.1 is used to train and test the models. In Section 2.3 the training and test sets are described. The datasets *Brent ICE*, *Brent Dated*, *F35 ROT FB*, *GO01 ROT FB*, *HSFO380 GSP FC* and *JK ROT FB* are used as training data, and the *WTI NYMEX* dataset is used as test set. The neural networks are trained on the training sets. Once the neural networks are trained, the model is run on the test set.

PCA is compared against the autoencoder models covered in Section 5.1. For each model the parameters are specified. The effect of the size of the latent dimension on the reconstruction of the data is investigated. The other parameters are kept constant. The number and sizes of the hidden layers is chosen by some experimentation where the choice of hidden layers $(56, 40, 28, 12, 4)$ seems to perform fairly well. By this vector it is meant that the first layer is of size 56, the second of size 40, and so on. When the encoder and decoder both have this vector of layers $(56, 40, 28, 12, 4)$, it is meant that the decoder has layers $(4, 12, 28, 40, 56)$ in reverse order such as to create a two-sided funnel.

Model 7.1 (PCA)**.** *This is the PCA model (Section 3.3) with parameter* $k = 1, 2, 3, 4$. *The parameter* k *represents the dimension of the lower dimensional linear space that is projected on in the encoding.*

The PCA model is included as a benchmark model to compare the autoencoder models against. The autoencoders are non-linear, while PCA is linear, so in theory the autoencoders should be able to perform better, however in practice the results are not so clear.

Model 7.2 (Standard Autoencoder)**.** *This model is a standard autoencoder (Section 5.1.1) that encodes a vector of length 56 to a vector of length 1,2, 3 or 4, and decodes back to a vector of length 56. The parameters of the model are as follows:*

- *latent dimension:* $k = 1, 2, 3,$ *or* 4
- *Layers in encoder and decoder:* $(56, 40, 28, 12, 4, k)$
- *Activations: leakyReLU*(0.1)
- *Last activation: linear*
- *Loss: mean-squared error (MSE)*

In the standard autoencoder model (Model 7.2), the hidden layers correspond to a stack of fully connected layers for the encoder and the decoder, where each hidden layer has as activation function leakyReLU(0.1). The layers $(56, 40, 28, 12, 4, k)$ for the encoder and decoder represent the following feed-forward network:

$$56 \to 40 \to 28 \to 12 \to 4 \to k \to 4 \to 12 \to 28 \to 40 \to 56.$$

The final layer of size 56 has a linear activation function. The loss function used is the mean-squared error loss function.

Model 7.3 (Adversarial Autoencoder)**.** *This model is an adversarial autoencoder (Section 5.1.3) with the following properties:*

- *latent dimension:* $k = 1, 2, 3,$ *or* 4
- *Layers in encoder and decoder:* $(56, 40, 28, 12, 4, k)$
- *Layers in discriminator:* $(k, 2, 1)$
- *Activations: leakyReLU*(0.1)
- *Last activation generator: linear*
- *Last activation discriminator: sigmoid*

- *Generator loss: mean-squared error (MSE)*
- *Discriminator loss: binary cross-entropy.*

Model 7.4 (Variational Autoencoder)**.** *This model is a variational autoencoder (Section 5.1.2) with the following properties:*

- *latent dimension:* $k = 1, 2, 3,$ *or* 4
- *Layers in encoder and decoder:* $(56, 40, 28, 12, 4, k)$
- *Activations: leakyReLU*(0.1)
- *Last activation: linear*
- *Loss: mean-squared error (MSE).*

It is hoped that the variational and adversarial autoencoders perform better than the standard autoencoder because they allow for interpolation on the latent space which should mean that they are able to reconstruct unseen futures curves that look similar to curves from the training set.

7.1 Normalisation over the Tenors

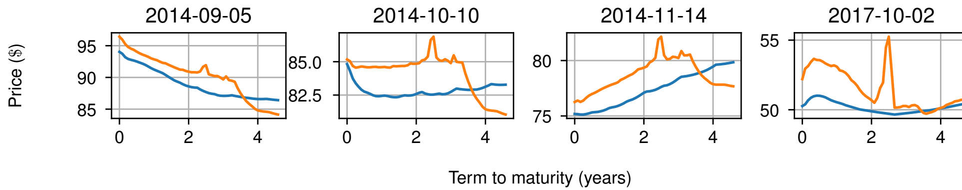
In Fig. 7.1 examples of futures curves versus their reconstruction under the methods PCA, standard autoencoder, AAE and VAE are examined, where the preprocessing method *normalisation over the tenors* is used. For each method the same four futures curves are used as examples. It is clear that all methods preserve the approximate price ranges of the original curves. For a lot of curves using *normalisation over the tenors* a jump occurs at just over the 2 years mark, however this jump does not seem to occur in Fig. 6.2. A possible reason behind this jump could be that this is being carried over from the datasets, since in the covariance of the log-returns Fig. 2.4 a spike can be seen at around 2.5 years.

In PCA (Fig. 7.1a), the price of the reconstructed curves seems consistently too high. The noise on the reconstructed curves is low compared to the autoencoders. The approximate shape is not properly reconstructed, and in the first three examples the curve dips around year 3. In AE (Fig. 7.1b) there is a lot of noise in the reconstruction, which can be addressed by applying the Savitzky-Golay filter (Definition 6.3). For simple curves such as in examples 1 and 3 it performs well, ignoring noise, but for some curves such as in example 2 the original shape cannot be seen in the reconstruction. In AAE (Fig. 7.1c) the reconstruction is similar as in Fig. 7.1b. In VAE (Fig. 7.1d) the performance is similar to that of PCA. Overall, from the examples, it looks like AE and AAE perform better than PCA and VAE.

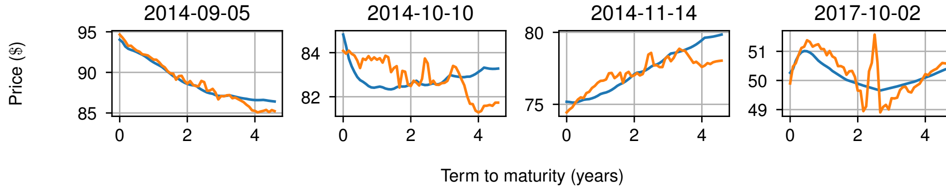
In Fig. 7.2 the encoded time-series of the test data is examined, normalised over the tenors and encoded with PCA, AE, VAE and AAE, with $k = 2$. Since $k = 2$ each curve is encoded into a point in $\mathbb{R}^2$, and the two lines in each sub-figure represent the two coordinates of the encoded curve for each day. For VAE (Fig. 7.2c) one dimension is barely used, since it is almost constant at 0. For AAE (Fig. 7.2d) there are similarities in the overall trends of both dimensions, suggesting that the encoding is not optimal. For PCA (Fig. 7.2a) and AE (Fig. 7.2b) is is more difficult to draw concrete conclusions out of the encoding, however they are better than that of Fig. 7.2c and Fig. 7.2d.

As described in Section 5.1.2 and Section 5.1.3 the variables in the latent space are by construction multivariate standard normal. The latent space for $k = 2$ is in $\mathbb{R}^2$. An element in the latent space can be randomly sampled by sampling from the bivariate standard normal distribution. This is shown in Fig. 7.3 where a 6 by 6 is made. The values on the axis come from the inverse cumulative density function of the standard normal distribution by inputting 6 values uniformly spaced over the interval $[0.05, 0.95]$. The interval $[0, 1]$ is not taken because the inverse cumulative density of 0 and 1 is $-\infty$ and ∞ respectively. For each pair of numbers (x, y) from the values along the x-axis and y-axis gives a sample from the bivariate standard normal distribution. The pair (x, y) is treated as a point from the latent space, and by decoding this point gives a curve in the original space. This curve is drawn in the grid. The grid shows how well the VAE and AAE manage to form the latent space such that sampling from the normal distribution and decoding this is similar to sampling directly from the data.

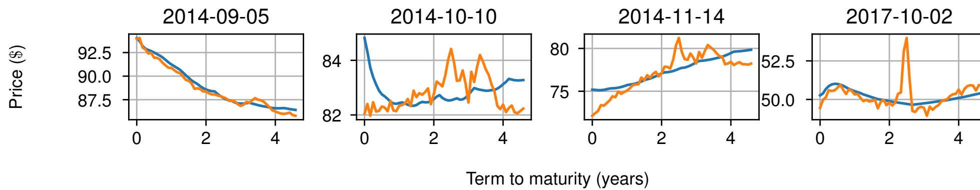
For the VAE, by Fig. 7.2c it is clear that one dimension is not used and this can again be seen here, where over one axis everything is constant. The spikes that occur in each reconstruction is ignored



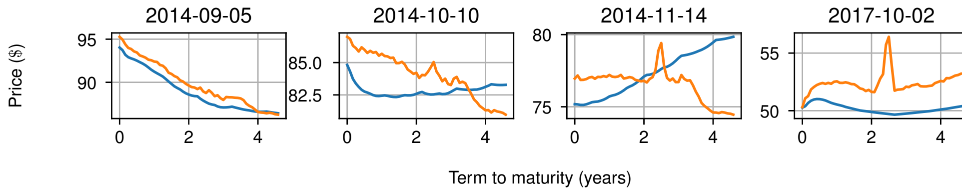
(a) Normalisation over the tenors and encoded with Model 7.1 PCA.



(b) Normalisation over the tenors and encoded with Model 7.2 standard autoencoder.



(c) Normalisation over the tenors and encoded with Model 7.3 AAE.

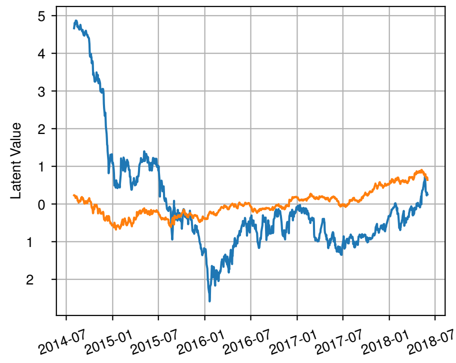


(d) Normalisation over the tenors and encoded with Model 7.4 VAE.

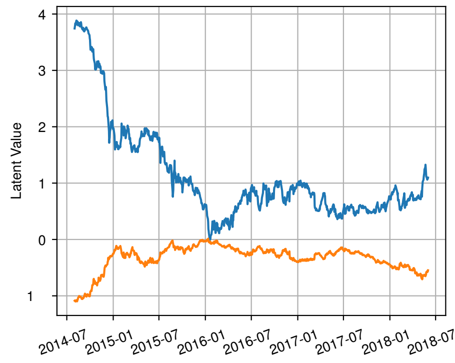
Figure 7.1: Comparison of the reconstruction of futures curves under *normalisation over the tenors* for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$. Blue is the original futures curve and orange is the reconstruction.

for the analysis. Over the other axis, fairly smooth changes in the curve shape is seen moving along the axis, however some of the futures curve shapes that appear often in the dataset are missing, and this shows that the VAE is not be able to reconstruct these well.

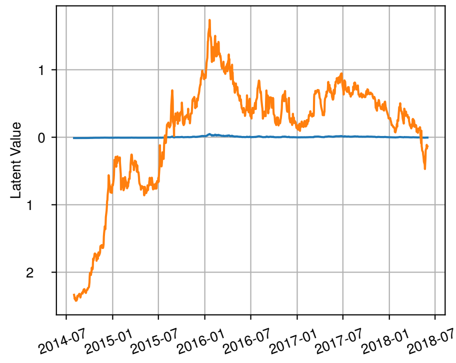
For the AAE, the curves show changes over both axes. In each diagonal opposing corner of the grid inversely shaped curves can be seen, in the one a downward sloping curve and in the other an upward sloping curve. This shows that the AAE has been much more successful at learning to encode the data compared to the VAE. There is a lot of noise in the curves and again a spike in each one.



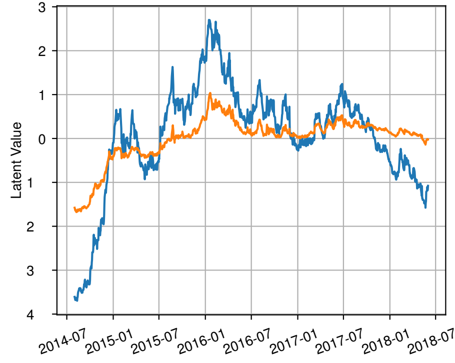
(a) Latent encoding of Model 7.1 PCA.



(b) Latent encoding of Model 7.2 AE.

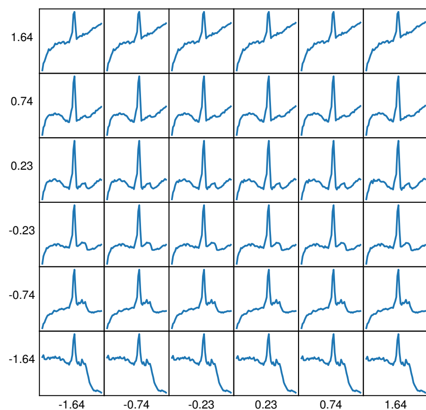


(c) Latent encoding of Model 7.4 VAE.

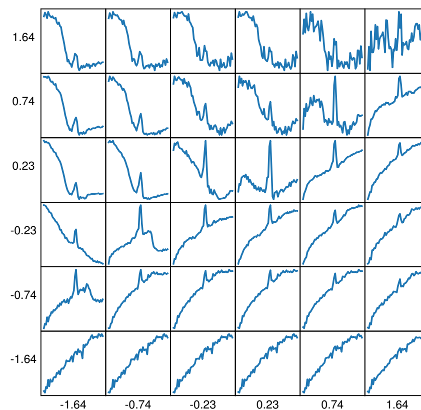


(d) Latent encoding of Model 7.3 AAE.

Figure 7.2: Comparison of the latent encodings for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$ with normalisation over the tenors. The blue and orange curves represent the first and second dimensions of the latent space.



(a) Variational Autoencoder (VAE)



(b) Adversarial Autoencoder (AAE).

Figure 7.3: Latent space of the Model 7.4 VAE and Model 7.3 AAE for $k = 2$ with normalisation over the tenors. The grid can be seen as an x, y graph where along each axis take 8 linear steps in the range [0.05, 0.095] and plug it into the inverse standard cumulative normal distribution function.

7.1.1 Results

Each of the models PCA, AE, AAE and VAE is trained on the data and SMAPE (Section 3.1) is used to gauge the performance of the model. For each reconstructed curve the SMAPE value is computed and from all the SMAPE values the mean and variance are computed. The training procedure is repeated 5 times because there is a certain randomness in the neural network methods. The initial values in the network are random and this gives variations in the final learned network. The calibration of the PCA method on the other hand is fully deterministic. In each method, the test data is used to determine the result. The test data is scaled by normalising over the tenors, encoded, decoded and then rescaled.

Method	Parameters	SMAPE, mean	SMAPE, std
Model 7.1 PCA	$k = 1$	0.044	0.019
Model 7.2 AE	$\mathbf{k = 1}$	**0.033**	**0.020**
Model 7.3 AAE	$k = 1$	0.033	0.021
Model 7.4 VAE	$k = 1$	0.051	0.032
Model 7.1 PCA	$k = 2$	0.027	0.0028
Model 7.2 AE	$\mathbf{k = 2}$	**0.011**	**0.0084**
Model 7.3 AAE	$k = 2$	0.025	0.017
Model 7.4 VAE	$k = 2$	0.054	0.035
Model 7.1 PCA	$k = 3$	0.026	0.0036
Model 7.2 AE	$\mathbf{k = 3}$	**0.0085**	**0.0063**
Model 7.3 AAE	$k = 3$	0.015	0.013
Model 7.4 VAE	$k = 3$	0.053	0.032
Model 7.1 PCA	$k = 4$	0.026	0.0037
Model 7.2 AE	$\mathbf{k = 4}$	**0.0071**	**0.0030**
Model 7.3 AAE	$k = 4$	0.011	0.0045
Model 7.4 VAE	$k = 4$	0.054	0.033

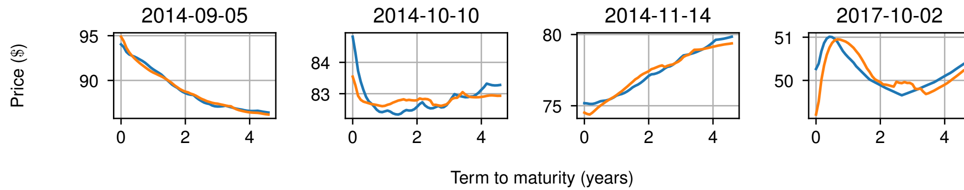
Table 7.1: Results for the reconstruction error for various methods with different latent dimension k. The data preprocessing used is normalisation over the tenors. The results are rounded to 2 significant figures. The SMAPE reconstruction error is applied to each individual input and output curve. The mean and variance are of all the reconstruction errors from the test set.

The results in Table 7.1 are examined. From best to worst: AE, AAE, PCA, VAE. For each k for the model that performs the best, the standard deviation has approximately the same order as the mean. By increasing k the mean and standard deviation generally become smaller. It is clear that the variational autoencoder is not suitable for this task. It also does not improve for larger k. It is possible that the variational autoencoder applies a too large a restriction on the latent space, whereby not all curves can have an accurate representation on the latent space. It is also interesting to see that the standard autoencoder performs better than the adversarial autoencoder. The performance decrease of using the AAE and not the AE does however come with the advantage of being able to directly sample form the latent space.

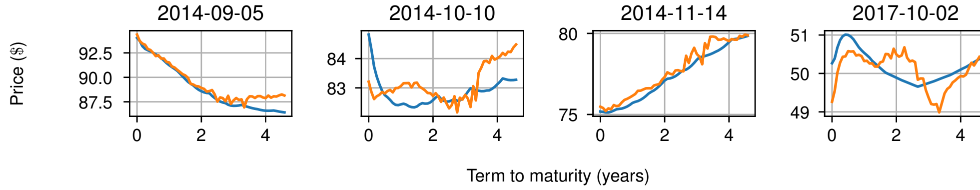
7.2 Standardisation over the Tenors

In this section a similar analysis as in Section 7.1 is made, with the important change of using *standardisation over the tenors* instead of *normalisation over the tenors*. In Fig. 7.4 a similar comparison as in Fig. 7.1 is made. For each model, the prices of the reconstructed curves lie in the same range as the original curves.

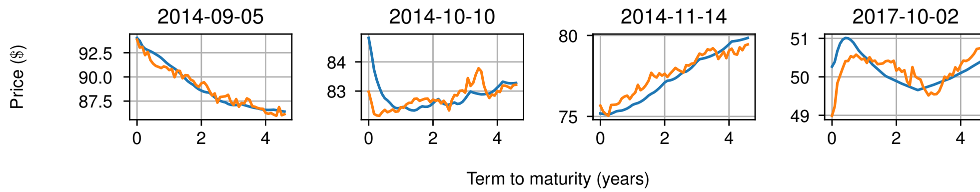
For PCA in Fig. 7.4a a remarkably good reconstruction with little noise is achieved. The AE and AAE models do quite well, however they have a lot of noise and have some difficulties reconstructing the exact shape of the last example and cannot reconstruct the second example too well either. The VAE Fig. 7.4d has difficulties reconstructing the shape of all the examples and it is not likely to perform well overall. In comparison with Section 7.1, no spike is seen at the 2.5-year mark, and this hints that Section 7.2 coupled with dimension reduction techniques is more robust against errors in the training data.



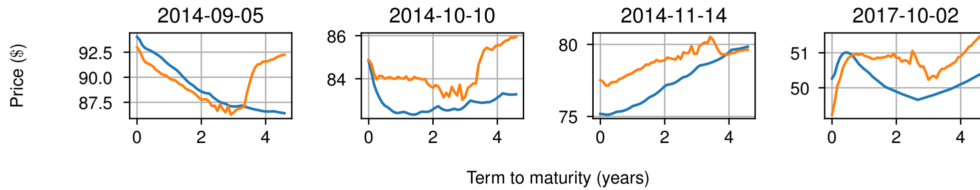
(a) Standardisation over the tenors and encoded with Model 7.1 PCA.



(b) Standardisation over the tenors and encoded with Model 7.2 standard autoencoder.



(c) Standardisation over the tenors and encoded with Model 7.3 AAE.

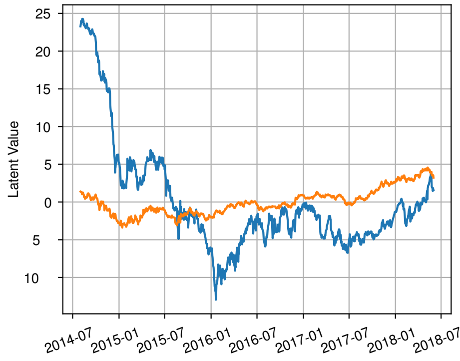


(d) Standardisation over the tenors and encoded with Model 7.4 VAE.

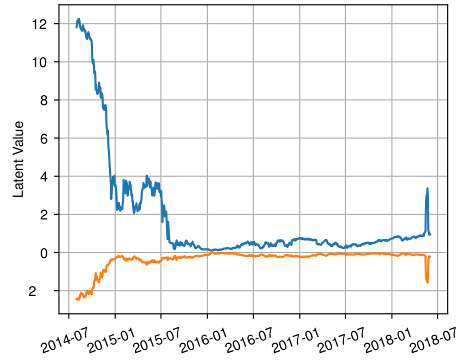
Figure 7.4: Comparison of the reconstruction of futures curves under *standardisation over the tenors* for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$.

In Fig. 7.5, the latent encodings of the test data are examined. For the VAE Fig. 7.5c both dimensions are used this time, however there are some similarities in the two curves. This shows that the VAE does not optimally use both available dimensions.

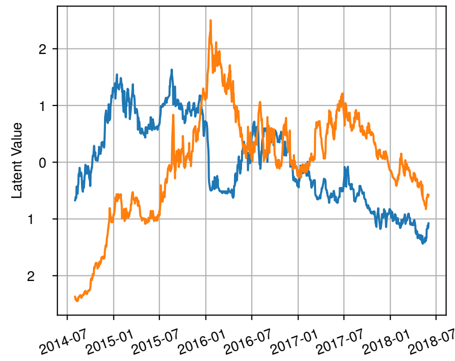
In Fig. 7.6 the VAE performs much better this time than in Fig. 7.3. For both the VAE and the AAE, samples from the decoder can in some cases look quite noisy. In both cases, the latent space maps to a variety of different shaped curves, and it is difficult to tell the performance solely by looking at these grids.



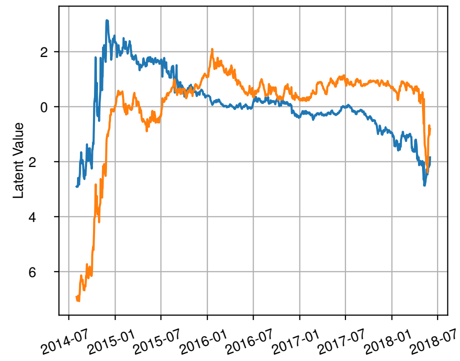
(a) Latent encoding of Model 7.1 PCA.



(b) Latent encoding of Model 7.2 AE.

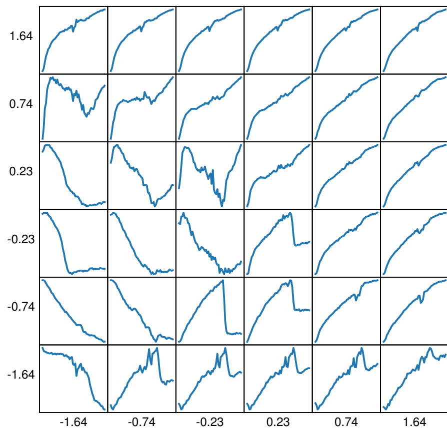


(c) Latent encoding of Model 7.4 VAE.

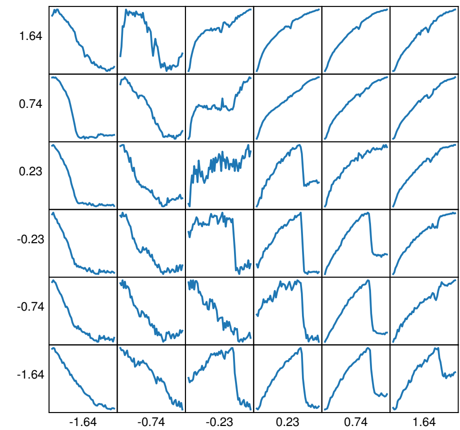


(d) Latent encoding of Model 7.3 AAE.

Figure 7.5: Comparison of the latent encodings for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$ with standardisation over the tenors.



(a) Variational Autoencoder (VAE)



(b) Adversarial Autoencoder (AAE).

Figure 7.6: Latent space of the Model 7.4 VAE and Model 7.3 AAE for $k = 2$ with standardisation over the tenors. See Fig. 7.3 for explanation of what the grid represents.

7.2.1 Results

The results of dimension reduction is shown, using the models PCA, AE, AAE and VAE, with the pre-processing method *standardisation over the tenors*. The dimension reduction techniques are applied and the reconstruction errors are computed using SMAPE. For $k = 1$ all four models perform similarly. For $k = 2, 3, 4$ PCA performs the best followed closely by AE and AAE. The VAE does not perform well on $k = 2, 3, 4$ compared to the other models. There are hardly any performance gains for $k = 3$ or 4 compared to $k = 2$.

The PCA model outperforms the neural networks for $k > 1$, and this was also hinted at earlier when examining Fig. 7.4. Furthermore, for each model, applying standardisation over the tenors gives better performance compared to normalisation over the tenors. A reason why is that in comparison to normalisation over the tenors, there are no jumps just past the two year mark, which would effect the SMAPE score. These jumps may come from errors in the data causing outliers. The results suggest that standardisation over the tenors may be more robust against these errors.

Method	Parameters	SMAPE, mean	SMAPE, std
Model 7.1 PCA	$k = 1$	0.032	0.023
Model 7.2 AE	$k = 1$	0.025	0.016
Model 7.3 AAE	$k = 1$	0.025	0.016
Model 7.4 VAE	**k = 1**	**0.024**	**0.013**
Model 7.1 PCA	**k = 2**	**0.0043**	**0.0026**
Model 7.2 AE	$k = 2$	0.0073	0.0039
Model 7.3 AAE	$k = 2$	0.0086	0.0061
Model 7.4 VAE	$k = 2$	0.015	0.0085
Model 7.1 PCA	**k = 3**	**0.0032**	**0.0020**
Model 7.2 AE	$k = 3$	0.0065	0.0033
Model 7.3 AAE	$k = 3$	0.0065	0.0033
Model 7.4 VAE	$k = 3$	0.016	0.010
Model 7.1 PCA	**k = 4**	**0.0030**	**0.0019**
Model 7.2 AE	$k = 4$	0.0065	0.0036
Model 7.3 AAE	$k = 4$	0.0083	0.0052
Model 7.4 VAE	$k = 4$	0.016	0.0096

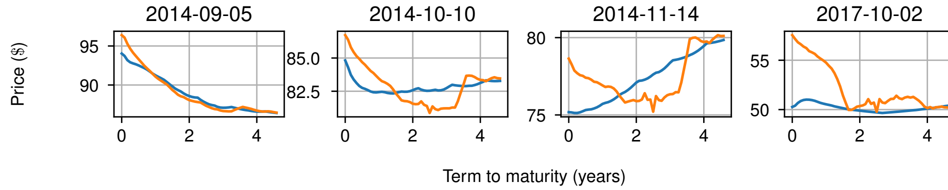
Table 7.2: Similar to Table 7.1 except now using the preprocessing method: standardisation over tenors.

7.3 Log-Returns over the Tenors

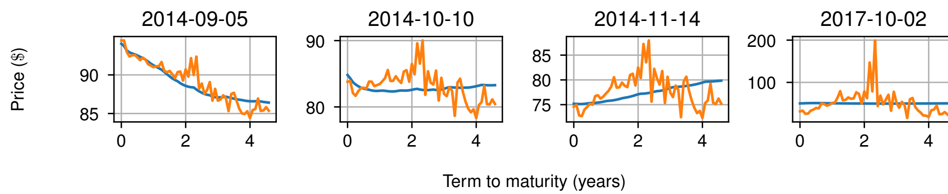
A similar analysis as in Section 7.1 and in Section 7.2 is made but now using log-returns over the tenors. From the figures below it is clear that using log-returns over the tenors for encoding futures curves delivers unusable results.

The reconstructed curves for PCA and for the autoencoders AE, AAE and VAE in Fig. 7.7 are in most cases not in the same price range as the original curves. The reconstructed curves contain an extremely large amount of noise, from which it is impossible to see the shape of the original curve. This indicates that using log-returns over the tenors for encoding futures curves is not possible. There is not so much reason to consider the latent encodings, however for completeness they are included in Fig. 7.8 and Fig. 7.9. The latent encoding of the test data is shown in Fig. 7.8 and it looks very much like noise. This is very different to what is seen for normalisation over the tenors and standardisation over the tenors in Fig. 7.2 and Fig. 7.5 respectively. The grid showing the representation of the latent space in Fig. 7.9 shows only one shape of curve, meaning that the VAE and AAE have only learnt to encode one curve shape. This is not useful for encoding futures curves.

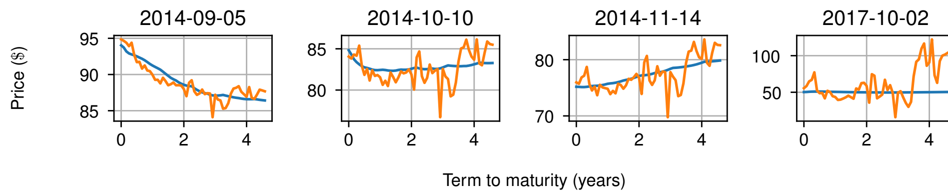
The latent space of the VAE and AAE is shown in Fig. 7.8 and it is a constant grid, showing that neither have been able to learn the representation of the data. It is clear that using *log-returns over the tenors* is an ineffective pre-processing method for learning futures curves.



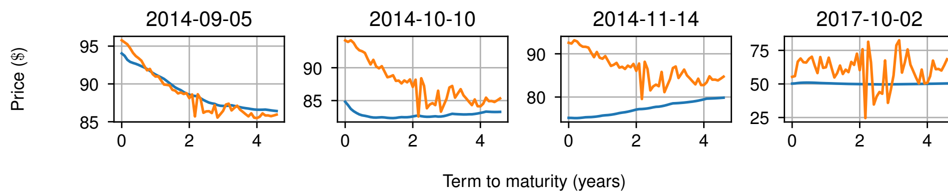
(a) Log-returns over the tenors and encoded with Model 7.1 PCA.



(b) Log-returns over the tenors and encoded with Model 7.2 standard autoencoder.

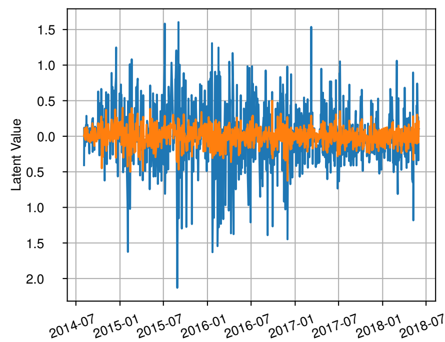


(c) Log-returns over the tenors and encoded with Model 7.3 AAE.

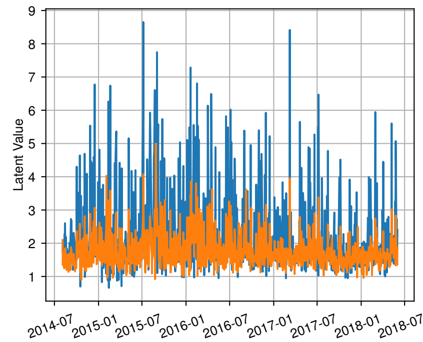


(d) Log-returns over the tenors and encoded with Model 7.4 VAE.

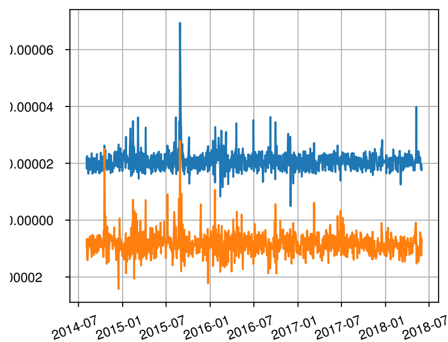
Figure 7.7: Comparison of the reconstruction of futures curves under *log-returns over the tenors* for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$.



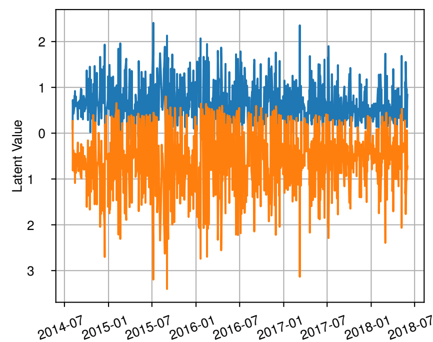
(a) Latent encoding of Model 7.1 PCA.



(b) Latent encoding of Model 7.2 AE.

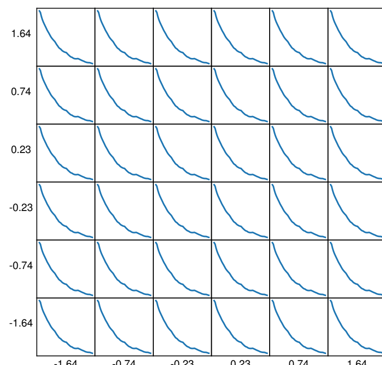


(c) Latent encoding of Model 7.4 VAE.

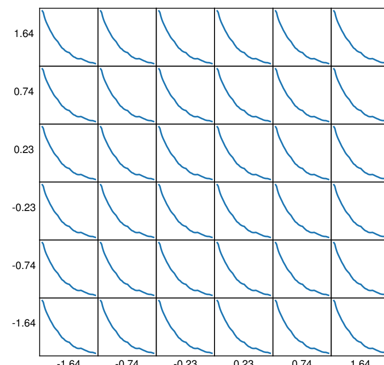


(d) Latent encoding of Model 7.3 AAE.

Figure 7.8: Comparison of the latent encodings of the test data for Model 7.1 PCA, Model 7.2 AE, Model 7.4 VAE, and Model 7.3 AAE for $k = 2$ with Log-returns over the tenors.



(a) Variational Autoencoder (VAE)



(b) Adversarial Autoencoder (AAE)

Figure 7.9: Latent space of the Model 7.4 VAE and Model 7.3 AAE for $k = 2$ with log-returns over the tenors. See Fig. 7.3 for explanation of what the grid represents.

7.3.1 Results

The SMAPE scores of the test data when using the preprocessing method *log-returns over the tenors* with the dimension reduction models PCA, AE, AAE and VAE, is examined in Table 7.3. The reconstruction errors are much larger than when using normalisation or standardisation over the tenors. From Fig. 7.7 it can be seen that AE, AAE, and VAE produce extremely noisy curves, and this is also shown in the results where these models perform much worse than PCA by an order of 10. To encode futures curves it would therefore not be wise to use log-returns as the preprocessing method.

Method	Parameters	SMAPE, mean	SMAPE, std
Model 7.1 PCA	$k = 1$	0.086	0.032
Model 7.2 AE	$k = 1$	0.50	0.28
Model 7.3 AAE	$k = 1$	0.22	0.12
Model 7.4 VAE	$k = 1$	0.40	0.19
Model 7.1 PCA	$k = 2$	0.030	0.01
Model 7.2 AE	$k = 2$	0.18	0.11
Model 7.3 AAE	$k = 2$	0.15	0.087
Model 7.4 VAE	$k = 2$	0.42	0.27
Model 7.1 PCA	$k = 3$	0.033	0.018
Model 7.2 AE	$k = 3$	0.35	0.19
Model 7.3 AAE	$k = 3$	0.13	0.075
Model 7.4 VAE	$k = 3$	0.37	0.20
Model 7.1 PCA	$k = 4$	0.024	0.012
Model 7.2 AE	$k = 4$	0.27	0.17
Model 7.3 AAE	$k = 4$	0.15	0.13
Model 7.4 VAE	$k = 4$	0.35	0.21

Table 7.3: Data Preparation: Log-returns over the tenors.

7.4 Dimension Reduction over Windows of Futures Curves

Previously in this chapter the dimension reduction techniques have been constructed such that they encode a futures curve, represented by a vector of length 56, to an encoded curve of length 1, 2, 3, or 4. Consider an autoencoder that encodes a sequence of futures curves into a sequence of encoded futures curves. Such a sequence is called a **window**. The motivation behind this is that over a time-series of futures curves the change in curve shape over time is usually smooth, so there are similarities between adjacent curves. An autoencoder should hopefully be able to pick up on these similarities and thereby lead to a better encoding compared to encoding single futures curves.

The autoencoder takes a window of size 20. The input array for the autoencoder is $(20, 56)$, this is flattened to a vector of length 1120, and the same autoencoder model as Model 7.2 is applied.

Method	Parameters	SMAPE, mean	SMAPE, std
standardisation over tenors	$k = 2$	0.057	0.046
normalisation over tenors	$k = 2$	0.014	0.0072
log-returns over tenors	$k = 2$	0.030	0.087

Table 7.4: Dimension reduction over windows of futures curves using a dense autoencoder.

7.5 Semi-Invertible Preprocessing Methods

The semi-invertible preprocessing methods discussed were: normalisation over the curves, standardisation over the curves and log-returns over the curves. These methods are not further examined because after some preliminary testing they were found perform much worse compared to the invertible preprocessing methods. The main problem with these methods is that the scaled data is not enough to reconstruct the original data. The additional data required to rescale the data must also be pre-processed. This leads to a complex setup to preprocess data.

7.6 Conclusion

The results for the three data preprocessing techniques: *normalisation over the tenors*, *standardisation over the tenors*, and *log-returns over the tenors*, followed by the dimension reduction techniques: PCA, AE, AAE, and VAE, are examined. On the test set, *standardisation over the tenors* performs in all cases better than *normalisation over the tenors* and *log-returns over the tenors*. The *log-returns over the tenors* give dramatically poor results and this is not further used for dimension reduction. In Table 7.5 a comparison of the best results for each k from Tables 7.1 and 7.2 is made. For $k = 2, 3, 4$, standardisation performs more than twice as good as normalisation. One factor affecting the results maybe be the peaks appearing in the data for the normalisation method, which would negatively affect the SMAPE.

Method	Parameters	SMAPE, mean	SMAPE, std
Model 7.2 AE	$k = 1$	0.033	0.020
Standardisation, Model 7.4 VAE	**k = 1**	**0.024**	**0.013**
Model 7.2 AE	$k = 2$	0.011	0.0084
Standardisation, Model 7.1 PCA	**k = 2**	**0.0043**	**0.0026**
Model 7.2 AE	$k = 3$	0.0085	0.0063
Standardisation, Model 7.1 PCA	**k = 3**	**0.0032**	**0.0020**
Normalisation, Model 7.2 AE	$k = 4$	0.0071	0.0030
Standardisation, Model 7.1 PCA	**k = 4**	**0.0030**	**0.0019**

Table 7.5: Comparison of best results under *standardisation over the tenors* and *normalisation over the tenors*.

Considering *standardisation over the tenors*, PCA gives the smallest reconstruction error, followed by Model 7.2 AE and AAE. This is a surprising result since the PCA method is linear, while the Autoencoder used is non-linear. The advantage of using the AAE instead of PCA, sacrificing the performance of PCA for the slightly worse performance of AAE, is that with AAE it is possible to sample from a normal distribution and feed this into the decoder to sample curves. This cannot be done with PCA as it is not a generative method and there is no smoothness property forced on the latent space.

8 Generative Adversarial Networks for Time-Series Simulation

In this section GANs are examined for their use in time-series simulation. Simulations of time-series of futures curves are made using GAN methods and a comparison is made to the Andersen Markov model (Section 3.2) which acts as the benchmark model. In Section 8.1 a simplified case is considered where a one-dimensional time series is simulated using GANs and this provides the motivation for simulating a time-series of futures curves (Section 8.2) using GANs.

8.1 Spot Price Time-Series Simulation

From the datasets of time-series futures curves, the short ends of the curves are taken to give a one-dimensional time-series, that is for each futures curve represented by a vector, the first element in the vector is the spot price and this is referred to as the short end. Therefore for each date in the time-series there is one scalar value representing the spot price.

In Model 8.1 a model to simulate time-series data is set up. The conditional GAN (cGAN Section 5.2.2) is used in this model. The time-series is broken down into intervals of 42 steps, which are referred to as time-series **windows** and these windows are used as samples for the network. The conditional GAN is trained such that given a window of 42 steps as the conditional data, it should generate the window of 42 steps that follows. Once the cGAN is trained, condition on the first 42 steps of the test set, to simulate the next 42 steps. The simulated window is compared to the true window from the data using SMAPE to give a measure of the performance of the model.

To prepare the data for the cGAN two preprocessing methods are applied. First, standardisation is applied such that the values fitted to the right range and second log-returns are taken such that the cGAN is given relative prices. Only standardisation over the tenors is considered and normalisation over the tenors is no longer used because from Section 7.2 standardisation over the tenors performs decisively better.

Model 8.1 (Spot Price Time-Series Simulation)**.** *Consider the following model:*

1. *Preprocessing with standardisation over the tenors*
2. *Preprocessing with log-returns over the tenors*
3. *cGAN trained on windows of length* 42 *of the log-returns.*
4. *On the trained cGAN model input the first* 42 *steps of the test set. The cGAN returns a matrix* $(100, 42)$*, which represents a 100 simulations each of* 42 *steps.*
5. *Reverse preprocessing methods on the cGAN output.*

In Fig. 8.1 an example of the output of the trained conditional GAN is shown. The simulated grey paths move similarly to the red (real) path. The SMAPE is used to measure how well the simulations compare to the real path.

If the length of the simulation needs to be increased, either the window size that the model is trained on can be increased, or the trained cGAN can be recursively used. More clearly, by recursively using the cGAN, the simulated time-series is fed back into the cGAN to get a simulation of the next 42 days conditioned on the previous 42 days.

Method	SMAPE, mean	SMAPE, std
GAN	0.14	0.057
WGAN	0.14	0.036

Table 8.1: Results for Model 8.1 with data preprocessing standardisation over tenors followed by log-returns over tenors. No dimension reduction is applied.

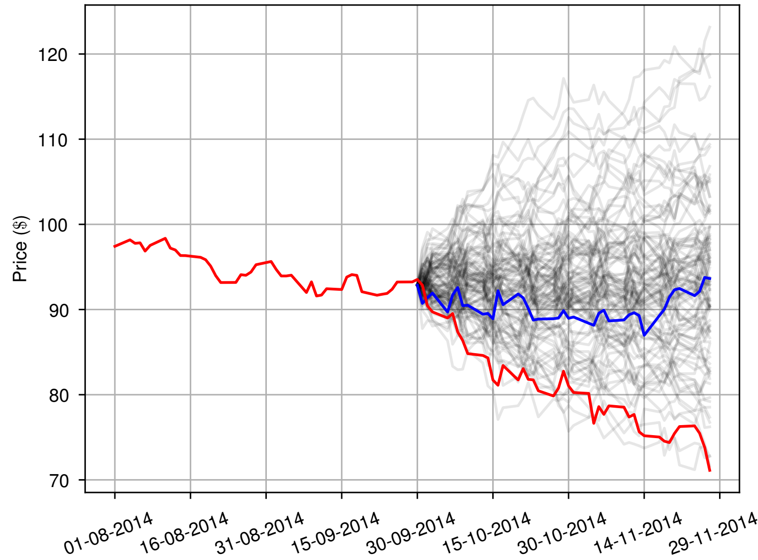


Figure 8.1: GAN conditioned on windows of length 42. The red line shows the real time-series, the grey lines show 100 simulated time-series for the following 42 time-steps, and the blue is one such example of the 100 simulated time-series paths.

8.2 Time-Series of Futures Curves Simulation

A time-series of futures curves can be represented by a matrix (or array) where the columns represent tenors and the rows represent dates, with a row being a futures-curve (Definition 2.3). In Section 8.1, the time series was given as a vector. To apply Model 8.1 but to a matrix, the matrix can be flattened into a vector, more clearly, if the matrix is of size $m \times n$ this can be reshaped into a vector of length $m \cdot n$. For time-series data with the number of tenors $n > 4$ it is found that the GAN model trains very slowly. Since the time-series data has $n = 56$ tenors it is not possible to use the GAN model directly. Instead the dimension reduction techniques covered in Section 7 are applied to encode the data to a smaller dimension and feed the encoded data into the GAN. The GAN model is trained on the encoded data, and simulates according to the encoded data.

In Section 7.6 it is found that the best dimension reduction models are PCA, AE and AAE with latent dimension $k = 2$ and using the preprocessing method *standardisation over the tenors* on the data. Therefore in this section, the methods used are restricted to these.

Consider Model 8.2 given as the extension to Model 8.1 with the addition of using dimension reduction.

Model 8.2 (Time-Series Futures Curves Simulation)**.** *This model extends Model 8.1 with the addition of dimension reduction. The model is set out in the steps below. The training and test datasets are represented as matrices of size $m \times n$, where m the number of time-steps and n is the number of tenors. The number of tenors is $n = 56$. The preprocessing methods are applied to each dataset individually.*

1. *Apply preprocessing with* ***standardisation over the tenors*** *to the data.*
2. *Train one of the dimension reduction models, PCA, AE or AAE, and apply to the test set.*
3. *Apply reprocessing with* ***log-returns over the tenors*** *to the encoded data.*
4. *Train the* ***conditional GAN*** *on windows of length 42 of the log-returns of the encoded training sets. Take the first window of the log-returns of the encoded test set and use the conditional GAN on this set.*

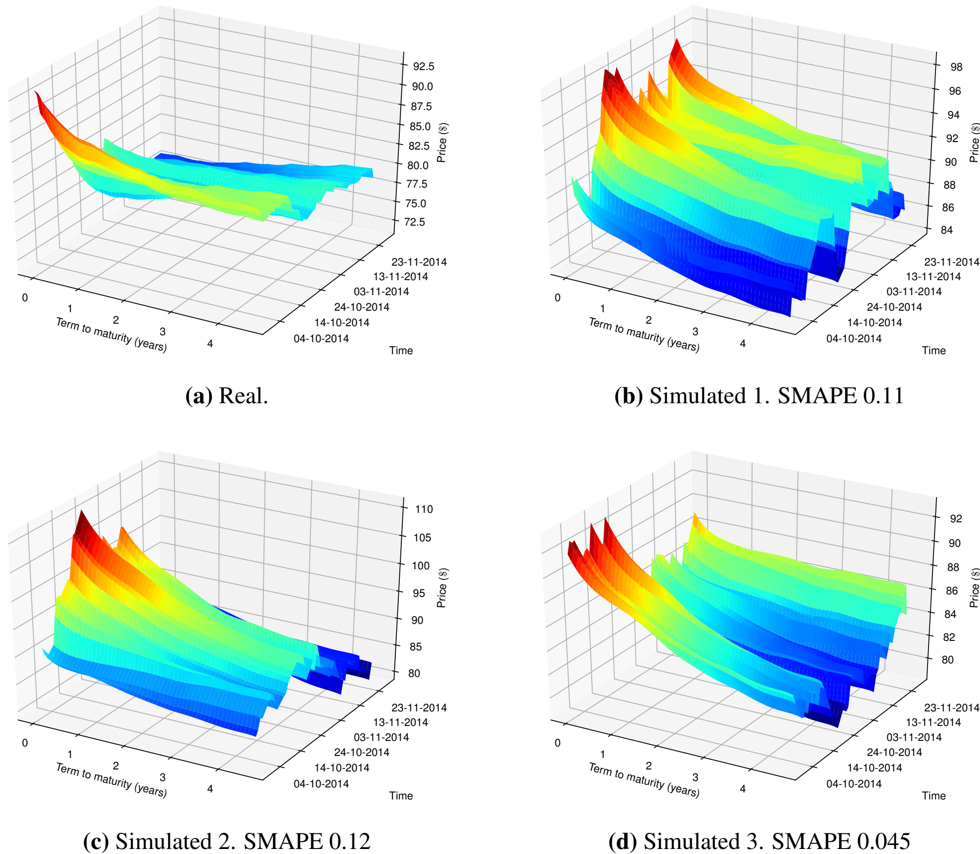
On the generated data take the inverse operations:

1. *Undo log-returns over the tenors by providing the first futures curve of each set.*
2. *Decode using the used dimension reduction method.*
3. *Undo the standardisation over the tenors for each set.*

Given the first window of length 42 *from the test set, the next window of* 42 *futures curves can be simulated. This can be compared to the real futures-curves and the performance can be measured using SMAPE.*

In Definition 2.5 the parameter k represents the multiplication factor and the parameter c represents the addition factor. By examining the ranges of the encoded log-returns when applying Model 8.2 the parameter values $k = 10$ and $c = 25$ are chosen. This choice in parameters leads to a good learning ability of the conditional GAN. The choice of c is to make the values positive such that the logarithm can be taken. The multiplication factor is chosen such that the log-returns are approximately in the range $[-1, 1]$.

In Fig. 8.2 3d plots of futures curves time-series of length 42 are shown. The plot in Fig. 8.2a is of the second set of 42 futures-curves from the test set. This is compared to 3 simulations made by the cGAN from Model 8.2. In general, observe that the simulations are more volatile than the real time-series by examining the plots.



(a) Real.

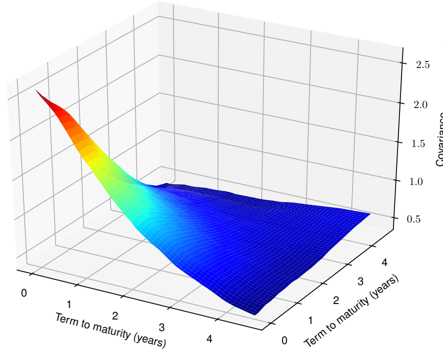
(b) Simulated 1. SMAPE 0.11

(c) Simulated 2. SMAPE 0.12

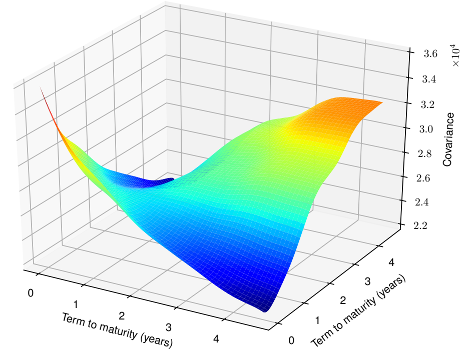
(d) Simulated 3. SMAPE 0.045

Figure 8.2: Comparison of the real data compared to 3 simulations using PCA with GAN.

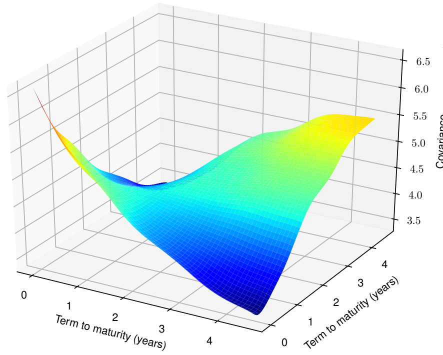
The results for combinations of dimension reduction methods PCA, AE and AAE coupled with generative models GAN, GAN-CONV and WGAN are shown in Table 8.2. All of the combinations of methods give results quite close to each other. Overall PCA performs the worst, since although the mean is roughly the same in the case of the GAN and WGAN, the standard deviation is almost double. The GAN-CONV with PCA has a very low standard deviation, however the mean is the highest for the three methods under PCA showing that PCA GAN-CONV performs consistently badly.



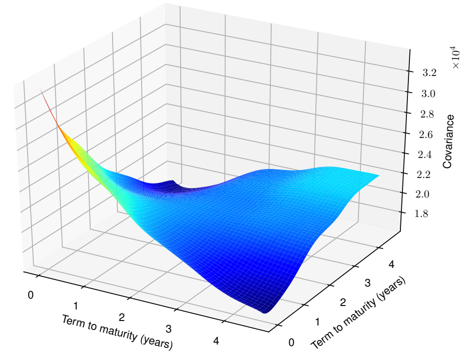
(a) Covariance of real data.



(b) Simulated 1, covariance. SMAPE 0.11



(c) Simulated 2, covariance. SMAPE 0.12



(d) Simulated 3, covariance. SMAPE 0.045

Figure 8.3: Comparison of the covariance of the log-returns of the real data and simulations using PCA with GAN.

For the three GAN models under AE, the means are very close to each other, and the standard deviations are the lowest compared to PCA and AAE. For the GANs under AAE, the GAN and WGAN perform similarly to in the AE case. The GAN-CONV again performs poorly and worse than in the PCA case.

In general the GANs under AE perform better than under PCA or AAE and the best performing model is AE GAN-CONV. It is surprising that the combination of AE with GAN-CONV performs well because the combinations of PCA with GAN-CONV and AAE with GAN-CONV perform the worst. This conclusion comes from the results in Table 8.2 and is based only on empirical evidence.

Method	SMAPE, mean	SMAPE, std
PCA GAN	0.090	0.045
PCA GAN-CONV	0.13	0.0092
PCA WGAN	0.084	0.042
AE GAN	0.085	0.028
AE GAN-CONV	**0.083**	**0.01**
AE WGAN	0.078	0.025
AAE GAN	0.082	0.033
AAE GAN-CONV	0.15	0.024
AAE WGAN	0.074	0.029

Table 8.2: Data Preparation: 1. standardisation over tenors. 2. log-returns over tenors. Dimension reduction applied with $k = 2$.

In order to generate larger simulations there are two possibilities with Model 8.2. The first is that the model can be trained on larger sized windows. The disadvantage of this is that this reduced the amount of training data available because for a larger window size the training data splits into a smaller number of such windows. The second method is to repeatedly apply Model 8.2. In this procedure the simulated output of the model is fed back into the model as the window that should be conditioned on. This can be repeated multiple times to obtain a simulation of length $r \cdot 42$ for $r \in \mathbf{N}$ applications of the model.

The benchmark model in this paper is the Andersen Markov model (AMM), see Section 3.2. The AMM is calibrated on the first 42 curves of the test set, to simulate the next 42 curves, and repeating this 100 times the SMAPE mean of 0.26 is found with a standard deviation of 0.15. Comparing this with the results of the GAN models in Table 8.2, all GAN models perform better than the AMM, with AE GAN-CONV having a mean SMAPE 3 times smaller than the mean SMAPE of AMM, and standard deviation 1/10th the size.

The simulations produced by the AMM in Fig. 3.1 can be compared with those produced by the GAN model Fig. 8.2. In the GAN simulations, the curves look much more like the real curves in that they are not perfectly smooth and are not only increasing or decreasing as in the AMM simulations. As discussed in Section 3.2, this is a constraint of the AMM.

The covariance log-returns of the real and simulated data from Fig. 8.2 are shown in Fig. 8.3. This can be compared with the covariance of the log-returns of the entire test set in Section 2.4. It is clear that by looking at the covariance of log-returns of a section of the test data as in Fig. 8.3a, that the plot is a lot less smooth than over more steps. Figs. 8.3b to 8.3d show that the covariance plots have a more complex nature than the approximately decreasing surface shape in Fig. 8.3a. This could be due to training the GAN on too small windows of time-series data.

9 Time-Series Simulation with RNNs

Sequence to sequence models have been used for language translation models to translate sentences from one language to another, for abstract text summarisation and for text generation, to name a few examples. Recently sequence to sequence models have been used for time-series simulation [Tonin 2019], motivated by their ability for good pattern extraction from the input space where the input can be long sequences. The memory properties of RNN networks (see Section 5.4) could be useful in analysing time-series data, since this can give the model the ability to recognise patterns, which would be useful for simulation.

The examples in Tonin [2019] were reproducible when using GRU cells to make predictions on sequences of sine curves with added noise and performed well at predicting the next 42 steps. This is a many-to-many RNN model that is composed of an encoder and a decoder. The encoder and decoder in this case do not form an autoencoder as previously seen, in this case the output of the decoder is not trained the resemble the input of the encoder. Instead they are used to split the time-series prediction problem into two parts. The task for the encoder is to capture the current state of the system. Using the current state of the system, the task for the decoder is to make a simulation of the time-series in the future. The encoder and decoder are RNNs made up of a number of GRU cells. LSTMs could be used instead as well. Two GRU cells with output space of size 35 for both the encoder and decoder is used. The input sequence length for the encoder is 42, the output sequence length of the decoder is also 42. An example of the RNN simulating a sine curve with noise can be seen in Fig. 9.1. The RNN manages to predict the sine curve frequency reasonably correctly, but it does not simulate randomness in the amplitude.

Using the same model but training on time-series of short end of futures curves, the same as in Section 8.1 with data pre-processing standardisation followed by taking log-returns as in Model 8.1, is wholly ineffective and the prediction from the RNN is a constant time-series set to 0. This can be seen in Fig. 9.2 where the time-series shows the true log-returns from the data and the simulated log-returns from application of the model. A reason for the non-working model may be that the parameters are not configured correctly for this different data, or alternatively the RNN may be incapable of finding patterns in the data. The patterns behind oil prices are considerably more complex than sine curves with added noise.

Comparisons are made between classical and machine learning algorithms for time-series simulation in Brownlee [2018b] and the author concludes that RNNs and LSTMs perform poorly against classical methods on univariate datasets. The taxi company Uber has used LSTMs with success to forecast user demand of their services [Laptev et al. 2017]. It may be that RNN models require expert tuning to perform well.

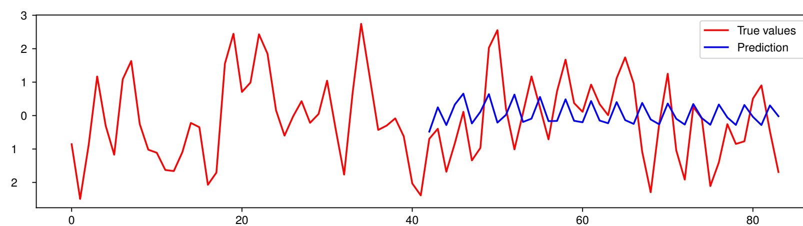


Figure 9.1: Network of GRU cells predicting the movement of a time-series of a sine curve with added noise. The example is from Kompella [2018].

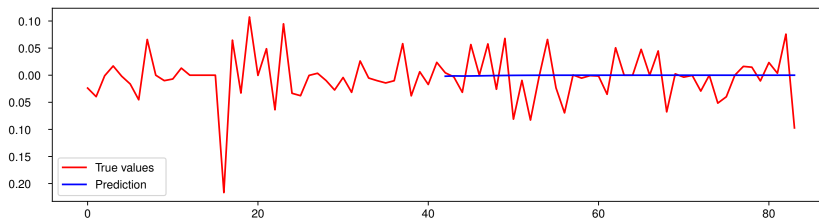


Figure 9.2: Network of GRU cells predicting the log-returns of a time-series on test data.

10 Simulation of Time Series Data with Missing Values

Imputation is filling in missing data with suitable values based on available information. Imputation methods can be categorised as discriminative or generative. The focus is on generative methods based on deep learning. In the literature, the majority of methods are either extensions of the autoencoder model or of the generative adversarial model. The Generative Adversarial Imputation Network (GAIN) [Yoon et al. 2018] is examined in this section. The GAIN model is an extension of the generative adversarial network.

The generator is given a sample of corrupted data knowing which values are corrupted, and a random noise vector. The generator outputs a completed vector conditioned on this information.

The discriminator receives a complete vector and attempts to determine which values are real and which values have been generated by the generator. Additional information is provided to the discriminator in the form of a hint vector, that provides partial information in the form of the probability that a value has been generated. This hint vector is used to steer the focus of the discriminator and it ensures that the generator learns the true data distribution.

One advantage of the GAIN model is that it can be trained on data that is already missing values, and does not require complete data unlike other imputation models such as the denoising autoencoder (DAE)

Consider the space $\mathcal{X} = \mathcal{X}_1 \times \ldots \times \mathcal{X}_d$. Let the true data distribution be $\mathbf{x} \sim p(\mathbf{x})$ with $\mathbf{x} = (x_1, \ldots, x_d)$ over this space. Consider the space $\tilde{\mathcal{X}} = \tilde{\mathcal{X}}_1 \times \ldots \times \tilde{\mathcal{X}}_d$ where $\tilde{\mathcal{X}}_i = \mathcal{X}_i \cup \{*\}$ where $*$ is a point that is not in $\mathcal{X}_i$ and represents a corrupted point. Let the data distribution be given by $\tilde{\mathbf{x}} \sim p(\tilde{\mathbf{x}})$ over the space $\tilde{\mathcal{X}}$. For a $\tilde{\mathbf{x}}$ define the associated mask vector $\mathbf{m} = (m_1, \ldots, m_d)$ in $\{0,1\}^d$ as follows. In $\tilde{\mathbf{x}}$ the corrupted points are represented by $\{*\}$, and for every $\tilde{\mathbf{x}}$ there is a corresponding $\mathbf{m}$ where $m_i = 1$ if $\tilde{x}_i \in \mathcal{X}_i$ and $m_i = 0$ if $\tilde{x}_i = *$.

Let the generator be given by $G : \tilde{\mathcal{X}} \times \{0,1\}^d \times [0,1]^d \to \mathcal{X}$, where the generator takes an $\tilde{\mathbf{x}} \sim p(\tilde{\mathbf{x}})$ with corresponding $\mathbf{m}$ (that can be derived from $\tilde{\mathbf{x}}$), and a d-dimensional noise vector $\mathbf{z} \sim p(\mathbf{z})$. The generator is the identity for the values $\tilde{x}_i$ in the vector $\tilde{\mathbf{x}}$ where $m_i = 1$. Let the distribution of G be given by $\hat{\mathbf{x}} \sim p(\hat{\mathbf{x}})$.

Let $\mathcal{H}$ be the hint space, which can be simply taken as $\mathcal{H} = [0,1]^d$. Let the discriminator be given by $D : \mathcal{X} \times \mathcal{H} \to [0,1]^d$. Let the hint distribution be $\mathbf{h} \sim p(\mathbf{h}|\mathbf{m})$ over $\mathcal{H}$, where the mask vector $\mathbf{m}$ corresponds to the vector $\mathbf{x} \in \mathcal{X}$. The amount of information contained in $\mathbf{h}$ about $\mathbf{m}$ is controlled with $p(\mathbf{h}|\mathbf{m})$. In Yoon et al. [2018] it is shown that if not enough information about $\mathbf{m}$ is passed to D then there are multiple distributions that G could learn that are all optimal w.r.t. D.

The generator G and the discriminator D are trained similar to in the conditional GAN (Section 5.2.2). For $\tilde{\mathbf{x}} \sim p(\tilde{\mathbf{x}})$ with associated $\mathbf{m}$, and $\mathbf{z} \sim p(\mathbf{z})$ noise, let

$$\bar{\mathbf{x}} := G(\tilde{\mathbf{x}}, \mathbf{m}, (1 - \mathbf{m}) \odot \mathbf{z}),$$
$$\hat{\mathbf{x}} := \mathbf{m} \odot \tilde{\mathbf{x}} + (1 - \mathbf{m}) \odot \bar{\mathbf{x}},$$

where $\odot$ is the element-wise product. Here $\bar{\mathbf{x}}$ is the vector of only the imputed values, and $\hat{\mathbf{x}}$ is the new vector containing the original data and the imputed values.

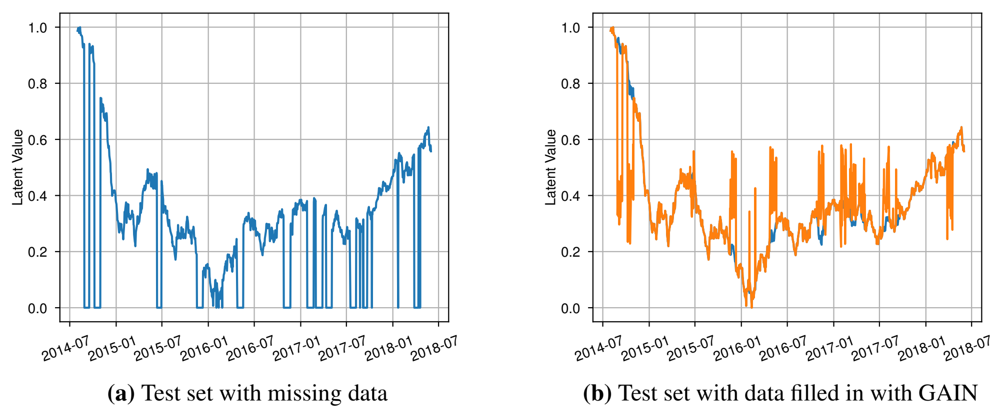
Let $\tilde{\mathbf{h}} \sim (\text{Bern}(p))^d$ be a d-dimensional vector of independent Bernoulli distributed values with probability p. The hint is defined as $\mathbf{h} := \mathbf{m} \odot \tilde{\mathbf{h}}$. The value p is the probability of keeping values in the mask for the hint. Choose $p = 0.1$ as that performs well according to Yoon [2019].

The discriminator D is given the imputed $\hat{\mathbf{x}}$ from the generator and a hint vector $\mathbf{h}$ which gives information about some of the mask values. Write the discriminator as $D(\hat{\mathbf{x}}, \mathbf{h})$. The discriminator returns a vector representing the probability of each element in $\mathbf{x}$ being imputed. Let

$$v(G, D) = \mathbf{E}_{\tilde{\mathbf{x}}, \tilde{\mathbf{h}}, \mathbf{z}} \left[\mathbf{m} \log D(\hat{\mathbf{x}}, h) + (\mathbf{1} - \mathbf{m}) \log(\mathbf{1} - D(\hat{\mathbf{x}}, \mathbf{h})) \right]$$

where $\hat{\mathbf{x}}$ is generated by G. The training procedure follows similarly to the conditional GAN in Section 5.2.2.

Consider the spot prices of the time-series, that is the short end of the futures curves, similar to in Section 8.1. Missing values over the training and test time-series are artificially created by randomly setting small intervals of 5-10 days as missing. The standardisation preprocessing method is applied



(a) Test set with missing data

(b) Test set with data filled in with GAIN

Figure 10.1: GAIN simulations on short end of WTI NYMEX with missing data. In Fig. 10.1a the missing data is represented by the latent value of 0 and this is shown in the graph by vertical lines going to 0.

to the data. The GAIN model is trained on the data with missing values and an example of imputation on the test set is shown in Fig. 10.1. It can be seen in Fig. 10.1b that the imputation adds a lot of noise to the time-series rather than filling in the missing values by interoplation with some small amount of noise like a classical model could do. Adjusting the p value for the hint vector does not give better results. Using log-returns data instead of standardised data seems to cause much instability in the model and further research would need to be taken to understand where this comes from. At the moment, the model as it stands is not useful for data imputation.

11 Detecting Unrealistic Scenarios

When working with large datasets from multiple sources, it can often be the case that there are problems with the data. When there are errors in the data, the data shows unrealistic market scenarios. For oil futures time-series, such unrealistic scenarios can consist of: pricing jumps in futures curves, constant priced futures curves, completely corrupt futures curves, and large pricing jumps and shape changes between day-to-day futures curves. Futures time-series are volatile over time and there is volatility in day-to-day futures curves however the shape of the futures curves usually changes gradually.

Given training data of oil futures where the data has been checked for errors, a model can be trained to detect unrealistic scenarios on other datasets. Model 11.1 can provide a value for how realistic a curve is. The model works by running curves through a trained autoencoder, where the autoencoder is trained on realistic data, and then the SMAPE value is calculated between the original curve and the reconstruction. If the reconstruction is good, then a low SMAPE value should be returned, while if the reconstruction is bad a high SMAPE value should be returned. The idea behind the model is that, the autoencoder works well on curves it has seen and also works well on interpolations between seen curves. Since the autoencoder has been trained on realistic curves, it can reconstruct those curves well. However, for curves it has not seen before, that are not similar to curves it has seen, it will not be able to reconstruct them well and this will lead to a high SMAPE score.

The model only takes into account the shape and pricing of individual futures curves. It does not examine whether futures curves over time are realistic. This could be done with a windowed autoencoder instead. This is a possibility for future research and it is not examined here.

Model 11.1 (Unrealistic Curve Detector)**.** *Let the training data be the realistic data. Let the test data be data to be check for unrealistic scenarios.*

1. *Apply preprocessing method standardisation over the tenors to the data.*
2. *Train the standard autoencoder Model 7.2 with $k = 2$ on the training data.*
3. *Encode and then decode the test data using the autoencoder.*
4. *For each curve in the test set, compute the mean squared error between the curve, and the curve after applying: preprocessing, encoding, decoding and undoing the preprocessing.*
5. *For a large enough SMAPE the curve is labeled as unrealistic.*

In Fig. 11.1 the futures time-series of WTI NYMEX is shown, with the associated SMAPE value for the curves shown in grey (scale on the right of the graph). The mean SMAPE value is 0.0074, with standard deviation 0.0039, minimum 0.0022, and maximum 0.026. Examining the 3d-plot shown in Fig. 2.3a no inconsistencies in the data can clearly be seen. The peaks in the SMAPE may be due to the training set not covering particular futures curves appearing in the test set, or due to the autoencoder having more difficulty encoding some curves compared to others. The highest SMAPE value of 0.026 is still small when compared with the SMAPE values of unrealistic curves that are shown in Fig. 11.2.

In Fig. 11.2, the curves are all self-generated and they do not originate from the available time-series data. The first 10 curves are constant priced curves, the next 8 curves are uniformly randomly distributed over a number of ranges, the next 7 curves are linearly upward sloping, and last 7 curves are linearly upward sloping. Apply Model 11.1 to these fake curves and examine the SMAPE values produced.

From the constant curves, for the curves priced very close to $\$0$ a high SMAPE value is returned, this makes sense because the training data does not have any curves priced close to $\$0$. For constant curves priced $\$50$ and higher the SMAPE value is small, and with respect to Fig. 11.1 would correspond to realistic curves. From the constant curves with jump, the curve with a jump from 100 to 150 has the lowest SMAPE value of 0.081, but this is still more than 3 times the worst SMAPE value from the test set. All the uniformly random curves perform badly and have a very high SMAPE score, this is exactly what is expected. The linearly increasing curves that operate over a too large a range have high SMAPE scores and the same occurs for the linearly decreasing curves, this is again what is expected, since curves in the training sets have prices of the long end not extremely far from prices of the short end.

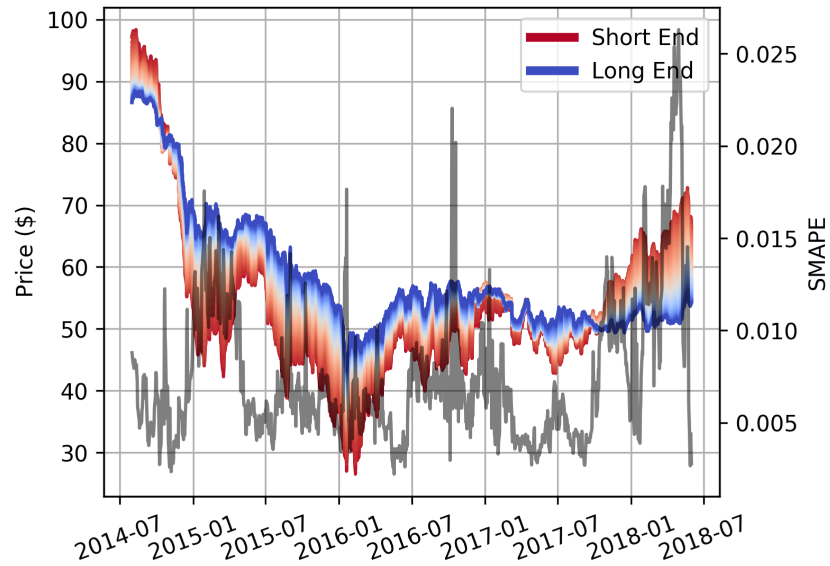


Figure 11.1: Detecting unrealistic scenarios on the test set, WTI NYMEX. The price axis of the futures time-series is shown on the left-hand size corresponding to the graph in red-blue. The realistic nature of the data, as a SMAPE score, is shown as the grey line with the axis on the right-hand side. The autoencoder is chosen as Model 7.2 with a latent dimension of $k = 2$. The preprocessing method chosen is *standardisation over the tenors*. The futures time-series is shown as a 2d representation where each futures curve is projected onto the x-y plane with the colour representing the maturity. The short end is shown in red and the long end (with a maturity of 4.3 years) is shown in blue. For each futures curve the associated SMAPE value corresponding to how realistic the curve is, is shown as the grey line.

From the results in Figs. 11.1 and 11.2 it can be concluded that Model 11.1 works well in general at detecting unrealistic curves. The SMAPE score gives a good indication on how realistic a curve is. For large SMAPE values, above 0.1 for instance, it is likely that these refer to unrealistic curves. For SMAPE values in the range of 0.3-0.1 it is more difficult to say with certainty whether a curve is unrealistic, however it does give an indication that the curve may have some unrealistic characteristics. One limitation of the model as mentioned previously, is that it cannot take into account the price movement over the time-series but only the shape of the futures curves.

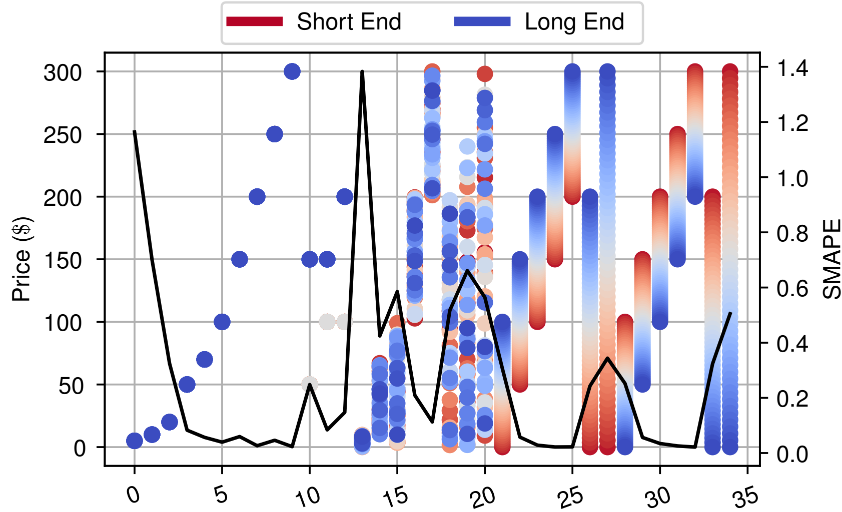


Figure 11.2: Fake data presented similarly to in Fig. 11.1. Fake curves with standardisation over tenors to show how well the autoencoder performs on unrealistic data. The curves are shown as a 2d representation. The first 10 points correspond to constant priced curves. The next 8 curves are uniformly randomly distributed. The next 7 curves are linearly upward sloping, and the last 7 curves are linearly downward sloping.

Type	Range	SMAPE
Constant	**5**	**1.2**
Constant	10	0.83
Constant	20	0.38
Constant	50	0.093
Constant	70	0.06
Constant	100	0.041
Constant	150	0.082
Constant	200	0.06
Constant	250	0.053
Constant	300	0.025
Constant with jump	**50, 150**	**0.25**
Constant with jump	100, 150	0.081
Constant with jump	**100, 200**	**0.15**
Uniformly random	**$[0, 10]$**	**1.2**
Uniformly random	**$[10, 70]$**	**0.30**
Uniformly random	**$[0, 100]$**	**0.55**
Uniformly random	**$[100, 200]$**	**0.22**
Uniformly random	**$[200, 300]$**	**0.10**
Uniformly random	**$[0, 200]$**	**0.64**
Uniformly random	**$[0, 250]$**	**0.60**
Uniformly random	**$[0, 300]$**	**0.52**
Linearly increasing	**$[0, 100]$**	**0.34**
Linearly increasing	$[50, 150]$	0.057
Linearly increasing	$[100, 200]$	0.029
Linearly increasing	$[150, 250]$	0.026
Linearly increasing	$[200, 300]$	0.026
Linearly increasing	**$[0, 200]$**	**0.25**
Linearly increasing	**$[0, 300]$**	**0.35**
Linearly decreasing	**$[100, 0]$**	**0.26**
Linearly decreasing	$[150, 50]$	0.052
Linearly decreasing	$[200, 100]$	0.039
Linearly decreasing	$[250, 150]$	0.028
Linearly decreasing	$[300, 200]$	0.022
Linearly decreasing	**$[200, 0]$**	**0.31**
Linearly decreasing	**$[300, 0]$**	**0.50**

Table 11.1: The data shown in Fig. 11.2 as a table. All rows in bold have a SMAPE of 0.1 or higher, indicating that it is clearly unrealistic.

12 Conclusion and Remarks

This thesis was written in co-operation with a large Dutch bank. The findings in this paper give them a solid overview of some of the most popular neural network models with direct applications to futures time-series data. The research gives a basis for techniques that worked well and others that are more difficult to implement, require further tuning, or are shown not to be useful. Moreover, it provides a starting point for further research into the application of neural networks on time-series data. Where some models give successful results, such as the performance improvement of using the GAN model over AMM, adjustments can still be made. The difficulty in applying more complex models such as RNNs and GAIN are shown, and although not usable in their current state, they provide a starting point for further development. The Unrealistic Curve Detector (Model 11.1) would be useful for the bank to filter incoming time-series futures data. The model can easily be trained on other products as well, which is one of the benefits of a neural network model.

The GAN models in Section 8 perform better than the benchmark two-factor mean-reverting Andersen Markov model. The AMM has a SMAPE mean of 0.26 and a standard deviation of 0.15 under the test set, while the GAN model using standardisation over the tenors, autoencoder and GAN-CONV as shown in Table 8.2 has a SMAPE mean of 0.083 with standard deviation of 0.01. In further research the GAN model could be used to account for interrelationships between market variables. Up to this point, the data fed into the GAN has been a d-dimensional encoded time-series of a single commodity. As mentioned in Section 2.3, a different method of splitting the datasets into training, validation and test sets could be chosen. For example, the first $3/4$ of each dataset could be treated as training data, the next $1/8$ as validation data and the last $1/8$ as test data. This would have allowed learning of the interrelationships in the different available products. Instead of training the GAN models in Section 8 on a single encoded commodity, the GAN could be trained on multiple commodities simultaneously. This has limitations when the number of commodities that are simulated simultaneously increases in the same manner that the GANs cannot handle encoded data with the size of the latent dimension above four. The reasons for not training GANs on encoded data with $k > 4$ is because these would have taken large amounts of time to train. For d much larger, other methods for reducing the computational complexity of the training task would need to be considered. One option would be to use something other than dense layers, for example convolutional layers, as seen before, to reduce the size of the network. Another option may be to apply dimension reduction techniques to multiple commodities simultaneously.

There is a new model called GAN-FD that promises better results for time-series simulation [Zhou et al. 2018]. The model is very similar to the models covered in Section 8, with the addition of incorporating the direction of the price movement in training. The motivation behind this is that the direction of the price is very important in trading. The adjustment in the model requires only the loss function of the generator to be adjusted to penalise simulations where the price is moving in the wrong direction. Although the paper states that the model benchmarked against various classical models performs well, there remains to be some skepticism because the day-to-day price direction is usually quite random, and therefore not be a useful indicator in the model.

The GAIN model that is trained to fill in missing values in time-series data is examined in Section 10. It was found that although the method is capable of filling missing values, they are usually far from the true values and introduce a lot of noise into the time-series. With more attention to the model, it may produce better results. In theory, once the missing values have been filled, methods in Section 8 to make simulations of the time series can be applied. It would be interesting to combine these models such that simulations from time-series with missing values can be made without first having to fill in the missing values. This could be achieved by allowing the conditional sequence of the GAN model to contain missing values. The problem is that filling in missing values may have some bias factor, and this will be incorporated in the simulations made from the time-series with filled missing values. By directly being able to simulate from time-series data with missing values, this possible bias would not affect the simulations.

In Section 11 an autoencoder model that can detect unrealistic scenarios in individual futures curves is examined. The method can successfully detect the difference between realistic data from the historical data available and purposely made unrealistic data. An interesting extension of this model would be to also factor in unrealistic movement between futures curves over the time-series. This could be done by incorporating a windowed autoencoder (Section 7.4).

A Mathematical Concepts

Some mathematical concepts are introduced that are used in the body of the thesis, most importantly in Sections 4 and 5.

A.1 Notation

Notation that is used in this paper is introduced. The distribution p can refer to different distributions based on the context, for example $p(x)$ and $p(z)$. The $p(\cdot)$ can refer to the probability distribution or the density function. The true distribution that the data is sampled from is referred to as p_{true} and the empirical probability distribution from the data is p_d. The probability distribution of the model is referred to as $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ or $p_{\boldsymbol{\theta}}(\mathbf{x})$, where $\boldsymbol{\theta}$ are the parameters of the model.

A.2 Information Theory

Shannon's entropy gives a measure of the amount of uncertainty in a random variable x given its distribution $p(x)$.

Definition A.1 (Entropy). *The entropy of a probability mass function p is*

$$H(p) := -\mathbb{E}_{x \sim p} \left[\log p(x) \right].$$

The entropy is always non-negative, $H(p) \geq 0$. The entropy H is maximal when all values that the random variable x can take are equally probable. For X a random variable with continuous density function f, the **continuous entropy** is defined as

$$H(X) := -\int f(x) \log f(x) \, dx.$$

The same properties as for the discrete entropy do not apply to the continuous entropy. In the continuous case, distributions can have negative entropy.

The cross-entropy can be seen as the expected number of bits (when using $\log_2$) required to encode random variable x when a distribution q is used while the true distribution of the data is p.

Definition A.2 (Cross-Entropy). *The cross-entropy between two probability distributions p and q over the same probability space is*

$$H(p, q) := -\mathbb{E}_{x \sim p} \left[\log q(x) \right].$$

The Kullback-Leibler divergence is defined.

Definition A.3 (Kullback-Leibler). *For continuous distributions with densities p and q defined on the same probability space, the Kullback-Leibler divergence from p to q is given by*

$$\begin{aligned} \mathbf{KL}(p||q) &:= \int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx \\ &= \mathbf{E}_{x \sim p(x)} \left[\log \left(\frac{p(x)}{q(x)} \right) \right], \end{aligned}$$

with $\frac{p(x)}{q(x)} = \infty$ if $p(x) > 0$ and $q(x) = 0$, and with $\frac{0}{0} =: 0$.

The Kullback-Leibler divergence is always non-negative, $\mathbf{KL}(p||q) \geq 0$ for all p, q. The Kullback-Leibler divergence can be defined in terms of entropy and cross-entropy as

$$\mathbf{KL}(p||q) = H(p, q) - H(p) \geq 0.$$

Note $H(p, q) \geq H(p) \geq 0$ since $H(p) \geq 0$. The Kullback-Leibler divergence can thus be seen as the relative entropy of p with respect to q. If p is fixed, then the cross-entropy and KL divergence is equal up to the additive constant $H(p)$. Minimising the cross-entropy w.r.t. q is then equivalent to minimising the KL divergence. In Bayesian terms, $\mathbf{KL}(p||q)$ can be seen as the information gained when beliefs from the prior probability distribution q is updated to the posterior probability distribution p.

Lemma A.1. *Let $p(x) \sim N(\mu_1, \sigma_1^2)$ and $q(x) \sim N(\mu_2, \sigma_2^2)$. Then a well known result is*

$$\mathbf{KL}(p||q) = \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}.$$

For $p(x) \sim N(\mu, \sigma^2)$ and $q(x) \sim N(0, 1)$ it follows that

$$\mathbf{KL}(p||q) = \frac{1}{2}\left(\mu^2 + \sigma^2 - \log \sigma^2 - 1\right).$$

A.3 Bayesian Statistics

Consider a model providing a joint probability distribution for $\mathbf{z}$ and $\mathbf{x}$. The joint probability density can be written as a product of the prior distribution $p(\mathbf{z})$ and the sampling distribution $p(\mathbf{x}|\mathbf{z})$. Namely

$$p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z}).$$

Conditioning on the known value of the data $\mathbf{x}$, using Bayes' rule, gives the posterior density

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{z}, \mathbf{x})}{p(\mathbf{x})} = \frac{p(\mathbf{z})p(\mathbf{x}|\mathbf{z})}{p(\mathbf{x})}. \tag{7}$$

The marginal distribution of $\mathbf{x}$, also called the prior predictive distribution, is

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) d\mathbf{z}. \tag{8}$$

Let $\tilde{\mathbf{x}}$ be unknown. Given $\mathbf{x}$ the prediction for $\tilde{\mathbf{x}}$, called the posterior predictive distribution, is

$$p(\tilde{\mathbf{x}}|\mathbf{x}) = \int p(\tilde{\mathbf{x}}, \mathbf{z}|\mathbf{x}) d\mathbf{z} = \int p(\tilde{\mathbf{x}}|\mathbf{z}, \mathbf{x})p(\mathbf{z}|\mathbf{x}) d\mathbf{z} = \int p(\tilde{\mathbf{x}}|\mathbf{z})p(\mathbf{z}|\mathbf{x}) d\mathbf{z}$$

where the last equality follows under the assumed conditional independence of $\mathbf{x}$ and $\tilde{\mathbf{x}}$ given $\mathbf{z}$ [Gelman et al. 2013].

A.4 Maximum Likelihood Estimation

Usually the parameters $\boldsymbol{\theta}$ are tuned such that $p_{\text{model}}(\cdot; \boldsymbol{\theta})$ is as close as possible to p_d and thus obtain an estimate for p_{true}. For a good estimate of p_d, the distribution p_d should lie in, or be close to the family given by $p_{\text{model}}(\cdot; \boldsymbol{\theta})$ over $\boldsymbol{\theta}$.

Theorem A.2. *Let $\mathbf{X} = \{\mathbf{x}^{(1)}, \dots \mathbf{x}^{(m)}\}$. Let p_d be the empirical distribution of $\mathbf{X}$. Assume the $\mathbf{x}^{(i)}$ are independently distributed. Given a family of distributions $\{p_{model}(\cdot; \theta) \mid \theta \in \Theta\}$, the maximum likelihood finds the parameter θ that maximises the likelihood function $p_{model}(\theta; \mathbf{x})$. The estimator for θ is defined as*

$$\begin{aligned}
\theta_{ML} &:= \arg\max_{\theta} p_{model}(\mathbf{x}; \theta) \\
&= \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \theta) \\
&= \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(\mathbf{x}^{(i)}; \theta) \\
&= \arg\max_{\theta} \mathbf{E}_{\mathbf{x} \sim p_d} \log p_{model}(\mathbf{x}; \theta) \\
&= \arg\max_{\theta} -H(p_d(\mathbf{x}), p_{model}(\mathbf{x}; \theta)).
\end{aligned}$$

Thus maximising the likelihood corresponds to minimising the cross-entropy between p_d and p_{model}.

A.5 Variational Inference

Variational inference is used to approximate posterior densities, and is an alternative to Markov chain Monte Carlo (MCMC). Using the Metropolis-Hastings method [van Es 2017], which is a type of

MCMC method, the distribution $p(\mathbf{z}|\mathbf{x})$ can be estimated given that sampling from $p(\mathbf{z})$ and $p(\mathbf{x}|\mathbf{z})$ can be done. This is useful when the integral in the denominator is intractable in

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{z})p(\mathbf{x}|\mathbf{z})}{\int p(\mathbf{z})p(\mathbf{x}|\mathbf{z})d\mathbf{z}} \propto p(\mathbf{z})p(\mathbf{x}|\mathbf{z}).$$

Since $p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$ is proportional to the desired distribution $p(\mathbf{z}|\mathbf{x})$, the Metropolis Hastings algorithm can be used. Variational inference is used for large datasets or complex models since it can approximate the posterior distribution faster than with MCMC, however MCMC has asymptotic consistency which is not guaranteed with variational inference [Blei et al. 2017].

To find an estimate for the posterior distribution $p(\mathbf{z}|\mathbf{x})$ and for $p(\mathbf{x})$ can be difficult. Consider Bayes' theorem in Eq. (7), finding the denominator $p(\mathbf{x})$ can be hard since the marginal distribution in Eq. (8) can be difficult to compute. Let $\mathcal{D}$ be a family of distributions $q(\mathbf{z})$ which are simpler than the posterior but can be good approximations for it. To find $q^* \in \mathcal{D}$ such that $q^*(\mathbf{z})$ is the best approximation to $p(\mathbf{z}|\mathbf{x})$ in $\mathcal{D}$, the following steps are taken. The Kullback-Leibler divergence of p from q is most commonly chosen to measure how different p is from q. Using the Kullback-Leibler divergence

$$q^*(\mathbf{z}) = \operatorname*{arg\,min}_{q \in \mathcal{D}} \mathbf{KL}(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x})).$$

Then

Since $p(\mathbf{x})$ is a constant w.r.t. to $q(\mathbf{z})$, minimising $\mathbf{KL}(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x}))$ corresponds to maximising

$$\mathcal{L}(q) := -\mathbf{KL}(q(\mathbf{z})||p(\mathbf{z})) + \mathbf{E}_{q(\mathbf{z})}\left[\log\left(p(\mathbf{x}|\mathbf{z})\right)\right]$$

where $\mathcal{L}(q)$ is called the evidence lower bound. Substituting $\mathcal{L}$ into **??** and rearranging for $\log p(\mathbf{x})$ gives

$$\log p(\mathbf{x}) = \mathbf{KL}(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x})) + \mathcal{L}(q).$$

Since $\log p(\mathbf{x})$ is fixed with respect to q, maximising $\mathcal{L}(q)$ minimises $\mathbf{KL}(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x}))$. Since $\mathbf{KL}(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x}))$ is non-negative, the lower bound $\log p(\mathbf{x}) \geq \mathcal{L}(q)$ is attained. The problem is reduced to maximising $\mathcal{L}(q)$ over all probability distributions $q \in \mathcal{D}$. This is the method of variational inference. Variational inference is used in the variational autoencoder (Section 5.1.2).

A.6 Reparametrisation Trick

The general setting is that something of the following form needs to be computed

$$\nabla_\theta \mathbb{E}_{x\sim q_\theta}\left[f(x)\right].$$

This can be rewritten as

$$\begin{aligned} \nabla_\theta \int q_\theta(x) f(x)\, dx &= \int \frac{q_\theta(x)}{q_\theta(x)} f(x) \nabla_\theta q_\theta(x)\, dx \\ &= \mathbb{E}_{x\sim q_\theta}\left[f(x)\frac{1}{q_\theta(x)}\nabla_\theta q_\theta(x)\right] \\ &= \mathbb{E}_{x\sim q_\theta}\left[f(x)\nabla_\theta \log(q_\theta(x))\right] \end{aligned}$$

Unfortunately an empirical estimate of $f(x)\nabla_\theta \log(q_\theta(x))$ to estimate $\nabla_\theta \mathbb{E}_{x\sim q_\theta}\left[f(x)\right]$ can be of high variance [Kingma et al. 2013; Paisley et al. 2012]. The problem is that rare values of x from $q_\theta(x)$ maybe be essential in $\nabla_\theta \mathbb{E}_{x\sim q_\theta}\left[f(x)\right]$. One method to address this is called the **reparametrisation trick**.

Lemma A.3 (Reparametrisation trick)**.** *Rewrite q_θ as a function $g(\theta,\epsilon)$ of θ and a random variable $\epsilon \sim p(\epsilon)$ that does not depend on θ. Assume q_θ satisfies the conditions outlined in [Kingma et al. 2013]. Then $x \sim g(\theta,\epsilon)$, giving*

$$\begin{aligned} \nabla_\theta \mathbb{E}_{x\sim q_\theta}\left[f(x)\right] &= \nabla_\theta \mathbb{E}_{\epsilon\sim p}\left[f(g(\theta,\epsilon))\right] \\ &= \mathbb{E}_{\epsilon\sim p}\left[\nabla_\theta f(g(\theta,\epsilon))\right]. \end{aligned}$$

With the reparametrisation trick, the earlier problems are avoided since the distribution p does not depend on θ.

Example A.1. *Let* $q_\theta = N(\theta, 1)$ *and* $x \sim q_\theta(x)$. *Suppose one wants to compute* $\nabla_\theta \mathbb{E}_{x \sim q_\theta}[f(x)]$. *With the reparametrisation trick, the distribution under which the expectation is taken can be rewritten such that it is independent of parameter* θ. *Let* $p(\epsilon) = N(0, 1)$ *and* $\epsilon \sim p(\epsilon)$. *Now* $x \sim \theta + \epsilon$ *and*

$$\begin{aligned}\nabla_\theta \mathbb{E}_{q_\theta}[f(x)] &= \nabla_\theta \mathbb{E}_{\epsilon \sim p}[f(\theta + \epsilon)] \\ &= \mathbb{E}_{\epsilon \sim p}[\nabla_\theta f(\theta + \epsilon)].\end{aligned}$$

Introduced in Kingma et al. [2013, Section 2.4], the reparametrisation trick is needed to backpropagate through a random node and this will be used for the variational autoencoder (Section 5.1.2).

B Andersen Markov Model

Let W_1, W_2 be independent Brownian motions under the risk-neutral measure Q. The forward prices are determined by the SDE in Eq. (1). The seasonality and drift terms are ignored by setting $a(T) = 0$ and $\mu(t, T) = 0$. The model written in vector form

$$\begin{aligned}\frac{d\hat{F}(t,T)}{\hat{F}(t,T)} &= \begin{pmatrix}\eta_1 e^{\kappa(T-t)+\eta_\infty} \\ \eta_2 e^{-\kappa(T-t)}\end{pmatrix}^\top d\begin{pmatrix}W_1(t)\\ W_2(t)\end{pmatrix} \\ &= \sigma(t,T)^\top dW(t)\end{aligned}$$

where

$$\sigma(t,T) := \alpha(t)\beta(T), \quad \alpha(t) := \begin{pmatrix}\eta_1 e^{\kappa t} & \eta_\infty \\ \eta_2 e^{\kappa t} & 0\end{pmatrix}, \quad \beta(T) := \begin{pmatrix}e^{-\kappa T}\\ 1\end{pmatrix}, \quad W(t) = \begin{pmatrix}W_1(t)\\ W_2(t)\end{pmatrix}.$$

When the seasonality adjustment $a(T) = 0$, then $\sigma(t, T)$ is a function of only time to maturity $T - t$ instead of calendar time t and time of maturity T separately. The function $\sigma(t, T)$ can then be written as $\sigma(T - t)$. This is called volatility term structure stationarity.

Define

$$\begin{aligned}x(t) &:= \int_0^t \alpha(u)^\top dW(u)\\ y(t) &:= \int_0^t \alpha(u)^\top \alpha(u) du,\end{aligned}$$

with $x(0) = y(0) = 0$. Note $\sigma(t, T)$ can be written in the form $\sigma(t, T) = \alpha(t)\beta(T)$. Then by Itô's formula

$$\begin{aligned}d\ln(\hat{F}(t,T)) &= \frac{1}{\hat{F}(t,T)} d\hat{F}(t,T) - \frac{1}{2}\frac{1}{(\hat{F}(t,T))^2} d\left\langle \hat{F}(t,T)\right\rangle \\ &= \sigma(t,T)^\top dW(t) - \frac{1}{2}\frac{1}{(\hat{F}(t,T))^2}\left(\hat{F}(t,T)\sigma(t,T)^\top\right)^2 dt \\ &= (\alpha(t)\beta(T))^\top dW(t) - \frac{1}{2}\left((\alpha(t)\beta(T))^\top\right)^2 dt.\end{aligned}$$

Then

$$\ln(\hat{F}(t,T)) = \ln(\hat{F}(0,T)) + \int_0^t (\alpha(u)\beta(T))^\top dW(u) - \frac{1}{2}\int_0^t \left((\alpha(u)\beta(T))^\top\right)^2 du.$$

Thus

$$\hat{F}(t,T) = \hat{F}(0,T)\exp\left(\beta(T)^\top x(t) - \frac{1}{2}\beta(T)^\top y(t)\beta(T)\right), \quad T \geq t. \tag{9}$$

Writing out $x(t) = (x_1(t), x_2(t))^\top$ gives

$$\begin{aligned}dx_1(t) &= \eta_1 e^{\kappa t} dW_1(t) + \eta_2 e^{\kappa t} dW_2(t),\\ dx_2(t) &= \eta_\infty dW_1(t).\end{aligned}$$

Perform the following change of variables on $x(t)$

$$z_1(t) := e^{-\kappa t} x_1(t), \quad z_2(t) := x_2(t),$$

with $z_1(0) = z_2(0) = 0$. By integrating

$$\begin{aligned}z_1(t) &= z_1(0)e^{-\kappa t} + \int_0^t e^{-\kappa(t-s)}\eta_1 dW_1(s) + \int_0^t e^{-\kappa(t-s)}\eta_2 dW_2(s)\\ &= \int_0^t e^{-\kappa(t-s)}\eta_1 dW_1(s) + \int_0^t e^{-\kappa(t-s)}\eta_2 dW_2(s)\\ &= \eta_1 Y_1(t) + \eta_2 Y_2(t),\\ z_2(t) &= \int_0^t \eta_\infty dW_1(s) = \eta_\infty W_1(t),\end{aligned}$$

where $Y_i(t) = \int_0^t e^{-\kappa(t-s)} dW_i(s)$. The $Y_i(t)$ are Ornstein-Uhlenbeck processes with mean reversion speed κ, no drift component, and volatility 1.

Writing out $y(t)$ is straightforward, and plugging $y(t)$ and $x(t)$ into Eq. (9) gives Andersen 2008[Eq. (1.122)]

$$\ln \hat{F}(t,T) = \ln \hat{F}(0,T) + \Big(z_1(t)e^{-\kappa(T-t)} + z_2(t)\Big) - \mu'(t,T), \tag{10}$$

where the drift term $\mu'(t,T)$ is defined as

$$\mu'(t,T) = \frac{1}{2}e^{2a(T)}\frac{e^{-2\kappa t}\left(e^{2\kappa t-1}\left(\eta_1^2+\eta_2^2\right) + 4\eta_1\eta_\infty e^{-\kappa T}\right)\left(e^{\kappa t}-1\right) + 2\eta_\infty^2 t\kappa}{2\kappa}.$$

The variables $z_1(t)$ and $z_2(t)$ are written such that they can easily be simulated, allowing a discrete simulation for $F(t,T)$ to be made. For given t and δ define the increments

$$I_1(t) := z_1(t+\delta) - e^{-\kappa\delta}z_1(t) = \sum_{i=1}^{2}\eta_i\int_t^{t+\delta} e^{-\kappa(t+\delta-s)}dW_i(s), \tag{11}$$

$$I_2(t) := z_2(t+\delta) - z_2(t) = \eta_\infty(W_1(t+\delta) - W_1(t)). \tag{12}$$

Using the Ito isometry

$$\operatorname{Var}(I_1(t)) = \sum_{i=1}^{2}\eta_i^2 e^{-2\kappa(t+\delta)}\int_t^{t+\delta} e^{2\kappa s}ds = (\eta_1^2+\eta_2^2)\frac{1}{2\kappa}(1-e^{-2\kappa\delta}).$$

By Gaussian increments of a Wiener process

$$\operatorname{Var}(I_2(t)) = \eta_\infty^2\delta.$$

Using the Ito isometry

$$\begin{aligned}
\operatorname{cov}(I_1(t), I_2(t)) = \mathbf{E}\left[I_1(t)I_2(t)\right] &= \eta_\infty\eta_1\mathbf{E}\left[(W_1(t+\delta)-W_1(t))\int_t^{t+\delta}e^{-\kappa(t+\delta-s)}dW_1(s)\right] \\
&\quad + \eta_\infty\eta_2\mathbf{E}\left[(W_1(t+\delta)-W_1(t))\int_t^{t+\delta}e^{-\kappa(t+\delta-s)}dW_2(s)\right] \\
&= \eta_\infty\eta_1\mathbf{E}\left[\int_t^{t+\delta}dW_1(s)\int_t^{t+\delta}e^{-\kappa(t+\delta-s)}dW_1(s)\right] \\
&\quad + \eta_\infty\eta_2\mathbf{E}\left[\int_t^{t+\delta}dW_1(s)\right]\mathbf{E}\left[\int_t^{t+\delta}e^{-\kappa(t+\delta-s)}dW_2(s)\right] \\
&= \eta_\infty\eta_1\mathbf{E}\left[\int_t^{t+\delta}e^{-\kappa(t+\delta-s)}ds\right] \\
&= \eta_\infty\eta_1\frac{1}{\kappa}(1-e^{-\kappa\delta}).
\end{aligned}$$

The correlation is then given by

$$\rho := \frac{\operatorname{cov}(I_1(t), I_2(t))}{\sqrt{\operatorname{Var}(I_1(t))\operatorname{Var}(I_2(t))}} = \frac{\eta_1}{\sqrt{\eta_1^2+\eta_2^2}}\frac{1-e^{-\kappa\delta}}{\kappa\sqrt{\delta}}\left(\frac{1-e^{-2\kappa\delta}}{2\kappa}\right)^{-\frac{1}{2}}.$$

Using this, the increments can be written as

$$\begin{aligned}
z_1(t+\delta) &= e^{-\kappa\delta}z_1(t) + \sqrt{\operatorname{Var} I_1(t)}\left(\rho Z_1 + \sqrt{1-\rho^2}Z_2\right) \\
z_2(t+\delta) &= z_2(t) + \sqrt{\operatorname{Var} I_2(t)}Z_2,
\end{aligned}$$

where Z_1, Z_2 are i.i.d. standard normal variables.

The number of business days in a year is approximately 252. The time increment is set to one business day, giving the δ value of $\delta = 1/252$. Assume that the values for parameters $\kappa, \eta_\infty, \eta_1$ and

η_2 are known. Given that $z_1(0) = z_2(0) = 0$, the $z_1(t), z_2(t)$ can be simulated for discrete steps $0, \delta, 2\delta, 3\delta, ...$ The $F(t,T)$ can be simulated using Eq. (10).

The parameters κ, η_1, η_2, and η_∞ are calibrated to match the covariance structure of the historical time-series of futures curves.

Let the tenors T_i be durations. The choice for $T_1 = 0, T_2 = 30, T_3 = 60, ...$ where the durations are in days is made, where 30 days represents one month. The T_i are therefore not dates. Note that $F(t,T_i) = \hat{F}(t, t+T_i)$, where $t + T_i$ can be considered as a date.

Define log-returns as

$$I_{T_i} := \ln(\hat{F}(t+\delta, t+T_i+\delta)) - \ln(\hat{F}(t, t+T_i+\delta)).$$

This is the same as writing

$$I_{T_i} = \ln\left(F(t+\delta, T_i+\delta)\right) - \ln\left(F(t, T_i+\delta)\right).$$

Let $\mu' = \mu'(t, t+T_i+\delta) - \mu'(t+\delta, t+T_i+\delta)$ it follows from Eq. (10) that

$$\begin{aligned} I_{T_i} &= \left(z_1(t+\delta)e^{-\kappa(t+T_i+\delta-(t+\delta))} + z_2(t+\delta)\right) - \left(z_1(t)e^{-\kappa(t+T_i+\delta-t)} + z_2(t)\right) + \mu' \\ &= e^{-\kappa T_i}(z_1(t+\delta) - e^{-\kappa\delta}z_1(t)) + (z_2(t+\delta) - z_2(t)) + \mu' \\ &= e^{-\kappa T_i} I_1(t) + I_2(t) + \mu', \end{aligned}$$

where I_1 and I_2 defined in Eqs. (11) and (12) respectively. The covariance of log-returns is given by

$$\begin{aligned} \operatorname{cov}(I_{T_i}, I_{T_j}) &= e^{-\kappa T_i - \kappa T_j} \operatorname{Var}(I_1(t)) + \operatorname{Var}(I_2(t)) \\ &\quad + (e^{-\kappa T_i} + e^{-\kappa T_j}) \operatorname{cov}(I_1(t), I_2(t)) \\ &= e^{-\kappa T_i - \kappa T_j}(\eta_1^2 + \eta_2^2)\frac{1 - e^{-2\kappa\delta}}{2\kappa} + \eta_\infty^2 \delta \\ &\quad + (e^{-\kappa T_i} + e^{-\kappa T_j})\frac{\eta_1 \eta_\infty}{\kappa}(1 - e^{-\kappa\delta}). \end{aligned}$$

Note that $\operatorname{cov}(I_{T_i}, I_{T_j})$ is independent of t.

Consider historical prices $F(t_i, T_j)$ for dates $t_1, ..., t_m$, with $t_i - t_{i-1} = \delta$, and tenors $T_1, ..., T_n$. Let $\mathbf{x}$ be the $((m-1) \times n)$ matrix of historical log-returns, with tenors $T_1, ..., T_n$ and times $t_1, ..., t_{m-1}$, then the covariance is given by $\mathbf{h} = \operatorname{Cov}(\mathbf{x}^\top)$ where the times $t_1, ..., t_{m-1}$ refer to realisations. The $\mathbf{h}$ is then a $(n \times n)$ matrix. The parameters $\kappa, \eta_1, \eta_2, \eta_\infty$ are found by tuning the theoretical covariance to match the covariance of the historical data by minimising the mean squared error value function

$$V(\kappa, \eta_1, \eta_2, \eta_\infty) = \sum_{i,j=1}^{n} (\operatorname{cov}(I_{T_i}, I_{T_j}) - \mathbf{h}_{ij})^2.$$

The volatility parameters η_1 and η_2, and the long-term volatility parameter η_∞ are difficult to interpret, however these can be used to specify an alternative parameterisation Wiemer 2015[7.1.2]. Let σ_0 be the short-term volatility, σ_∞ be the long-term volatility, and ρ_∞ be the correlation between them. Then

$$\begin{aligned} \eta_1 &= -\sigma_\infty + \rho_\infty \sigma_0, \\ \eta_2 &= \sigma_0 \sqrt{1 - \rho_\infty^2}, \\ \eta_\infty &= \sigma_\infty. \end{aligned}$$

The initial values for the minimisation can be chosen as

$$\kappa = 0.5, \quad \sigma_0 = \sqrt{\mathbf{h}_{11}}, \quad \sigma_\infty = \sqrt{\mathbf{h}_{nn}}, \quad \text{and} \quad \rho_\infty = \frac{\mathbf{h}_{1n}}{\sqrt{\mathbf{h}_{11}\mathbf{h}_{nn}}}.$$

The $V(\kappa, \eta_1, \eta_2, \eta_\infty)$ can be minimised by using a numerical method, for example the Broyden–Fletcher–Goldfarb–Shanno Fletcher 1987 quasi-Newton method.

C Code

All the code is written in Python and can be found on the Github repository Ruiz 2019 with instructions on how to set up the project. Note that the time-series data of oil futures is not included because the data is not publicly available. The data was attained in co-operation with a large Dutch bank. The code for the neural networks is written using the Keras library and the Tensorflow library. The code for the standard autoencoder, variational autoencoder and adversarial autoencoder, is based on the examples from the Keras website Chollet 2016. The code for the GAN, conditional GAN and Wasserstein GAN is based on examples from Linder-Norén 2019. The code for the GAIN model is based on the Tensorflow code from Yoon 2019. The code for the LSTM model is based on the code from Tonin 2019.

Popular Summary

In financial risk management, being able to simulate futures products is important to calculate certain risks. Futures are financial contracts between a buyer and a seller to buy/sell an asset at a predetermined time in the future for a fixed price. To do this, there exist mathematical models that can simulate these products. These models have to be considered carefully and be made to fit the type of product at hand. This requires expert knowledge with still the downside that certain features that occur in the historical pricing data may be overlooked and not incorporated in the model. Neural networks are introduced to offer solutions to this problem. They are functions with many parameters that can be trained on historical data and learn patterns in the data. They are much more malleable compared to classical models. They can be easily retrained on new historical data without having to recreate the model itself to include new features seen in the data. In a financial institution, there is a reliance on a number of data sources to provide the historical data. The data can sometimes contain missing values or contain pricing inconsistencies. It is a time consuming process requiring manual work to find these errors. A neural network is shown to be capable of learning what realistic data looks like and thereby discover such errors. In carrying out these tasks, an introduction to neural networks is given, together with an examination of the specialised types of neural networks required. In order to train the networks on the historical time-series futures data available, the data has to be prepared by transforming it into a format that can be understood by the neural network. This data preparation can be done using a multitude of different methods where the choice affects the performance of the simulations. Research is carried out into which type of data preparation coupled with which specialised neural network performs best at simulating time-series data and detecting unrealistic data. The best neural network model found for simulating time-series of futures data performs three times better compared to the benchmark Andersen Markov model.

References

Administration, U.S. Energy Information (July 16, 2012). *Crude oils have different quality characteristics - Today in Energy - U.S. Energy Information Administration (EIA)*. URL: https://www.eia.gov/todayinenergy/detail.php?id=7110 (visited on 08/29/2019).

Ahmed, Nesreen, Amir Atiya, Neamat El Gayar, and Hisham El-Shishiny (2010). "An Empirical Comparison of Machine Learning Models for Time Series Forecasting". In: *Econometric Reviews* 29.5, pp. 594–621. URL: https://ideas.repec.org/a/taf/emetrv/v29y2010i5-6p594-621.html (visited on 05/24/2019).

Andersen, Leif B. G. (Sept. 30, 2008). *Markov Models for Commodity Futures: Theory and Practice*. SSRN Scholarly Paper ID 1138782. Rochester, NY: Social Science Research Network. URL: https://papers.ssrn.com/abstract=1138782 (visited on 07/08/2019).

Arjovsky, Martin and Léon Bottou (Jan. 17, 2017a). "Towards Principled Methods for Training Generative Adversarial Networks". In: *arXiv:1701.04862 [cs, stat]*. arXiv: 1701.04862. URL: http://arxiv.org/abs/1701.04862 (visited on 09/11/2019).

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (Jan. 26, 2017b). "Wasserstein GAN". In: *arXiv:1701.07875 [cs, stat]*. arXiv: 1701.07875. URL: http://arxiv.org/abs/1701.07875 (visited on 06/24/2019).

Barunik, Jozef and Barbora Malinska (Apr. 2015). "Forecasting the term structure of crude oil futures prices with neural networks". In: *arXiv:1504.04819 [q-fin]*. arXiv: 1504.04819. URL: http://arxiv.org/abs/1504.04819 (visited on 01/24/2019).

Bishop, Christopher M. (2009). *Pattern recognition and machine learning*. Corrected at 8th printing 2009. Information science and statistics. OCLC: 845772798. New York, NY: Springer. 738 pp. ISBN: 978-0-387-31073-2 978-1-4939-3843-8.

Blei, David M., Alp Kucukelbir, and Jon D. McAuliffe (Apr. 3, 2017). "Variational Inference: A Review for Statisticians". In: *Journal of the American Statistical Association* 112.518, pp. 859–877. ISSN: 0162-1459, 1537-274X. DOI: 10.1080/01621459.2017.1285773. arXiv: 1601.00670. URL: http://arxiv.org/abs/1601.00670 (visited on 05/14/2019).

Box, G. E. P. and D. R. Cox (1964). "An Analysis of Transformations". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 26.2, pp. 211–252. ISSN: 0035-9246. URL: https://www.jstor.org/stable/2984418 (visited on 05/24/2019).

Brownlee, Jason (May 10, 2018a). *A Gentle Introduction to Normality Tests in Python*. Machine Learning Mastery. URL: https://machinelearningmastery.com/a-gentle-introduction-to-normality-tests-in-python/ (visited on 05/24/2019).

– (Oct. 30, 2018b). *Comparing Classical and Machine Learning Algorithms for Time Series Forecasting*. Machine Learning Mastery. URL: https://machinelearningmastery.com/findings-comparing-classical-and-machine-learning-methods-for-time-series-forecasting/ (visited on 10/18/2019).

– (Oct. 7, 2018c). *How to Develop Convolutional Neural Networks for Multi-Step Time Series Forecasting*. Machine Learning Mastery. URL: https://machinelearningmastery.com/how-to-develop-convolutional-neural-networks-for-multi-step-time-series-forecasting/ (visited on 09/16/2019).

– (Feb. 3, 2019). *How to Improve Neural Network Stability and Modeling Performance With Data Scaling*. Machine Learning Mastery. URL: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/ (visited on 05/23/2019).

Chen, Chao, Jamie Twycross, and Jonathan M. Garibaldi (Mar. 24, 2017). "A new accuracy measure based on bounded relative error for time series forecasting". In: *PLoS ONE* 12.3. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0174202. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5365136/ (visited on 07/29/2019).

Cho, Kyunghyun, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio (June 3, 2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *arXiv:1406.1078 [cs, stat]*. arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078 (visited on 06/18/2019).

Chollet, Francois (May 14, 2016). *Building Autoencoders in Keras*. URL: https://blog.keras.io/building-autoencoders-in-keras.html (visited on 01/24/2019).

Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio (Dec. 11, 2014). "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *arXiv:1412.3555 [cs]*. arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555 (visited on 06/18/2019).

Eddy, Joseph (2018). *Time Series Forecasting with Convolutional Neural Networks - a Look at WaveNet*. URL: https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_conv/ (visited on 09/16/2019).

Edwards, D. A. (Jan. 1, 2011). "On the Kantorovich–Rubinstein theorem". In: *Expositiones Mathematicae* 29.4, pp. 387–398. ISSN: 0723-0869. DOI: 10.1016/j.exmath.2011.06.005. URL: http://www.sciencedirect.com/science/article/pii/S0723086911000430 (visited on 09/11/2019).

Filipovic, Damir (2009). *Term-Structure Models: A Graduate Course*. Springer Finance Textbooks. Berlin Heidelberg: Springer-Verlag. ISBN: 978-3-540-09726-6. URL: https://www.springer.com/gp/book/9783540097266 (visited on 02/04/2019).

Fletcher, R. (1987). *Practical Methods of Optimization; (2Nd Ed.)* New York, NY, USA: Wiley-Interscience. ISBN: 978-0-471-91547-8.

Forecasters, International Institute of (2000). *M3-Competition*. International Institute of Forecasters. URL: https://forecasters.org/resources/time-series-data/m3-competition/ (visited on 03/28/2019).

Fu, Rao, Jie Chen, Shutian Zeng, Yiping Zhuang, and Agus Sudjianto (Apr. 25, 2019). "Time Series Simulation by Conditional Generative Adversarial Net". In: *arXiv:1904.11419 [cs, eess, stat]*. arXiv: 1904.11419. URL: http://arxiv.org/abs/1904.11419 (visited on 07/03/2019).

Gelman, Andrew, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin (Nov. 1, 2013). *Bayesian Data Analysis, Third Edition*. Google-Books-ID: ZXL6AQAAQBAJ. CRC Press. 677 pp. ISBN: 978-1-4398-4095-5.

Goetzmann, William and Geert Rouwenhorst (2008). *The History of Financial Innovation, in Carbon Finance, Environmental Market Solutions to Climate Change.* URL: https://environment.yale.edu/publication-series/5767.html (visited on 08/29/2019).

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (June 2014a). "Generative Adversarial Networks". In: *arXiv:1406.2661 [cs, stat]*. arXiv: 1406.2661. URL: http://arxiv.org/abs/1406.2661 (visited on 01/24/2019).

Goodfellow, Ian J., Oriol Vinyals, and Andrew M. Saxe (Dec. 19, 2014b). "Qualitatively characterizing neural network optimization problems". In: *arXiv:1412.6544 [cs, stat]*. arXiv: 1412.6544. URL: http://arxiv.org/abs/1412.6544 (visited on 05/17/2019).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville (Mar. 31, 2017). "Improved Training of Wasserstein GANs". In: *arXiv:1704.00028 [cs, stat]*. arXiv: 1704.00028. URL: http://arxiv.org/abs/1704.00028 (visited on 06/25/2019).

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-term Memory". In: *Neural computation* 9, pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735. URL: https://www.researchgate.net/publication/13853244_Long_Short-term_Memory.

Hotelling, Harold (Jan. 1, 1933). "Analysis of a complex of statistical variables into principal components". In: *Journal of Educational Psychology*. DOI: 10.1037/h0071325.

Interactive, EKT (2019). *Oil 101 - History of Oil - A Timeline of the Modern Oil Industry*. EKT Interactive. URL: https://www.ektinteractive.com/history-of-oil/ (visited on 03/28/2019).

Karpathy, Andrej (2015). *CS231n Convolutional Neural Networks for Visual Recognition*. Convolutional Neural Networks for Visual Recognition. URL: http://cs231n.github.io/convolutional-networks/ (visited on 06/04/2019).

Kingma, Diederik P. and Max Welling (Dec. 2013). "Auto-Encoding Variational Bayes". In: *arXiv:1312.6114 [cs, stat]*. arXiv: 1312.6114. URL: http://arxiv.org/abs/1312.6114 (visited on 01/24/2019).

Kompella, Ravindra (Jan. 17, 2018). *Using LSTMs to forecast time-series*. Towards Data Science. URL: https://towardsdatascience.com/using-lstms-to-forecast-time-series-4ab688386b1f (visited on 06/18/2019).

Laptev, Nikolay, Slawek Smyl, and Santhosh Shanmugam (June 9, 2017). *Engineering Extreme Event Forecasting at Uber with Recurrent Neural Networks*. Uber Engineering Blog. URL: https://eng.uber.com/neural-networks/ (visited on 06/28/2019).

Leon, J (May 19, 2018). *How do I draw an LSTM cell in Tikz?* TeX - LaTeX Stack Exchange. URL: https://tex.stackexchange.com/questions/432312/how-do-i-draw-an-lstm-cell-in-tikz (visited on 11/24/2019).

Linder-Norén, Erik (Feb. 18, 2019). *Keras implementations of Generative Adversarial Networks.: eriklindernoren/Keras-GAN*. original-date: 2017-07-11T16:24:53Z. URL: https://github.com/eriklindernoren/Keras-GAN (visited on 02/18/2019).

Makhzani, Alireza, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey (Nov. 2015). "Adversarial Autoencoders". In: *arXiv:1511.05644 [cs]*. arXiv: 1511.05644. URL: http://arxiv.org/abs/1511.05644 (visited on 01/24/2019).

Makridakis, Spyros, Evangelos Spiliotis, and Vassilios Assimakopoulos (Mar. 27, 2018). "Statistical and Machine Learning forecasting methods: Concerns and ways forward". In: *PLOS ONE* 13.3, e0194889. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0194889. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0194889 (visited on 06/18/2019).

Mouawad, Jad (Nov. 9, 2007). "Rising Demand for Oil Provokes New Energy Crisis". In: *The New York Times*. ISSN: 0362-4331. URL: https://www.nytimes.com/2007/11/09/business/worldbusiness/09oil.html (visited on 05/07/2019).

Ng, Andrew (2018). *CS229 Lecture Notes*. URL: http://cs229.stanford.edu/notes/cs229-notes1.pdf (visited on 05/20/2019).

Olah, Christopher (Aug. 25, 2015). *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 09/17/2019).

Paisley, John, David M Blei, and Michael I Jordan (2012). "Variational Bayesian Inference with Stochastic Search". In: p. 8.

Pearson, Karl (Nov. 1, 1901). "On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11, pp. 559–572. ISSN: 1941-5982. DOI: 10.1080/14786440109462720. URL: https://doi.org/10.1080/14786440109462720 (visited on 07/16/2019).

Ruiz, Sebastian (Nov. 11, 2019). *sebastian-ruiz/deep-learning-simulating-time-series-futures*. original-date: 2019-11-11T09:50:54Z. URL: https://github.com/sebastian-ruiz/deep-learning-simulating-time-series-futures (visited on 11/11/2019).

Saha, Sumit (Dec. 17, 2018). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Medium. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visited on 09/16/2019).

Schafer, Ronald (July 2011). "What Is a Savitzky-Golay Filter? [Lecture Notes]". In: *IEEE Signal Processing Magazine* 28.4, pp. 111–117. ISSN: 1053-5888. DOI: 10.1109/MSP.2011.941097. URL: http://ieeexplore.ieee.org/document/5888646/ (visited on 07/23/2019).

Schlegl, Thomas, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs (Mar. 17, 2017). "Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery". In: *arXiv:1703.05921 [cs]*. arXiv: 1703.05921. URL: http://arxiv.org/abs/1703.05921 (visited on 05/16/2019).

Shafkat, Irhum (Apr. 5, 2018). *Intuitively Understanding Variational Autoencoders*. Medium. URL: https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf (visited on 09/08/2019).

Taylor, Sean J. and Benjamin Letham (Sept. 27, 2017). *Forecasting at scale*. e3190v2. PeerJ Inc. DOI: 10.7287/peerj.preprints.3190v2. URL: https://peerj.com/preprints/3190 (visited on 08/28/2019).

Tonin, Luke (Oct. 11, 2019). *Sequence-2-Sequence Signal Prediction*. original-date: 2018-08-31T11:19:28Z. URL: https://github.com/LukeTonin/keras-seq-2-seq-signal-prediction (visited on 10/11/2019).

van Es, Bert (Feb. 1, 2017). *Monte Carlo Markov Chain Simulation*.

Villani, Cédric (2009). *Optimal Transport: Old and New*. Grundlehren der mathematischen Wissenschaften. Berlin Heidelberg: Springer-Verlag. ISBN: 978-3-540-71049-3. URL: https://www.springer.com/gp/book/9783540710493 (visited on 09/11/2019).

Wiemer, Jeroen (July 15, 2015). *Commodity simulation in Adaptiv Analytics*.

Wolpert, D.H. and W.G. Macready (Apr. 1997). "No free lunch theorems for optimization". In: *IEEE Transactions on Evolutionary Computation* 1.1, pp. 67–82. ISSN: 1089778X. DOI: 10.1109/4235.585893. URL: http://ieeexplore.ieee.org/document/585893/ (visited on 05/13/2019).

Yoon, Jinsung (Oct. 21, 2019). *jsyoon0823/GAIN*. original-date: 2018-07-09T13:42:44Z. URL: https://github.com/jsyoon0823/GAIN (visited on 10/23/2019).

Yoon, Jinsung, James Jordon, and Mihaela van der Schaar (June 7, 2018). "GAIN: Missing Data Imputation using Generative Adversarial Nets". In: *arXiv:1806.02920 [cs, stat]*. arXiv: 1806.02920. URL: http://arxiv.org/abs/1806.02920 (visited on 06/17/2019).

Zhou, Xingyu, Zhisong Pan, Guyu Hu, Siqi Tang, and Cheng Zhao (2018). *Stock Market Prediction on High-Frequency Data Using Generative Adversarial Nets*. Mathematical Problems in Engineering. URL: https://www.hindawi.com/journals/mpe/2018/4907423/ (visited on 07/03/2019).