AALBORG UNIVERSITY MASTER'S THESIS

Predicting The EUR/USD Exchange Rate: A Deep Learning Approach

Authors:
Jonas Lund Nadj
Daniel Labuschagne Jensen
Asger Bo Rønsholdt

Supervisor:
Joachim Hagbart Madsen

Aalborg University Business School

3. Juni, 2024

AALBORG UNIVERSITY

Abstract

Faculty of Social Sciences and Humanities

Economics, MSc

Predicting The EUR/USD Exchange Rate: A Deep Learning Approach

by Jonas Lund Nadj, Daniel Labuschagne Jensen and Asger Bo Rønsholdt

This thesis leverages the Long Short-Term Memory (LSTM) neural network, known for its efficiency in time-series predictions and long-term dependencies, to forecast FOREX market directions. By integrating macroeconomic data and technical indicators into a selection of alternative LSTM models, our approach can be thought of as a starting point of employing more complex versions to more accurately predict daily deviations in the currencies pairs of the FOREX market or any other financial market. The paper presents the different building blocks necessary to understand the deep neural network framework more in-depth.

Contents

Abstract iii

1 Introduction 1
- 1.1 Problem Definition . . . 2
- 1.2 Method Section . . . 2
- 1.3 Boundary Framework . . . 3

2 Machine Learning: Theoretical Insights 5
- 2.1 Linearity . . . 5
  - 2.1.1 Linear Regression . . . 5
    - Gradient-based optimization . . . 7
- 2.2 Classification and Logistic Regression . . . 8
  - 2.2.1 Input variables . . . 8
- 2.3 Statistical Foundations of Binary Classification via Logistic Regression 8
    - Maximum Likelihood Estimation For Logistic Regression . . . 10
  - 2.3.1 Decision Boundaries . . . 11
- 2.4 Capacity of An ML Model . . . 11
- 2.5 Estimation bias and variance . . . 14
  - 2.5.1 Bias of an estimator . . . 15
  - 2.5.2 Variance of the estimator . . . 18
  - 2.5.3 Trade-off: Bias VS. Variance . . . 18
- 2.6 Maximum Likelihood Estimation . . . 20
  - 2.6.1 Conditional log-likelihood and MSE . . . 21
- 2.7 Regularization . . . 22
  - 2.7.1 Ridge regularization . . . 23
  - 2.7.2 Dropout . . . 23
- 2.8 Numerical Optimization . . . 24
  - 2.8.1 Gradient Descent . . . 24
  - 2.8.2 Stochastic Gradient Descent . . . 25
  - 2.8.3 Optimisation Algorithms with Adaptive Learning Rates . . . 28
- 2.9 Feedforward Neural Network . . . 30
  - 2.9.1 The Perceptron . . . 30
  - 2.9.2 From A Perceptron To An FNN . . . 32
  - 2.9.3 Back-propagation . . . 34
  - 2.9.4 Activation functions . . . 35
    - ReLU . . . 36
    - Sigmoid (Logistic) Activation . . . 36
    - Hyperbolic Tangent (tanh) . . . 37
    - Softmax Activation . . . 38
- 2.10 Recurrent Neural Network . . . 40
  - 2.10.1 Unfolding a Recurrent Neural Network . . . 40
  - 2.10.2 The mechanics of a Recurrent Neural Network . . . 41

2.10.3 Back-propagation Through Time 42
2.10.4 The vanishing and exploding Gradient Problem 42
2.11 The Long Short-term Network Model 43
2.11.1 The short- and long-term memories of the model 43
2.11.2 The Three Stages of the LSTM Model 45
2.11.3 The Mechanics of the LSTM Model 47
2.12 Exchange Rate Theories 50
Theories on exchange rates 50

3 Literature Review **53**

4 Data description **57**
4.1 Preprocessing and Feature Engineering 57
4.1.1 Two Pillars of Statistics 57
Correlation and Multicollinearity 58
Stationarity 58
4.1.2 Look Ahead Bias 59
4.2 The output variable - EUR/USD 59
4.2.1 EUR/USD - raw data review 59
4.2.2 EUR/USD transformations and preparation 60
4.3 Macroeconomic and fundamental data - inputs 62
4.3.1 Inflation, CPI announcements 62
4.3.2 Unemployment rate announcements 63
4.3.3 Yield curves 63
4.3.4 Short-term interest rates 63
4.3.5 Stock Indices (Euro50 and S&P500) 63
4.3.6 Volatility Index, VIX 64
4.3.7 Brent Crude Oil 64
4.4 Technical indicators - inputs 64
4.4.1 Simple Moving Average, SMA 65
4.4.2 Moving average convergence/divergence, MACD 65
4.4.3 Relative Strength Index, RSI 66

5 LSTM Predictive Trading Strategy **69**
5.1 Model Framework 69
Preparing the Data 69
Model Architecture 71
Hyperparameter Tuning 72
Performance Metric Through Several Iterations 73
5.2 Single-Layer LSTM - Model 1 75
5.2.1 The results of model 1 76
5.3 Single-Layer LSTM With Regularization - Model 2 77
5.3.1 The results of model 2 77
5.4 Two-layer LSTM - Model 3 78
5.4.1 The results of model 3 79
5.5 LSTM with restricted data - Model 4 80

6 The role of AI in future financial markets and asset pricing **83**
6.1 The evolution of prediction in financial market 83
6.2 The computational challenges of an AI future 85

7 Conclusion **87**

A Appendix **91**
A.1 Appendix A . . . 91
A.1.1 Least squares and normal equations . . . 91
A.1.2 Derivation of Decision Boundary . . . 92
A.2 Appendix B . . . 93
A.3 Appendix C . . . 94
A.4 Appendix D . . . 95
A.5 Appendix E . . . 96
A.6 Appendix F . . . 97
A.7 Appendix G . . . 98
A.8 Appendix H . . . 99

Bibliography **101**

List of Figures

2.1 Examples of different complexities on the same data points; Linear function (left), 2. degree polynomial regression (middle) and 9. degree polynomial regression (right). Own illustration. 13
2.2 Behaviour of training and testing error when capacity increases. Own illustration 14
2.3 Bias vs Capacity: Optimal Generalization Error. Own illustration . . . 19
2.4 Three types of stationary points: global minimum (*left*) and maximum (*middle*), and saddle point (*right*). Own illustration 24
2.5 Custom function with multiple local minima (shifted 150 degrees for a better viewing angle). *Custom function*: $(x^2 + y^2) * \sin(x) * \cos(y) + (x^2 + y^2)$. *Parameters:* $\epsilon = 10^{-2}$, no. steps = 20, starting coordinates $= [-2, -3]$. Own illustration 27
2.6 A perceptron (or neuron) - forward propagation process. Own illustrations 31
2.7 Linear vs non-linear function, own illustrations 32
2.8 A neural network with one hidden layer. Own illustrations 33
2.9 **Rectified Linear Activation Unit** (ReLU). Own illustration 37
2.10 Sigmoid Activation Function (logistic). Own illustration 38
2.11 Derivative of sigmoid function. Own illustration 38
2.12 The tanh activation function. Own illustration 39
2.13 Softmax activation transfer process. Example values given in text section. Own illustration 39
2.14 Sequencing system for the hidden state - own illustration. 40
2.15 Basic recurrent neural network with 3 iterations. Own illustration . . . 41
2.16 Basic recurrent neural network with 3 iterations - own illustration . . . 41
2.17 Basic recurrent neural network with 3 iterations - own illustration . . . 42
2.18 The long-term memory for one time step in an LSTM model - own illustration 43
2.19 The long-term memory for one time step in an LSTM model 44
2.20 The short-term memory and hidden state weights for one time step in an LSTM model. 45
2.21 The input variables and input weights for one time step in an LSTM model - own illustration. 45
2.22 The first stage for one time step in an LSTM model. 46
2.23 The second stage for one time step in an LSTM model. 47
2.24 The second stage for one time step in an LSTM model. 48
2.25 The second stage for one time step in an LSTM model. 48

4.1 EUR/USD Exchange rate. Daily frequency. 2005-2024. Data: FED St. Louis. Own illustration 60
4.2 Distribution of daily changes in EUR/USD. Own illustration 61

4.3 Two different threshold sets. *Left: 25th/75th percentiles. Right: 40th/60th percentile*. Own illustration . . . 62
4.4 **SMA(10) and SMA(20)**. 10 and 20 days simple moving average of EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration . . . 65
4.5 **MACD(26,12)**. 26-12 days moving average convergence/divergence of EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration . . . 66
4.6 **RSI(10)**. 10 days relative strength index of the EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration . . . 67

5.1 The architecture of model 1 . . . 75
5.2 The model loss of model 1 . . . 76
5.3 The model loss of model 2 . . . 78
5.4 The architecture of model 3 . . . 79
5.5 The model loss of model 3 . . . 80

6.1 AI-driven hedge funds versus traditional hedge funds. 2010-2016 return index. Data and illustration by Eurekahedge Investment research firm. . . . 84

A.1 **US CPI**. Month-to-month inflation rates. Data: FED St. Louis. Own illustration . . . 93
A.2 **EU CPI**. Yearly inflation rates, monthly frequency. Data: ECB. Own illustration . . . 93
A.3 **US Unemployment rates**. Data: FED St. Louis. Own illustration . . . 94
A.4 **EU unemployment rates**. Data: ECB. Own illustration . . . 94
A.5 **US Treasury Yields**. From left to right: 3m, 10y and 30y yields. 2005-2024. Data: Alpha Vantage API. Own illustrations . . . 95
A.6 **Euro Area AAA Bond Yields**. From left to right: 3m, 10y and 30y yields. 2005-2024. Data: ECB. Own illustrations . . . 95
A.7 **S&P500 Index**. 2005-2024. Data: FED St. Louis. Own illustrations . . . 96
A.8 **Euro STOXX 50 Index**. 2005-2024. Data: Google Finance. Own illustrations . . . 96
A.9 **US Volatility Index, VIX**. For S&P500. 2005-2024. Data: Cboe. Own illustrations . . . 97
A.10 **EU Volatility Index, VIX**. For EURO STOXX 50. 2005-2024. Data: STOXX. Own illustrations . . . 97
A.11 **Brent Crude Oil**. 2005-2024. Data: Alpha Vantage API. Own illustrations . . . 98
A.12 Visualization of a tensor used for the LSTM model. Own illustration . . . 99

List of Tables

5.1 Summary of model 1's performance metrics 76
5.2 Summary of model 2's performance metrics 77
5.3 Summary of model 3's performance metrics 79
5.4 Summary of model 4a's performance metrics 81
5.5 Summary of model 4b's performance metrics 81
5.6 Summary of model 4c's performance metrics 82

List of Abbreviations

Adam Adaptive Moment Estimation
AI Artificial Intelligence
API Application Programming Interface
ECB European Central Bank
EMA Exponential Moving Average
FED Federal Reserve System
FFR Federal Funds Rate
FNN Feedforward Neural Network
FOREX Foreign Exchange
IID Independent and Identically Distributed
IRP Interest Rate Parity
KL Kullback-Leibler
LSTM Long-Short-Term Memory
MACD Moving Average Convergence/Divergence
MLE Maximum Likelihood Estimation
ML Machine Learning
MRO Main Refinancing Operations
MSE Mean Squared Error
NN Neural Network
PCA Principal Component Analysis
PMF Probability Mass Function
PPP Purchasing Power Parity
RNN Recurrent Neural Network
ROC Rate of Change
RSI Relative Strength Index
SMA Simple Moving Average
SGD Stochastic Gradient Descent

Chapter 1

Introduction

In the rapidly evolving dynamics of financial markets, predicting asset prices and currency exchange rates remains a very complex issue. This is especially the case for the foreign exchange (FOREX) market, the largest and most liquid financial market to ever exist with a valuation of more than 2.4 quadrillion dollars (Beattie, 2022). Among the different currency pairs, the EUR/USD pair stands out as the most traded and arguable important relation, characterized by high volatility and significant importance to global investments, trades and businesses, and so, more accurate predictions of its movements can be of crucial importance among traders, investors, risk managers and policymakers.

Historically, predicting exchange rates (and other assets) has relied on using fundamental as well as technical data, however fundamental data involves a vast amount of data formats such as economic indicators, geopolitical events, market sentiment and generally indicators that are highly advanced to interpret numerically. But traditional models are built in ways that cannot always comprehend these advanced and complex datasets as well as the multitude of inputs needed to properly predict the behaviour of such a currency pair. This has motivated for a more complex framework that can more precisely estimate the *non-linearity* which exists in most real-world data relations.

Through the last decades, advancements in the artificial intelligence (AI) specific category, machine learning (ML) has opened new opportunities in financial forecasting. The specific branch of ML, deep learning, especially manifests its importance in capturing intricate patterns in massive datasets with multiple input features. Given the need to effectively capture long-term dependencies, the *Long-short Term Memory* (LSTM) model has gained prominence in the area of forecasting on time series. The LSTM, a special kind of recurrent neural network, proves its remarkable proficiency due to its unique ability to separate long-term dependencies from the short-term ones, and so, by leveraging the long historical exchange rate, EUR/USD, in combination with a multitude of input features with non-linear relationships, the LSTM might anticipate currency movement with better precision than traditional econometric methods. Additionally, the nature of AI allows for dynamic learning, which might enhance the modelling of financial data which is known to be a *complex adaptive system*, meaning it dynamically changes in its structural nature through time.

Recent events in the real world highlight the relevance of improving forecasting of financial markets for especially risk managers and businesses thriving to continue growth; the Covid-19 pandemic, geopolitical tensions, unconventional monetary policies and much more have all induced economic uncertainty and volatility

among financial markets - not to also mention the increase in automated trading operations, slowly excluding human beings from participating in certain markets. At the same time, the exponential growth of AI implementation, in all kinds of areas in the world, raises some serious questions and discussions regarding the ethics, privacy and compliance of AI and its rapidly increasing implementation in businesses and personal lives. There might be a need to introduce international laws governing the growth and implications of such technology.

This thesis aims to explore the application of LSTM networks in predicting the EUR/USD exchange rate through training over the last 20 years on a daily frequency. Through experimentation of multiple different features and hyper-parameters, we seek to find the optimal predictor. Furthermore, the literature has developed enough similar material to allow for a more thorough comparison of different methods, both in terms of features and models. Given the deep liquidity and complexity in this particular currency pair, we anticipate that precisely predicting the movements with consistency will be exceedingly challenging - even with the advanced utility of an LSTM.

1.1 Problem Definition

How can the LSTM neural network be utilized to forecast the direction of the EUR/USD exchange rate by drawing upon macroeconomic and technical indicators?

- What is the theoretical framework required to build an LSTM model structure that can forecast the EUR/USD exchange rate.
- What types of fundamental and technical indicators are most effective as inputs into LSTM models for FOREX market forecasting?
- Can a predictive LSTM be built to forecast the exchange rate for the EUR/USD accurately on a daily basis?
- How is the future development of AI affecting future forecasting, and are the traditional techniques used in econometrics being outperformed?

1.2 Method Section

This study is built on the theoretical foundations of ML, and in particular, deep neural networks. Since the framework for these models are advanced and arguably very heavy, the paper focuses a lot on building up the necessary theories that help understand the entire modelling process. In that way, this paper can be seen as a guidance, (both for ourselves and the reader), through the different building blocks that collectively define ML and specifically the LSTM model.

In order to achieve this deeper understanding, it is natural that the theoretical part takes up a great fraction of the paper. As a starting point when going through the ML framework, we introduce some more basic foundations that pulls from the more traditional world of statistics. This includes briefly introducing simpler mathematical areas like linearity, regressions and classification methods. Maximum likelihood is also central in the training process of neural networks, (NN), as it used to derive the loss function, and so this is also introduced. We gradually move from these statistical areas into the more NN-relevant areas of non-linearity, regularization and optimization of neural networks. Understanding these features opens the possibility to

finally build up the specific models used for the analysis. To optimally describe the final LSTM model, we gradually increase the complexity of the models; beginning from the simpler but foundational *Feed Forward Network*, (FFN), then into the *Recurrent Neural Network*, (RNN), and finally the LSTM model. This choice of method is desirable because each advancement in model is just adding extra components to the latter model, making it a natural transition.

With the theoretical understanding of the LSTM model, we can move on to the more practical parts of analysing real-world time series. Before doing this, a literature review indicates what can be expected and helps prepare what challenges might arise. After walking through the different input datasets and necessary transformations, we dive into the coding part. Here, we use Python and luckily, instead of having to build the entire code manually, there are some attractive Python packages that supports most of tasks needed, including Tensorflow, Keras, sklearn and more. Python is also the main source of coding used to produce plots and figures in the paper.

The analytical part required computation of a multitude of models. To reach results closer to the desired ones, we iterated different versions of LSTMs multiple times. This included performing grid searches to optimally choose the architecture and the hyperparameters to use in the training process. Additionally each iteration was evaluated through loss functions on both training- and validation sets. After iterating many different alternatives, we decided to finally compute four models with certain characteristics to highlight differences in results. We utilized a remote GPU server to leverage the potential for training several models.

To round things off, the paper dives into discussions and challenges regarding the development of AI in general. Here we seek to objectively highlight the ongoing competition between traditional econometric models and the more modern ML models. Furthermore, we cast a light on some of the limitations that exist in the computational aspects of AI.

The study draws upon various scientific fields. First, in the theoretical sections, we utilize properties and features from the econometric field. Second, the mathematical field is included when deriving notations to construct the ML framework. Third, insights from both micro- and macroeconomics help inform and decide the selection of inputs related to the EUR/USD exchange rate. Here, we also include some pure economic theory to further motivate for input variables.

1.3 Boundary Framework

One major limitation that we faced is related to computational time when training the different models. The LSTM models can take a very long time to train, even when running them remotely on very strong GPUs, and so, when wanting to test very small and marginal changes in hyperparameters to compare different results, training processes need to be redone entirely. This forced us to limit the complexity in terms of parameters, layers, units etc. Even these "simpler" models took a lot hours to train, and at some point, we were forced to limit further exploration. Furthermore, spending computational units also came at a monetary cost.

Another limitation was the access to data. Fetching data from 2005-2024 (preferably on a daily frequency) was a bigger challenge. Some data was purchased through Alpha Vantage API (Application Programming Interface), however the data amount

and periods was insufficient, therefore it was necessary to combine the data retrieved through the API with other open sources to complete it.

Some data initially wanted included in the data set, which was on quarterly and monthly granularities, were excluded due to their missing publication dates, potentially leading the models to suffer under look-ahead bias. Furthermore, since the target variable suffered under missing data for days which the security was not traded, because of either holidays or weekends, the corresponding observation for other variables was thus excluded as well.

Chapter 2

Machine Learning: Theoretical Insights

In the following chapter the fundamentals of the theoretical framework will be constructed. Firstly, a linear framework will be built in order to define a staring point necessary to understand an NN. This will be expanded into a section for classification and logistic regression which is crucial to later understand and interpret the models. Afterwards the chapter will focus more on the the fundamentals of an NN, beginning with the capacity, maximum likelihood estimation, regularization and optimization of system. The chapter will then construct an LSTM model starting from a simple feedforward NN through to a recurrent NN which is built into an LSTM. Finally, economic theory for the exchange rate will be set up in order to motivate an informed choice of variables.

This chapter is built mainly from two textbooks; (Lindholm et al., 2022), for establishing the main building blocks of ML, whereas (Goodfellow, Bengio, and Courville, 2016) is used both for the aforementioned, but also the introduction to deep learning techniques.

*The notation of this chapter follows (Goodfellow, Bengio, and Courville, 2016), more specifically, **scalars** are identified by lowercase variable names. **Vectors** are lowercase and in bold, such as* $\mathbf{w}$. ***Matrices** are uppercase and in bold, such as* $\mathbf{A}$.

2.1 Linearity

As in econometrics, ML builds on traditional statistical concepts, building upon them to achieve models that capture the relationships between inputs and outputs in a more meaningful way. The theoretical chapter begins with an introduction to linearity, which inherently entails *linear regression*. Linear regression is the fundamental building block for numerical prediction in ML, where *classification* being the counterpart for categorical variables. Historically, linear regression, notwithstanding its simplicity, might be the most popular method for regression tasks, and as will be shown in later sections, it is also a critical part of more advanced methods for predictions, thus making it essential before delving into these concepts.

2.1.1 Linear Regression

The goal of linear regression is to predict the true value of $y \in \mathbb{R}^n$ using an input vector $x \in \mathbb{R}^n$. Defining $\hat{y}$ as the predictions made by the model;

$$\hat{y} = \mathbf{w}^T \mathbf{x} \tag{2.1}$$

Here, $\mathbf{w}$ is a vector of parameters. The vector can be thought of as a coefficient multiplied on the inputs, $\mathbf{x}$. To cement this idea even more, one can imagine them being *weights*. This is a convenient structure since our multiplication involves *transposing* w, so in essence, the equation is multiplying each *feature* from the input vector x by a corresponding weight, indicating its importance to the system of linear equations.

Having defined the target for predictions, a performance measure is needed, in order to evaluate it; this is where *mean squared error*, (MSE), is utilised. Suppose one had a *design matrix*, referring to a matrix of m example inputs of explanatory variables. These examples will not be used for training but solely for evaluating the model's performance. Secondly, we possess a vector of regression targets with the true value of y for each of these m examples. Since this matrix and vector will not be used for training, it will be referred to as the test set and denoting the two as $\mathbf{X}^{(test)}$ and $\hat{\mathbf{y}}^{(test)}$ for the design matrix and vector of regression targets, respectively.

$$MSE_{test} = \frac{1}{m} \sum_i (\hat{\mathbf{y}}^{(test)} - \mathbf{y}^{(test)})_i^2 \tag{2.2}$$

From equation 2.2, it shows that if, the model fits well then, $\hat{\mathbf{y}}^{(test)} \approx \mathbf{y}^{(test)}$, so the MSE should take on as small a value as possible. Equation 2.2 is also what is referred to as a *loss* - or *cost function*. The terms are analogous, but the former is for a single example and the latter is for the entire test set.

Since these are vectors, they can be rewritten equation 2.2 to the following;

$$MSE_{test} = \frac{1}{m} \mid\mid \hat{\mathbf{y}}^{(test)} - \mathbf{y}^{(test)} \mid\mid_2^2 \tag{2.3}$$

Rewriting equation 2.2 to equation 2.3, provides both clarity in terms of mathematical notation and the intuition of computing the difference between two vectors in a much more compact way. Here, $\mid\mid \cdot \mid\mid_2$ denotes the *Euclidean distance*, and $\mid\mid \cdot \mid\mid_2^2$ its square. We use this specific measure to quantify the error between actual and predicted values in ML.

$$\mid\mid \mathbf{x} \mid\mid_2 := \sqrt{\sum_{i=1}^{n} x_i^2} \tag{2.4}$$

$$= \sqrt{\mathbf{x}^T \mathbf{x}} \tag{2.5}$$

Equation 2.4 calculates the *Euclidean distance* of $\mathbf{x}$ from the origin in *Euclidean space*. The *Euclidean norm* is often referred to as the ℓ_2-norm, since in later chapters a specific method named ℓ_2-norm, will be introduced, this term will be refrained from use, when referring to the Euclidean norm, (Deisenroth, Faisal, and Ong, 2020). An important question comes to mind when rewriting from summation to the Euclidean distance; why is it needed? i) Many programming libraries for ML are optimized for linear algebra notation, thereby decreasing computation time. ii) As later chapters

will show; it allows the models to reach numerical stability using different optimization strategies. iii) The linear algebra notation is a simple abstraction that allows for more clarity in the expression. iv) The Euclidean distance captures the idea of the distance between actual and predicted values in n-dimensional space, therefore making its geometric interpretation strong.

Having defined both a task and a performance measure for this simple linear regression model, it is now possible to investigate how it is possible to train such a model and optimise it, so as to, reduce MSE. Again, regurgitating our goal; change the parameters of w, to reduce MSE, by allowing the algorithm to discover the training set of, $\mathbf{X}^{(train)}$ and $\mathbf{y}^{(train)}$. The method for doing so is to minimize the MSE on the training set, MSE_{train}. At a later stage, interestingly, it will be shown that minimizing the MSE is equivalent to maximizing the *log-likelihood*.

Gradient-based optimization

In order to minimize MSE_{train} one can simply solve for where its *gradient* is equal to 0, but before going through with this derivation, *optimisation* will be motivated. Optimisation refers to the task of *minimising* or *maximising* some function. The function which is optimised is what is referred to as either the *objective function*, *loss function* or the *cost function*. Specifically this entails computing the *gradient* wrt. the parameter that is optimised on, such as the weight vector mentioned previously in equation 2.1 and setting it equal to 0, this is also referred to as the *zero-gradient*. The motivation for this method encapsulates the fact that the gradient computes the partial derivative of the objective function for each of the entries in the weight vector, this idea is represented in equation 2.6.

$$\nabla_{\mathbf{w}} MSE_{train} = \begin{bmatrix} \frac{\partial MSE_{train}}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial MSE_{train}}{\partial \mathbf{w}_i} \end{bmatrix} \tag{2.6}$$

The partial derivative is especially useful since it effectively shows how to change $\mathbf{w}$ in order to make a small improvement in the prediction. This computation allows the model to find a stationary point, but as will be shown later in Section 2.8, this stationary point is not necessarily the minima of the function, there can exist many local minima, and the preferred solution would yield a global minima. Therefore, Section 2.8 will introduce *numerical optimization*, which iteratively solves it. This derivation will abstract from this algorithm, for now, and focus on the more simple process. The derivation of the linear regression example springs out from Equation 2.3 and setting its gradient wrt. $\mathbf{w}$ equal to 0 and is showed in Appendix A. Having presented the simplest model in ML, the next section will turn to classification, which is the counterpart to regression and handles task in an entirely different manner allowing for predictions that solve a whole new plethora of problems.

2.2 Classification and Logistic Regression

In order to motivate specific structures of the later, more advanced models, it is necessary to provide a formal introduction to *logistic regression* and *classification*. This necessity stems from the idea of probability theory, but also, more importantly, the processing of binary variables in a probabilistic manner, which is the counterpart to regression, namely *classification*.

This section is based upon the book (Lindholm et al., 2022). The notation will differ slightly from previous and future sections, but the results will not vary from any other textbook on the subject.

2.2.1 Input variables

So far, for the linear regression case, it has been formulated from the standpoint that the inputs were numerical, either discrete or continuous. This may not be the case for all datasets; there are also inputs that have no natural ordering, such as labels or names. Even though the inputs are formatted in such a way, one can augment them through techniques such as *one-hot encoding*. To provide a formal example, using a dataset consisting of companies listed on NASDAQ, one can divide each company into a corresponding sector, such as technology, medicine, finance, etc., and then construct a binary column for each sector available in the dataset. The binary variable is activated if the sector corresponding to the correct company is present; by activated is meant a value of 1 and 0 elsewhere. This transformation allows a computer to process the information contained inside these string variables and consequently create a useful model.

2.3 Statistical Foundations of Binary Classification via Logistic Regression

In the realm of statistical modeling, classification tasks involve predicting a categorical dependent variable using a set of predictors. Among these tasks, binary classification is particularly significant due to its widespread applicability across various fields. This section discusses the statistical framework of logistic regression for binary classification.

$$p(y = m|\mathbf{x}) \tag{2.7}$$

Specifically, Equation 2.7 describes the probability for class m conditional on the information $\mathbf{x}$. The probability described in Equation 2.7 is implicitly treating y as a random variable; this terminology stems from the fact that the model is a reflection of the real world from which the data was generated; this inherently involves a degree of randomness. An equivalent to this would be the error term ϵ from regression, which is also known from standard econometric theory. The aim of this classification model is then not only to predict the label of the class but, moreover, to predict the class probabilities. Drawing a simple example of binary classification, we have that $M = 2$, where y is limited to two distinct values: 1 or -1. Assuming the distribution of the data generation follows a *centered Bernouli distribution*, the learned model, which we will call $g(x)$, can be learned through:

$$y = \begin{cases} 1 & \text{with probability} \quad p(y=1|\mathbf{x}) \quad \textbf{(class 1)} \\ -1 & \text{with probability} \quad 1 - p(y=1|\mathbf{x} \quad \textbf{(class 2)} \end{cases} \tag{2.8}$$

Where class 1 is modelled by $g(\mathbf{x})$, and class 2 is modelled by $1 - g(\mathbf{x})$.

Since probabilities are limited to the interval of $[0, 1]$, we have the following relationship:

$$p(y=1|\mathbf{x}) + p(y=-1|\mathbf{x}) = 1 \tag{2.9}$$

From these facts, we can also derive its probability density function (PDF), as follows:

$$p(y|\mathbf{x}) = p(y=1|\mathbf{x})^{0.5(1+y)}[1 - p(y|\mathbf{x})]^{0.5(1-y)} \tag{2.10}$$

In its current form, it is possible to apply linear regression, but that does not inherently yield probabilities that fall within the predefined range; instead, linear regression can take on values on the entire real line. In order to force the values to fall into this range, there is a need for a function that can *normalize* the values. Using the linear regression from earlier, in equation 2.1, which was on the form $\mathbf{X}^{(train)}\mathbf{w}$, in this section it will be described as $\boldsymbol{\theta}^T\mathbf{x}$. Defining that expression and using the *logistic function* yields the following setup:

$$z = \boldsymbol{\theta}^T\mathbf{x} \tag{2.11}$$

$$p(y=1|\mathbf{x};\boldsymbol{\theta}) = \frac{\exp(z)}{1+\exp(z)} \tag{2.12}$$

Finally inserting z into equation 2.12:

$$p(y=1|\mathbf{x};\boldsymbol{\theta}) = \frac{\exp(\boldsymbol{\theta}^T\mathbf{x})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x})} \tag{2.13}$$

Equation 2.13 describes how to obtain label predictions and probabilities for class 1, but conversely, the same method applies to class 2, represented by $1 - g(\mathbf{x})$:

$$1 - g(\mathbf{x}) = 1 - \frac{\exp(\boldsymbol{\theta}^T\mathbf{x})}{1+\exp(\boldsymbol{\theta}^T\mathbf{x})} \tag{2.14}$$

$$= \frac{1}{1+\exp\boldsymbol{\theta}^T\mathbf{x}} \tag{2.15}$$

$$= \frac{\exp(-\boldsymbol{\theta}^T\mathbf{x})}{1+\exp(-\boldsymbol{\theta}^T\mathbf{x})} \tag{2.16}$$

As shown in the above equations, logistic regression amounts to linear regression augmented with the logistic function, but even though they share many similarities, logistic regression is a method applied to classification problems and not regression problems. Another observation is made from the above result: the parameter $\boldsymbol{\theta}$ is

also present for this method, just as in linear regression. Logistic regression is therefore referred to as a parametric model in the linear regression case. The model is still missing a method for learning the optimal parameters given a specific training set: $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$. As the next section will show, this is not done through the same method as for the linear regression case but will instead draw upon *maximum likelihood* in order for the model to learn the parameters.

Maximum Likelihood Estimation For Logistic Regression

Since the logistic function introduces non-linearity into the model, it is not possible to model logistic regression through the normal equations presented in Section 2.1.1. Drawing back to the assumption of the distribution when setting up equation 2.8, a centred Bernouli distribution was implied. This direct assumption of a distribution fits the maximum likelihood approach since it implicitly draws upon an assumption of the distribution. Maximum likelihood makes for an interesting and suitable approach, not only because of the previously stated point but also, more importantly, because the distribution chosen has to increase the likelihood of observing our data.

$$\hat{\boldsymbol{\theta}} = \max_{\boldsymbol{\theta}} p(\boldsymbol{y}|\boldsymbol{X};\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \ln p(y_i|\boldsymbol{x}_i;\boldsymbol{\theta}) \tag{2.17}$$

Unpacking equation 2.17 is essential to understanding where this approach springs from. From a maximum likelihood perspective, it is written in terms of the product of the probabilities, but converting this to summation using the logarithm is crucial for two reasons: i) computing the product of probabilities is computationally heavy, especially as the number of observations increases; and ii) the logarithm is a monotonic transformation, meaning that the ordering of the numbers. The logarithm also transforms the optimisation problem into a convex one. Convex problems are easier to solve using numerical optimisation strategies, which will be presented in Section 2.8, since they guarantee that any local maximum or minimum is also a global one. Remembering that the model $p(y = 1|\mathbf{x};\boldsymbol{\theta}$ is $g(\mathbf{x})$, then equation 2.17 transforms it to;

$$\ln p(y_i|\mathbf{x}_i;\boldsymbol{\theta} = \begin{cases} \text{if} \quad y_i = 1 & \ln g(\mathbf{x}_i;\boldsymbol{\theta} \\ \text{if} \quad y_i = -1 & (1 - g(\mathbf{x}_i;\boldsymbol{\theta})) \end{cases} \tag{2.18}$$

Commonly, equation 2.18 is transformed into a minimization problem using the negative log-likelihood as a cost function. $J(\boldsymbol{\theta}) = -n^{-1} \sum \ln p(y_i|\mathbf{x}_i;\boldsymbol{\theta})$;

$$J(\boldsymbol{\theta}) = n^{-1} \sum_{i=1}^{n} \underbrace{\begin{cases} \text{if} \quad y_i = 1 & -\ln g(\mathbf{x}_i;\boldsymbol{\theta}) \\ \text{if} \quad y_i = -1 & -\ln(1 - g(\mathbf{x}_i;\boldsymbol{\theta}) \end{cases}}_{\text{Binary cross-entropy loss}, L(g(\mathbf{x}_i;\boldsymbol{\theta}), y_i)} \tag{2.19}$$

The above equation is referred to as the *binary cross-entropy loss*, it is not akin to logistic regression, but can be applied to any binary classifier that predicts class probabilities. One can also derive a loss function specific for logistic regression, but the specific loss function and model will is not applied or essential for the understanding of further section, so therefore it is not within the scope of this thesis to derive it.

The next subsection will provide a high-level description of decision boundaries as the logical next step for classification.

2.3.1 Decision Boundaries

Insofar, logistic regression and classification has been explained as a method for predicting class probabilities, but at times, it can be beneficial to make a concrete prediction for the test input. Predicting if it belongs to either class in the binary case. The most common approach for this is to let the prediction be *the most probable class*, this can be formulated as follows:

$$\hat{y}(\mathbf{X}^{(train)} = \begin{cases} 1 & \text{if} \quad g(\mathbf{X}^{(train)}) > r \\ -1 & \text{if} \quad g(\mathbf{X}^{(train)}) < r \end{cases} \tag{2.20}$$

In Equation 2.20 the parameter r is introduced to act as a decision threshold, in binary classification, this value is most commonly set to 0.5. The derivation for the decision boundary of a binary classification is done by solving the following:

$$g(\mathbf{X}^{(train)}) = 1 - g(\mathbf{X}^{(train)}) \tag{2.21}$$

The derivation is done in Appendix A.1.2 and show that the decision boundary for binary classification is a *linear hyperplane*. A linear hyperplane is a generalization of a Cartesian plane (2D) in three-dimensional space. Thus showing how the model partitions the feature space using a linear classifier. Why should one care about decision boundaries? i) It can determine whether the model has learned the correct patterns from the data, this is especially the case for sparse data. ii) Increasing the feature space to the n-dimensional plane, it is beneficial to have an understanding of how the model splits the data between classes can be very beneficial to understanding these much more complex models, which require, not linear, but non-linear separations. In later Sections, it will be shown that the model can predict, not only two, but three or more classes using specific function transformations.

2.4 Capacity of An ML Model

In the following section the term *capacity* for a machine learning model will be explained. The section provides parts of the understanding for how an NN fits multiple inputs to an output value. Furthermore, it explains the pitfalls in regards to under- and overfitting. This will later become a central discussion in the modelling section as the models are evaluated until the optimal capacity and complexity are found.

One of the central objectives with training a machine learning model is that it should perform well on new and unseen data. To do this, it must avoid falling into the pitfalls of modelling; *under-* or *overfitting* the data. Finding a balance between the two can pose a significant challenge.

From econometric models all the data is used in the training of the model, however, when constructing an ML model this is not the case. When building an ML model, the data is split into *training* and *testing* sets. The training set is usually set to 80% of the total dataset and is divided into an input training set, noted x_{train}, for the input

variables and an output training set that has the output variable, noted y_{train}. The error for the training set is referred to as the training error, noted $error_{train}$. For the testing set, the size of the dataset is typically 20% and it is divided into an input testing set, noted x_{test}, for the input variables and an output testing set that has the output variable, noted y_{test}. The error for the testing set is referred to as the testing error, noted $error_{test}$. The testing error is also referred to as the generalisation error.

In linear regression a look was taken at minimising MSE in Section 2.1.1. However, when having a training- and testing error both should be evaluated. Evaluating these has the same formula except with different data inputs; from the training- and the testing set:

$$error_{train} = \frac{1}{m^{(train)}} ||\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)}|_2^2 \tag{2.22}$$

$$error_{test} = \frac{1}{m^{(test)}} ||\mathbf{X}^{(test)}\mathbf{w} - \mathbf{y}^{(test)}|_2^2 \tag{2.23}$$

Further, exploring the difference between the training and test data one can look at the data-generation process, which is the process of how the data is being generated in the real world and some assumptions has to be made, these are called the *independent and identically distributed* (I.I.D) assumptions. It states that the test and training data have to be independent from each other and have the same underlying distribution. The underlying distribution is called data-generating distribution, P_{data}. This enables one to study the relationship between the training and testing error. As both data sets were drawn from the same underlying distribution that has the same data-generation process, this enables the expected training error for a randomly selected model to be equal to the expected testing error. Thereby when evaluating the testing error, it can be evaluated relative to the training error as they follow the same distributions and hence, they are comparable.

Having the relationship between the training and testing error in mind, making a great model requires two factors; ensuring the training error is low and that the gap between the training and testing error is as small as possible, which is noted as the *generalisation gap*. This again leads back to the pitfalls of modelling; underfitting and overfitting. The model is underfitting the data if the training error is high which means it is not able to capture the underlying pattern sufficiently during training. If the model is overfitting, the training error may be low but the gap between the training and test error is high which tells that the model does not perform well on new unseen data. To control if a model is under- or overfitting, the *capacity* of the model can be adjusted. Capacity describes the complexity of the model, which results in higher capacity having more complex functions that increases the training time of the model and changes both the training and testing errors.

To study how increasing complexity affects the error, three models with different complexities will be shown on the same dataset. The dataset is generated from a 2nd degree polynomial function and will therefore resemble a curve. The exact function that generates the data points is the following:

$$y = -x^2 + 2x + 3 \tag{2.24}$$

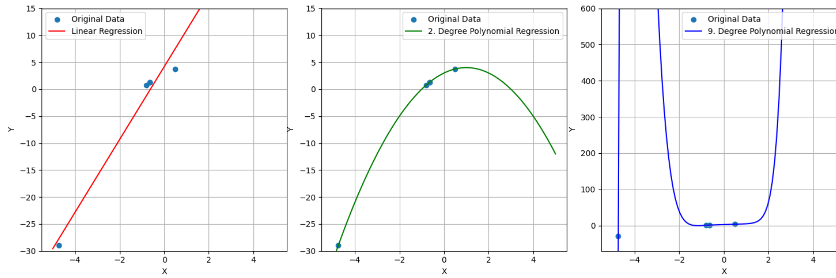

FIGURE 2.1: Examples of different complexities on the same data points; Linear function (left), 2. degree polynomial regression (middle) and 9. degree polynomial regression (right). Own illustration.

The different models with three different complexities are graphically presented in figure 2.1.

The first model is a standard linear regression model:

$$\hat{y} = b + w_1 x \tag{2.25}$$

It can be observed that the model oversimplifies the data and as a result the training error is higher. It does not fit well on the data and will not fit well on the new data either.

In the next model; a second term is added to create a 2nd degree polynomial function so a more precise model can be estimated:

$$\hat{y} = b + w_1 x + w_2 x^2 \tag{2.26}$$

The 2nd degree polynomial regression shows a perfect fit - and since the form of the data is known as a 2nd degree polynomial function the model will also fit close to perfectly on new data as well. In the real world, one cannot precisely know the exact distribution which generates the data, and so, it becomes harder to find the precise complexity to apply.

In the third model; the complexity is further increased to a 9th degree polynomial regression to observe the issue with excessive complex models:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i \tag{2.27}$$

In the graph for the 9th degree regression the model fits well to the data points given, however, as the data generating process is a second degree polynomial function, the area between the data points reveals a critical flaw in the model. The model might fit nearly "too" perfectly on the training data but will fit poorly on new data.

In conclusion, it is clear that, generally, an ML model will perform at its peak when it has the appropriate capacity, ie. is nearest to the true complexity of the data generating process. However, it can be hard to find the exact appropriate complexity. Firstly, it is not desirable to have a large training error, but if an overly complex model overfits the data and the testing error is too high the models applicability would diminish. However, a model's capacity is not only limited by the complexity of functions and number of variables which are called the input features. When reducing the training error, the set of functions that can be used are set by the complexity and

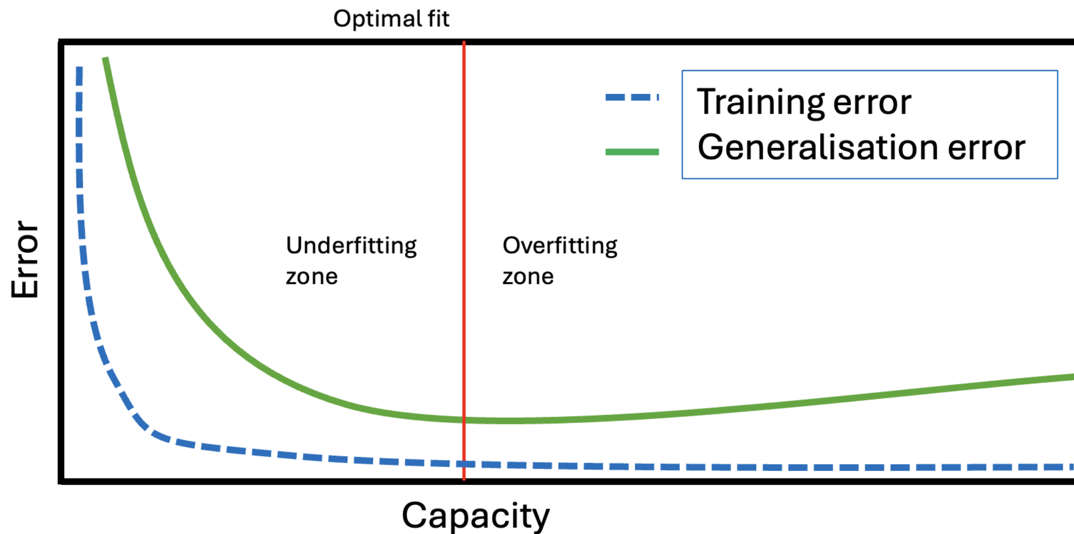

FIGURE 2.2: Behaviour of training and testing error when capacity increases. Own illustration

utilising the perfect fit between these input features is called the representational capacity. However, in practice finding the correct set of input features is a difficult optimisation problem. Thereby when considering the imperfection of different optimisation algorithms, an ML model will only have the effective capacity which is limited by the optimization algorithm and is lower than the representational capacity.

The optimal capacity is therefore what a good ML model is optimizing. The ideal model has a low training error and a small gap between training and testing error. When looking at the trade off between training error and testing error it is useful to look at the behaviors of the different errors. Typically, the training error decreases with increasing capacity and the testing error has a U-shaped curve, initially decreasing and eventually increasing. This is illustrated in figure 2.2.

From figure 2.2 it can be observed that at very low levels of capacity, both types of errors are high, however, when capacity is increased towards the optimum the errors are drastically lowering. When the optimal capacity is exceeded the training error will continue to decrease and the testing error will begin to increase once again. It should be noted that it is still possible to have the optimal model and still have a high gap between training and testing error. In this case, increasing the number of training examples could further assist in decreasing the training error.

2.5 Estimation bias and variance

The following part introduces, very briefly, some central statistical terms and properties that are rather important in understanding how a machine learning algorithm operates. The terms and properties are quite standard knowledge in the statistical world, and therefore only a brief and shallow walk through will be made. The following will be mentioned; estimation bias, estimation variance and estimation consistency.

2.5.1 Bias of an estimator

In both statistics and ML, the term *bias* carries a central role when dealing with estimations. In machine learning especially, the bias is very widely used, and so, it is necessary to briefly introduce the definition and properties of bias and unbiasedness. This is done in a rather simple manner using well-known distributions as examples.

First, a mathematical notation of the bias of an estimator, $\hat{\theta}$ can be expressed as:

$$bias(\hat{\theta}_m) = \mathbb{E}(\hat{\theta}_m) - \theta, \tag{2.28}$$

where the expectation is over some sample data and the true underlying value generating the entire data is θ. The estimator is *unbiased* if $bias(\hat{\theta}_m) = 0$. Likewise, taking the expectations of the estimator should yield $\mathbb{E}(\hat{\theta}_m) = \theta$. Furthermore, the estimator is *asymptotically unbiased* if $lim_{m\to\infty} bias(\hat{\theta}) = 0$. In other words, the estimator becomes unbiased as the size of the sample set increases towards infinity. Any unbiased estimator will always also be asymptotically unbiased, however, it is also possible for *biased* estimators to become unbiased as sample size increases, hence they are asymptotically biased.

The following provides two examples calculating whether an estimator is unbiased or biased. First, with a Bernoulli distribution, then a Gaussian distribution.

The Bernoulli Distribution

Let there be a set of samples: $(x^{(1)}, x^{(2)}, x^{(3)}, .., x^{(m)})$ following a Bernoulli distribution with mean θ:

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1-\theta)^{(1-x^{(i)})}, \tag{2.29}$$

where the equation represents the probability mass function *(PMF)* and θ is the success parameter determining the probability that the outcome of $x^{(i)}$ is 1.

A standard way of estimating θ is to compute the mean of the training data:

$$\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \tag{2.30}$$

It can be shown whether the estimator of this mean parameter is unbiased by substituting equation 2.30 into the formula for bias in equation 2.28 and reducing:

$$
\begin{aligned}
bias(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta && (2.31)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta && (2.32)\\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[x^{(i)}] - \theta && (2.33)\\
&= \frac{1}{m}\sum_{i=1}^{m}\sum_{x^{(1)}=0}^{1}(x^{(i)}\theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}) - \theta && (2.34)\\
&= \frac{1}{m}\sum_{i=1}^{m}(\theta) - \theta && (2.35)\\
&= \theta - \theta = 0 && (2.36)
\end{aligned}
$$

It is now clearly seen that $bias(\hat{\theta}) = 0$, and so, $\hat{\theta}$ is an unbiased estimator of the true underlying value of θ.

The Gaussian Distribution

Showing whether an estimator unbiased in the example of a Gaussian distribution is very similar to that of the Bernoulli distribution. Again, consider a set of samples: $(x^{(1)}, x^{(2)}, xx^{(3)}, .., x^{(m)})$ that are i.i.d.. Following a Gaussian distribution, it is given that they follow: $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$. The probability density function, *(PDF)*, of a Gaussian distribution can be show to be given by:

$$
p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{1}{2}\frac{(x^{(i)} - \mu)^2}{\sigma^2}) \qquad (2.37)
$$

Estimating the mean parameter of this distribution is done similarly to before by taking the mean of the training data sample set:

$$
\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \qquad (2.38)
$$

In determining the bias of this estimation, the expectation of equation 2.38 is, as done previously, compared with the actual population mean, μ, by substituting the equation into the bias equation:

$$
\begin{aligned}
bias(\hat{\mu}_m) &= E[\hat{\mu}_m] - \mu && (2.39)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] && (2.40)\\
&= (\frac{1}{m}\sum_{i=1}^{m}\mathbb{E}[x^{(i)}]) - \mu && (2.41)\\
&= (\frac{1}{m}\sum_{i=1}^{m}\mu) - \mu && (2.42)\\
&= \mu - \mu && (2.43)
\end{aligned}
$$

And once again, it is shown that the estimator of this mean parameter is unbiased.

Estimator for variance, Gaussian distribution

It can also be shown whether an estimator for the *variance* of some data following a Gaussian distribution is biased or not. This is done by first providing an example of a biased estimator and thereafter an unbiased estimator.

First, the variance, σ^2, is estimated through the following:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2 \tag{2.44}$$

where $\hat{\mu}_m$ is the sample mean. Note that the above equation simply represents the *sample variance*. The question at hand, though, is if this is also an unbiased representation of the true *population variance*. Once again, substituting equation 2.44 into the formula for the bias:

$$bias(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2, \tag{2.45}$$

and reducing, shows whether the estimator is unbiased:

$$bias(\hat{\sigma}_m^2) = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] - \sigma^2 \tag{2.46}$$

$$bias(\hat{\sigma}_m^2) = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] - \sigma^2 \tag{2.47}$$

$$= \frac{m-1}{m}\sigma^2 - \sigma^2. \tag{2.48}$$

As seen, the bias differs from 0, hence the sample variance is a biased estimator. Now, if a small configuration is done, the unbiased sample variance estimator is given by:

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2 \tag{2.49}$$

If the expectation of this equation is substituted into the bias formula, it is shown to be an unbiased estimator:

$$
\begin{aligned}
bias(\tilde{\sigma}_m^2) &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}(x^{(i)} - \hat{\mu}_m)^2\right] - \sigma^2 && (2.50)\\
&= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] - \sigma^2 && (2.51)\\
&= \frac{m}{m-1}(\frac{m-1}{m}\sigma^2) - \sigma^2 && (2.52)\\
&= \sigma^2 - \sigma^2 = 0 && (2.53)
\end{aligned}
$$

.

Now, what is the point of demonstrating both the biased and unbiased estimators? In machine learning, and in particular deep neural networks, biased estimators are sometimes preferred in that they carry properties that might be important. While unbiased estimators will always be desirable, sometimes a biased estimator is trained to be better performing, and hence, it is likely that those will be used frequently in later practical modelling.

2.5.2 Variance of the estimator

Just as the estimator can carry a bias, it also possesses a degree of variation through the sample set it is estimated from. The variance of an estimator, $VAR(\hat{\theta})$ is a measure of how much the estimator will vary as it is *independently* re-estimated from samples of the same training dataset. While keeping this low is very desirable, just like the bias, it cannot always be the case, and the two often behave as trade-offs to each other.

The degree of variation that the estimator will have is something that can be quantified, similarly to that of the bias. Taking the square root of the variance of e.g. the mean, the standard error of the mean is obtained and is given by:

$$
SE(\hat{\mu}_m) = \sqrt{Var\left[\frac{1}{m}\sum_{i=1}^{m}x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}. \qquad (2.54)
$$

This term becomes quite important in machine learning as it is used in the process of quantifying the *generalization error* that a model is expected to have in the test set. The generalization error is often estimated as the sample mean of the error on the test set, and naturally, as more examples are included in the test set, this estimate becomes more accurate. The fact that the variance of an estimator decreases with the amount of examples, m, included can easily be shown and is also important when later discussing *consistency*.

2.5.3 Trade-off: Bias VS. Variance

As briefly mentioned, the bias and variance of estimators are often seen to be related to a trade-off optimization. While both are preferred to be minimized, reducing one very likely increases the other and vice versa. The question is then; how does one optimally negotiate this trade-off? A commonly used method is to compute the **MSE** of the estimates:

$$MSE = \mathbb{E}\left[(\hat{\theta}_m - \theta)^2\right] \tag{2.55}$$
$$= bias(\hat{\theta}_m)^2 + Var(\hat{\theta}_m) \tag{2.56}$$

Keeping **MSE** optimally low requires that estimators do not deviate too much from the observed values of a point or function. Clearly from the above, this requires that both bias and variance are kept in check. As also mentioned in relation to model capacity earlier, bias and variance link closely to the concept of over- and underfitting. This is illustrated in figure **??**. Assuming that model capacity is on the x-axis, then increasing capacity often tends to decrease bias but increase variance. Intuitively, this makes great sense. A too "simple" function estimation (e.g. a linear model) is not expected to vary a lot when re-estimating on the same dataset, but it most likely wont fit observed values well, hence it will carry a low variance but a large bias. Then, increasing the capacity of the function estimation, it will likely fit a given training data well, but when re-estimating it on slightly different observations in the same dataset, it will change a lot in order to fit well again, and hence, the bias will be small but the variance will be large. As seen in the figure, the optimal trade-off is at the point of capacity, where the sum of the bias and variance are minimized - in other words; the expected generalization error will be minimized.

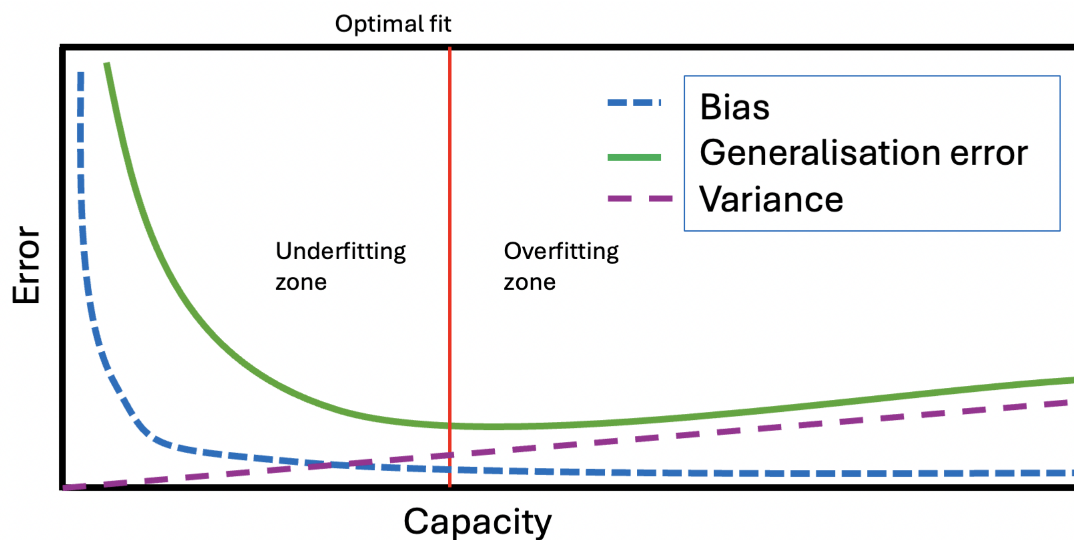

FIGURE 2.3: Bias vs Capacity: Optimal Generalization Error. Own illustration

Lastly, but also importantly, consistency in the data is also very desirable, i.e $plim_{m\to\infty}\hat{\theta}_m = \theta$. This indicates that as the training set grows in examples, the estimator is expected to converge towards the true parameter/function value. Consistency ensures that the bias decreases which, as mentioned before, is very preferred, though not necessary.

2.6 Maximum Likelihood Estimation

Up until this point, estimators have been described together with some of their properties. However, starting from zero, one knows very little about how exactly those estimators are chosen just based off of some analysis of their properties. Simply assuming that an estimator is good and then analysing the corresponding bias and variance of it to decide whether to use it or re-estimate is a rather suboptimal method. Instead, it is possible to follow specific *a priori* principles that help ensure that estimators will indeed be qualified as consistent and efficient. One principle, and arguably the most used one, to apply is the *maximum likelihood estimation* (MLE) principle. For that reason, it is necessary to briefly introduce the concept of MLE and some of the properties that make it desirable in the context of machine learning. MLE is very central in relation to supervised learning algorithms later as it helps determine the loss function used when testing performance metrics of the model. Some of the concepts behind MLE was already used in section 2.3, however this section walks through the foundational properties once again.

First, let a sample consisting of examples be given: $\mathbb{X} = \{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ and let those examples follow the true population distribution $p_{data}(\mathbf{x})$, which is unknown. Assuming a parametric model, $p_{model}(\mathbf{x};\boldsymbol{\theta})$, that estimates the true probability distribution of $p_{data}(\mathbf{x})$, the maximum likelihood estimator for $\boldsymbol{\theta}$ is given by:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p_{model}(\mathbb{X};\boldsymbol{\theta}), \tag{2.57}$$

where *argmax* is an argument to find the maximum of a given function - in this case, it finds the estimator, $\boldsymbol{\theta}$, that maximizes the probability of the model, $p_{model}(\mathbb{X};\boldsymbol{\theta})$, to fit the underlying true data-generating process. Remembering that $\mathbb{X}$ is a set of all m examples, the above can be written in terms of the product of the probabilities of all examples, $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}$:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)};\boldsymbol{\theta}). \tag{2.58}$$

Intuitively, the above takes the product of probabilities of all examples and finds the maximum of that. But taking the product of multiple probabilities, all of which are below 1, the number becomes astronomically small - and this becomes more and more evident the more examples are brought into the equation. The number, in fact, becomes so small and close to 0, that most machines cannot even compute it. This computational inconvenience is known as *numerical underflow*. This is dealt efficiently with by taking the logarithm of the likelihood, as this does not change the *argmax*, but changes a product into a sum:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{m} log\, p_{model}(\mathbf{x}^{(i)};\boldsymbol{\theta}). \tag{2.59}$$

Again, scaling the expression does not change the *argmax* function, and so, the above can conveniently be divided by m to rewrite the sum into an expectation w.r.t. the distribution of the training set:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}}\, p_{model}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \tag{2.60}$$

Equation 2.60 defines most intuitively the maximization problem at hand; obtaining the maximum expected log-likelihood of the model distribution trained to fit the true data-generating process of the empirical data. Alternatively, another way of interpreting MLE is to view it as *minimizing* the deviation between the model distribution and the empirical data distribution, $\hat{p}_{data}$. The dissimilarity between two distributions can be measured according to the *Kullback–Leibler* (KL) divergence. In this case, the KL divergence is given by:

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}}\left[log\, \hat{p}_{data}(\mathbf{x}) - log\, p_{model}(\mathbf{x})\right]. \tag{2.61}$$

The first term in the expectations bracket, $log\, \hat{p}_{data}(\mathbf{x})$, is just the data-generating process of the training data, and hence not a model, which means it cannot really be changed as it is observed data observations. This implies that when training the machine learning model to minimize the KL divergence, it is simply minimizing the last term:

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{data}}\left[log\, p_{model}(\mathbf{x})\right], \tag{2.62}$$

which is the *negative* log-likelihood. It is quite obvious that minimizing the negative log-likelihood is exactly the same as maximizing the log-likelihood as in equation 2.60. The approach of maximum likelihood estimation through minimizing the KL divergence is convenient in that the minimum value of KL-divergence is known; zero. This minimum limitation makes the minimization process more simple.

2.6.1 Conditional log-likelihood and MSE

To bring the MLE principles closer to the context of machine learning algorithms, one can generalize MLE into a *conditional* log-likelihood where some vector of output, $\mathbf{y}$, is predicted given a vector of inputs, $\mathbf{x}$ - and in probability terms; $P(\mathbf{y}|\mathbf{x};\boldsymbol{\theta})$. This specification most often forms the foundation of supervised learning algorithms in machine learning, which will later be described. The conditional maximum likelihood estimation of $\boldsymbol{\theta}$ can then be given by:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, P(\mathbf{Y}|\mathbf{X};\boldsymbol{\theta}) \tag{2.63}$$

And like earlier, by decomposing it into a product probability of all training examples and taking the logarithm to convert the product to a sum:

$$\boldsymbol{\theta}_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{m} log\, P(\mathbf{y}^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}) \tag{2.64}$$

To justify the use of maximum likelihood estimation, a simple example can show that using MLE to estimate a linear regression yields the same parameters for the weights, $\mathbf{w}$ as minimizing the mean squared error w.r.t the weights. This argument

can then be expanded to justify the use of MLE in most training data sets. Furthermore, some desirable properties follow the estimators of maximum likelihood, which will be discovered.

Until now, MSE has been used to configure the mapping between an input, $\mathbf{x}$ to a single prediction, $\hat{y}$. With maximum likelihood the procedure is changed so that the input creates a conditional distribution, $p(y|\mathbf{x})$. The log-likelihood of this distribution is:

$$\sum_{i=1}^{m} log\, P(y^{(i)}|\mathbf{x}^{(i)};\boldsymbol{\theta}) \tag{2.65}$$

Note here, that MLE is taken on a Gaussian distribution, and so it can be shown that taking the log-likelihood of equation 2.65, and hence the probability density function of a Gaussian distribution, the above can be expressed as:

$$-m\, log\, \sigma - \frac{m}{2} log(2\pi) - \sum_{i=1}^{m} \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}. \tag{2.66}$$

Important to note is that the variance σ^2 is fixed and given beforehand. The log-likelihood can be compared to the mean squared error:

$$MSE_{train} = \frac{1}{m} \sum_{i=1}^{m} \|\hat{y}^{(i)} - y^{(i)}\|^2, \tag{2.67}$$

to show that maximizing the log-likelihood in equation 2.66 is very similar to minimizing the mean squared error in order to find the optimum. In fact, the two will have the same optimum location. So, MLE is in fact a proper estimation method in machine learning. Furthermore, what makes MLE competitively better than most alternatives is that it ensure consistency in estimators and as the training set increases in examples, $m \to \infty$, MLE is the best estimator in terms of the rate in which it converges to the true parameter value. It is therefore a very efficient method of estimation, and it explains why it is the most commonly used estimator in the context of machine learning.

2.7 Regularization

In this section regularization methods used in the project will be reviewed as they are crucial to the structure of the ML model built and address several issues that the dataset causes. Firstly a motivation for regularization in an ML model will be given. Afterwards the *ridge regularization* will be presented and mathematically described. At last a technique, *dropout*, will be shown, which has shown to significantly improve model results.

Achieving a model in ML that can generalize and perform well on unknown data is challenging. The most prominent technique for obtaining a model that generalizes without over- or underfitting is regularization. Different techniques have been developed to solve this issue by generalizing the parameters in the training data. The

three methods used most widely in NN's to generalize the results are the lasso regression, ridge regression and the dropout method. These all cause a higher training error and bias, however, reduces the testing error and variance causing the model to perform better on unknown data. In this study, ridge and dropout is used as a regularization tool and therefore only these methods will be presented in this section.

Instead of giving the model a set of specific functions in a space that it can freely use or not, we can create preferences for some over others, however, thereby not completely excluding specific ones which makes the model prefer using some over others. In that case, it will only choose the preferred once unless the alternative is significantly better.

2.7.1 Ridge regularization

The first regularization tool is the ridge regression also called the L2 norm. Here a penalty term, λ, is introduced together with the loss function in order to create a bias that generalises the results of an NN. The penalty term controls for how large the weights are allowed to be and by reducing the weight of the training model it counteracts the overfitting problem. The penalty term in the ridge regularization can approach zero but will not be zero as it squares the coefficient (Essam, 2022).

$$J(\mathbf{w}) = MSE_{train} + \lambda \mathbf{w}^T \mathbf{w} \tag{2.68}$$

Where λ is a value that is applied as a punish value to inform the model how much is cared about the magnitude of weights. Looking at the equation: if we set λ close to 0, then the loss function will not get punished by larger weights, as they are multiplied by 0, therefore one is left with simply the loss function.

Inspecting the path for coefficient weights as λ increases, it is clear that increasing the penalty causes more coefficient weights to move towards zero, thereby punishing the model for imposing complexity. This ultimately boils to down to a specific trade-off: create a good fit for the training data or lower the weights and reduce the generalisation gap.

2.7.2 Dropout

Dropout is a computationally cheap method that regularizes the weights of an NN. Dropout restricts the nodes in the network so that some randomly selected nodes wont get activated in each batch and thereby makes a selective amnesia of the nodes. The benefit of this technique is that the dominant nodes are sometimes removed to find more significance in the non-dominant nodes. Thereby the dominant nodes do not overshadow the non- or less dominant nodes. The model will be able to generalize to a higher degree by being more resilient when dominant nodes are missing. When the random nodes are "turned off" the model is restricted from being too complex, and it is thereby forced to simplify its functional form with a dropout. It forces it to establish relationships with nodes it might not have otherwise considered, which might be good if those nodes become more important for unseen data which often happens. Dropout is notable more effect at a lower signal to noise ratio which mean that combining it with ridge regularization improves the regularizer immensely (Wei, 2022).

2.8 Numerical Optimization

As Section 2.1.1 highlighted, gradients are useful in find the optimal set of parameters. This section use exactly that as a stepping stone towards more efficient optimisation strategies. These strategies are not only crucial for methods such as linear regression, but also the more complex methods introduced later in the theoretical chapter. The method of *gradient descent* (GD), is the first of the algorithms, and therefore also the most simplistic, as a logical next step the algorithm *stochastic gradient descent* (SGD) shows further improvements to GD. The last part of this section will introduce algorithms with adaptive learning rates and show their superiority over the more simple algorithms GD and SGD.

2.8.1 Gradient Descent

GD is a strategy to find the minimum of a function by iteratively moving towards the steepest descent direction. This concept relies on the calculation of the gradient, $\nabla_x f(x)$, which, while indicating the direction of the steepest ascent, also suggests the opposite direction for the descent.

The challenge, however, lies in distinguishing between different types of stationary points: local minima, local maxima, and saddle points. Local minima are points where no small movement can decrease the function value, local maxima are the opposite, and saddle points are flat regions where the direction of ascent or descent changes. Figure 2.4 captures these concepts visually.

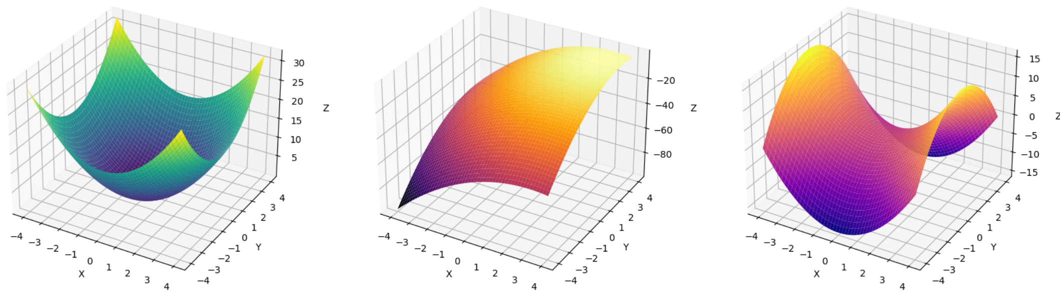

FIGURE 2.4: Three types of stationary points: global minimum (*left*) and maximum (*middle*), and saddle point (*right*). Own illustration

Understanding these concepts is essential, particularly in the high-dimensional spaces typical of deep learning, where functions often have complex landscapes with many local minima and saddle points.

Computing the gradient will not be sufficient for finding local or global minima - how is it possible for the gradient to know in which direction to *step*, in order to minimise the function? Gradient descent uses the notion of a *directional derivative* in direction **u**, which is a *unit vector*. Minimising some function f then involves finding the direction in which the function decreases the fastest. Using the directional derivative, we get the following:

$$\min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=\mathbf{1}} \mathbf{u}^T \nabla_x f(x) \tag{2.69}$$

$$= \min_{\mathbf{u},\mathbf{u}^T\mathbf{u}=\mathbf{1}} || \mathbf{u} ||_2 || \nabla_x f(x) ||_2 \cos\theta \tag{2.70}$$

θ represents the angle between the unit vector, $\mathbf{u}$ and the gradient. Since we are dealing with a unit vector, we can substitute in $|| \mathbf{u} ||_2 = 1$ and removing terms that do not depend upon $\mathbf{u}$. The gradient is positive and therefore points directly upwards, so in order to minimise the function the *negative gradient* is needed. This method for pointing the gradient in the correct direction is known as *gradient descent* (GD). The steepest descent then postulates:

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_x f(\mathbf{x}) \tag{2.71}$$

Here, ϵ represents the learning rate, a scalar that determines the step size. The gradient $\nabla_x f(\mathbf{x})$ points in the direction of steepest ascent; thus, the negative sign ensures movement towards a local minimum.

Gradient descent's efficiency and success depend heavily on choosing an appropriate learning rate and understanding the landscape of the function to be minimised, as poor choices can lead to convergence at non-optimal points or failure to converge at all. New algorithms is therefore needed to solve the issues of GD to achieve more optimal solutions.

2.8.2 Stochastic Gradient Descent

Even though GD is a powerful algorithm for numerical optimisation, it has some apparent flaws; computing the gradient is done for the entire training set, making its computational burden quite large. As datasets increase in size, it becomes much more apparent. The next algorithm for numerical optimisation that needs closer inspection is the *stochastic gradient descent* (SGD). It is very fair to say that this algorithm has been the catalyst for the increase both in popularity and in the applicability of AI and ML. A problem within statistics and all data-driven tasks is that processing and handling large amounts of data has a large computational toll. This also comes with the fact that some models require more memory and more extensive computations. SGD solves this in a very simple and elegant way, making it very efficient, and has transformed the AI and ML landscape since its introduction.

The cost function in machine learning algorithms is typically the sum over training examples of a per-example loss function. The negative conditional log-likelihood of the training data can be expressed as:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},y\sim\hat{p}_{data}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{2.72}$$

where L is the per-example training loss, $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y|\boldsymbol{x}, \boldsymbol{\theta})$.

The computational expense of calculating the gradient grows linearly with the size of the training set, as:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{2.73}$$

SGD addresses this by estimating the gradient using a *minibatch* of examples, $\mathbb{B} = \boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m')}$, drawn uniformly from the training set. The gradient estimate is computed as:

$$g = \frac{1}{m}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{2.74}$$

SGD then progresses by updating $\boldsymbol{\theta}$ according to the estimated gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon g \tag{2.75}$$

Here, ϵ represents the learning rate. Since the learning rate is determined by the modeller, we refer to this as a hyperparameter we can tune. Usually the learning rate is selected through *empirical testing*; in practice, this is done by defining a dictionary in Python with values such as $\{10^{-1}, 10^{-}2, 10^{-}3\}$, and adjustments are made based on the training outcomes. If the learning rate is chosen at a large value, then the model might fail to converge; conversely, a too low value will cause the model to converge slowly, and thereby it might not be feasible to find a minimum of the cost function. Even though SGD has been efficient and popular for generating deep learning models, it has some apparent flaws: the learning rate needs to be chosen and set across the entire training period, which means it needs to be appropriate along the entire curve of the functions. When one is working with convex functions, this does not pose an inherent issue, but in reality, with the data generated from this reality, the functions are often non-convex, making it difficult to choose a learning rate that is appropriate for the entire length of the functions.

Looking closer at Figure 2.5 also provides additional explanation for SGD's limitation; non-convex function which are typical of real world data, can be difficult to minimise since they have many local minima and especially saddle points can pose difficulties in convergence. The problem with saddle points are, that they are surrounded by a plateau with the same error, this makes it difficult for SGD to navigate since the gradient in all directions would be 0.

Figure 2.5 also reveals another interesting fact, besides the trouble of navigating multiple local minima, the choice of learning rate, number of steps and starting point for the optimisation algorithm have tremendous impact on finding the minima of the function. In the plot, the starting coordinates are selected to be $[-2, -3]$ with a learning rate of $\epsilon = 10^{-2}$ and 20 steps. As shown in the plot the minimum is not achieved entirely, increasing the learning rate or the number of steps would for this case increase the possibility of achieving the minima, but increasing the complexity of the data, i.e. high-dimensional data, and computing the gradient on that data, it is not difficult to imagine that achieving the minima is a complex task. In addition to this the figure also illustrates that computing the gradient implies that the optimisation makes large leaps at the beginning and slows down towards the optimum. This is in part due to the large values for the starting position of this case, but also due to the fact that the gradient points in the direction of the most rapid changes to the

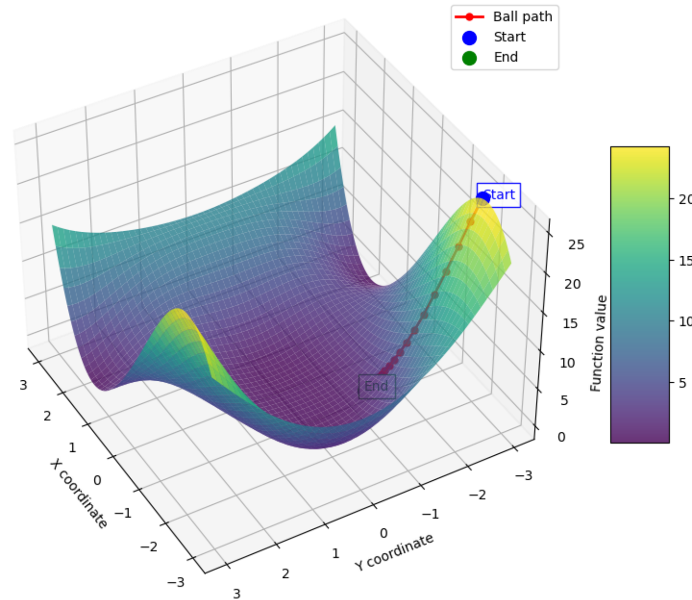

FIGURE 2.5: Custom function with multiple local minima (shifted 150 degrees for a better viewing angle). *Custom function*: $(x^2 + y^2) * \sin(x) * \cos(y) + (x^2 + y^2)$. *Parameters:* $\epsilon = 10^{-2}$, no. steps = 20, starting coordinates = $[-2, -3]$. Own illustration

function. The function graph has a steep slope at at the beginning of optimisation which changes towards the minimum therefore decreasing the size of the step.

With these known drawbacks, it is favorable to introduce the concept of *momentum*. SGD is well-known to get stuck in *ravines*; a ravine is a region where the surface of the function descends steeply in one direction but is relatively flat in another. In these specific areas, SGD has a tendency to oscillate across the narrow ravine, rather than moving towards the local or, in some cases, global minima, therefore leading to slow or no convergence at all. To tackle this issue, momentum was introduced as a method, as it helps SGD accelerate in the relevant direction. Very simply, the method applied adds a fraction, γ, of the update vector from the previous update step to the current one. One can imagine a ball rolling down a hill, accumulating momentum, and becoming faster and faster. Introducing a new variable for momentum, $\mathbf{v}_t$ and then using the previous momentum and the current mini-batch gradient $\mathbf{g}$ the following update rule is applied:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \epsilon \mathbf{g} \tag{2.76}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \mathbf{v}_t \tag{2.77}$$

Keeping with the analogy of a ball rolling down a hill, when computing the gradient, it is most importantly the negative gradient that is computed. Without any friction, in the case of $\gamma = 1$, the ball would accelerate continuously; if there is resistance, or air resistance, when $\gamma < 1$, the ball would initially accelerate but at some point reach a terminal velocity where the momentum is balanced by the air resistance. By simply adding a constant fraction, the update vector has reduced oscillation by smoothing

out changes when the gradients change direction. It is also a term that is dependent upon past gradients, which yields the effect of accelerating in the direction of gradients that are consistent with each other.

Even with the addition of momentum, the previously mentioned algorithms still have a significant caveat, namely the constant learning rate. A closer inspection of another algorithm, *Adaptive Moment Estimation*, usually referred to as *ADAM*, will show improvements made upon the SGD algorithm with specifically the learning rate in mind.

2.8.3 Optimisation Algorithms with Adaptive Learning Rates

With the use of momentum, the oscillation that the SGD suffers from was dampened. Though it comes at the expense of introducing yet another hyperparameter, within ML, this matters quite significantly as it increases the computation. Instead, algorithms with adaptive learning rates that use a separate learning rate for each parameter and adapt the learning rates for each parameter across the entire training period can be favorable. This section will provide a high-level introduction to two algorithms, *AdaGrad* and *RMSProp*, before providing a walk-through of the algorithm used for learning in the eventual model.

The AdaGrad method was first presented by (Duchi, Hazan, and Singer, 2011). In this algorithm the learning rate is reduced significantly for parameters with large gradients and very little for those with lower gradients. This method efficiently allows for larger updates in the directions with less steep gradients, so encouraging effective development in those areas of the parameter set. Later, in 2012, the RMSProp algorithm was introduced by (Krizhevsky, Sutskever, and Hinton, 2012), with modifications to AdaGrad increasing performance for non-convex functions by changing the gradient accumulation into an exponentially weighted moving average. Despite the high performance and success of these algorithms, this thesis instead turns to another algorithm that adopts the concept of an adaptive learning rate, the Adam optimizer.

Adam is a combination of RMSProp and momentum, with a few important tweaks. As RMSProp, Adam stores an exponentially decaying average of past squared gradients, $\mathbf{r}_t$, but it also includes an exponentially decaying average of past gradients, $\mathbf{s}_t$ which mimics the concept of momentum from earlier. Defining these two estimates:

$$\mathbf{s}_t = \rho_1 \mathbf{s}_{t-1} + (1 - \rho_1)\mathbf{g} \tag{2.78}$$

$$\mathbf{r}_t = \rho_2 \mathbf{r}_{t-1} + (1 - \rho_2)\mathbf{g} \odot \mathbf{g} \tag{2.79}$$

The exponential weighted moving average is well-known from its applications in finance, but are also included into this algorithm. We can think of the two Equations; 2.78 and 2.79 as being the estimates for the first moment, the mean, and the second moment, the uncentered variance of the gradients respectively, and this is exactly where the Adam algorithm got its name from. The first and second moments are concepts from statistics that describe the characteristics of the operation performed, hence the first moment (the mean), describes the momentum of the gradient in Equation 2.78. The second moment (the uncentered variance), captures the variability or spread of the gradient magnitudes. Hence these two equations provides a clear explanation of the *gradient distribution*. The two hyper-parameters $\rho_1, \rho_2 \in [0, 1]$,

control the exponential decay rates of the moving averages. In Equation 2.79 the operation $\odot$ is introduced, which encapsulates the *Hadamard product*, which is an element-wise product operation, it takes in two vectors (or matrices), and outputs a vector of the same dimensions, with each element corresponding to the product of the corresponding elements of two vectors. The rationale for using the Hadamard product over simply squaring the gradient vector, is to differentiate between different operations, making the operations done unambiguous. Additionally it generalizes well to high-dimensional data, such as tensors.

The authors behind the Adam optimiser discovered in their paper (P. Kingma and Li Ba, 2015) that since the moving averages are initialised as vectors of 0's, it leads to estimates that are biassed towards 0. To level this issue out, they computed bias-corrected first and second moment estimates:

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\rho_1^t} \tag{2.80}$$

$$\hat{\mathbf{r}}_t = \frac{\mathbf{r}_t}{1-\rho_2^t} \tag{2.81}$$

This leads the Adam optimizer to the following update rule for the parameter set. The important part is that it is a parameter set, so each parameter is being adjusted individually but combined in a vector:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta} \tag{2.82}$$

The update rule presented in Equation 2.82 presents the idea of scaling or normalising the gradient, which assists in scaling the step size appropriately for each of the parameters in the parameter set. δ is simply a constant used in this equation for numerical stability to ensure that the term avoids division by 0. Usually, this constant is chosen at a low value, such as 10^{-8}. Interestingly, the square root does not have too much meaning behind it; other than removing it, it worsens performance, according to (Ruder, 2016).

Having set up the rationale and mathematics behind the Adam optimizer, the algorithm in its entirety is presented in Algorithm 1. In the overview, a walk-through is provided as well as necessities, such as setting parameter values before beginning the iterative process. The algorithm also shows the suggested default values from the literature, provided by the authors of the optimisation strategy in the original paper.

This section has shown that choosing a specific optimisation strategy can matter quite significantly for the outcome of the training process. Not only the algorithm itself but also the parameters that the strategy uses can have a tremendous impact. The choice between SGD, AdaGrad, RMSProp, and Adam can at times be a heuristic choice but also a personal choice from the modeller. Often times, it does not clash in that a specific algorithm has superiority over another, but rather with the modeller's familiarity with it. With that being said, this section has shown that having an adaptive learning rate can be a compelling choice, as a constant learning rate has the risk of being stuck in local minima, thus decreasing the performance of the model, which ultimately is what one cares about in ML.

Algorithm 1 Adam Optimization Algorithm

(Goodfellow, Bengio, and Courville, 2016).

Require: Step size ϵ (Suggested default: 10^{-2})
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$ (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$

1: Initialize 1st moment vector $\mathbf{s} = 0$
2: Initialize 2nd moment vector $\mathbf{r} = 0$
3: Initialize time step $t = 0$
4: **while** stopping criterion not met **do**
5: Sample a mini-batch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$
6: Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(x^{(i)}; \boldsymbol{\theta}), y^{(i)})$
7: $t \leftarrow t + 1$
8: Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1)\mathbf{g}$
9: Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2)\mathbf{g} \odot \mathbf{g}$
10: Correct bias in first moment (mean): $\hat{\mathbf{s}} \leftarrow \frac{s}{1 - \rho_1^t}$
11: Correct bias in second moment (variance): $\hat{\mathbf{r}} \leftarrow \frac{r}{1 - \rho_2^t}$
12: Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ (operations applied element-wise)
13: Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
14: **end while**
15: **return** Updated parameters (according to update rule) $\boldsymbol{\theta}$

2.9 Feedforward Neural Network

Feedforward neural networks (FNN) form the baseline for the world of NN's. While it is not necessarily used a lot in practice due to some critical limitations, it is a quintessential building block in practically all other NN's, since most are expansions of it. The following will build up the architecture of an FNN by starting at the most basic level: introducing a perceptron. Then, combining multiple perceptrons, a feedforward neural network is defined. It will finally be shown why the model itself is not practically as efficient as desired. The limitations of the model motivate for the introduction of more complex models.

2.9.1 The Perceptron

An FNN is an ML architecture consisting of what is known as *perceptrons* (also known as *neurons*). In fact, all kinds of NN's are built on multiple of these perceptrons, and so, a good starting point in understanding how FNN's behave is to introduce the mechanism of a perceptron - both mathematically and illustratively.

In essence, a perceptron is quite simple and is best explained by going through the forward propagation of information through the single perceptron. The overall goal of a perceptron is to learn a function, $f(\mathbf{x}; \boldsymbol{\theta})$, that maps a connection between a vector of inputs, $\mathbf{x}$, with some output, $\mathbf{y}$. Obviously, through learning, the estimated function should represent something very similar to the true mapping between the given input and output data. In other words, the algorithm learns the optimal values of the parameter set, $\boldsymbol{\theta}$. The process of how this is done can be seen in figure 2.6. As seen in the figure, a neuron can take on multiple inputs, $\{x_1, x_2, ..., x_m\}$, each

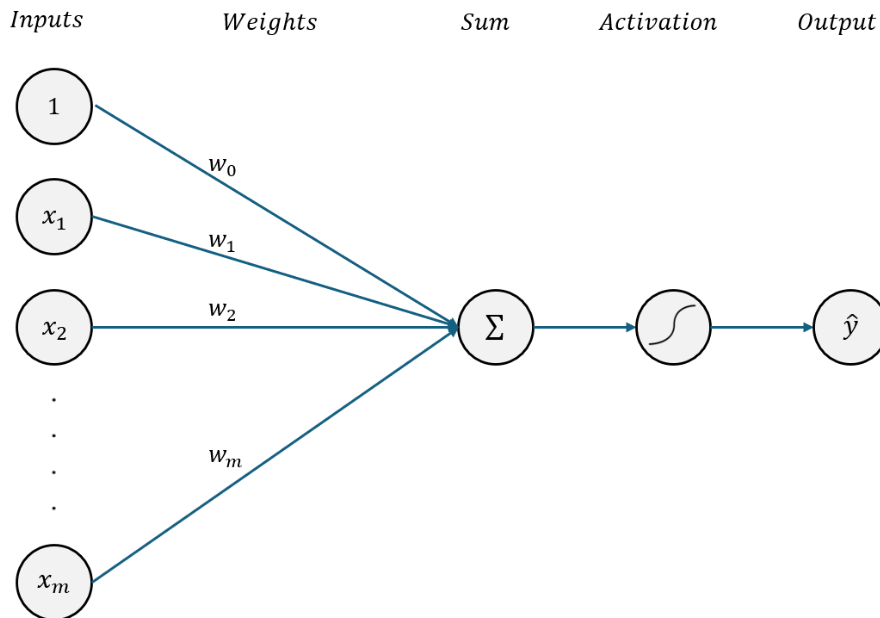

FIGURE 2.6: A perceptron (or neuron) - forward propagation process.
Own illustrations

of which can be argued to have some connection to the output variable of interest. These inputs are given unique weights, $\{w_1, w_2, ..., w_m\}$, that are multiplied with the respective inputs. This has in some ways similarities to basic coefficients of standard regressions; e.g. a weight of ≈ 0 signals an input with low to no importance, a weight of ≈ 1 signals an input with high importance and so on. Note also the input at the top of the illustration; this is the bias term, which is shown to be very useful in neural networks, as it provides the ability to shift the function to fit the output better. The bias input is set to 1 so that any weight multiplied with it will simply be the weight, w_0.

Next, all weighted inputs are combined into a sum and at this point, it is just a linear combination. Mathematically, this can be written as:

$$w_0 + \sum_{i=1}^{m} x_i w_i \tag{2.83}$$

This sum produces a single value that is then brought into an *activation function*. This is one of the defining steps of neural networks as it converts the function from a linear combination into one that is non-linear. The activation function thereby allows inputs to have a non-linear connection to the output, which is very desirable - after all, the world seems to be quite non-linear. The activation function can be defined by multiple different non-linear functions according to what type of output is desired. Some of the more frequently used activation functions are *sigmoid*, *softmax* or *rectified linear units* (ReLU) functions. As an example, the sigmoid function "squishes" any value brought into it and converts it into a number between 0 and 1, which is very practical if the desired output format is some probability. With the inclusion of the activation function, the mathematical expression defining the entire forward propagation of a perceptron can be written as:

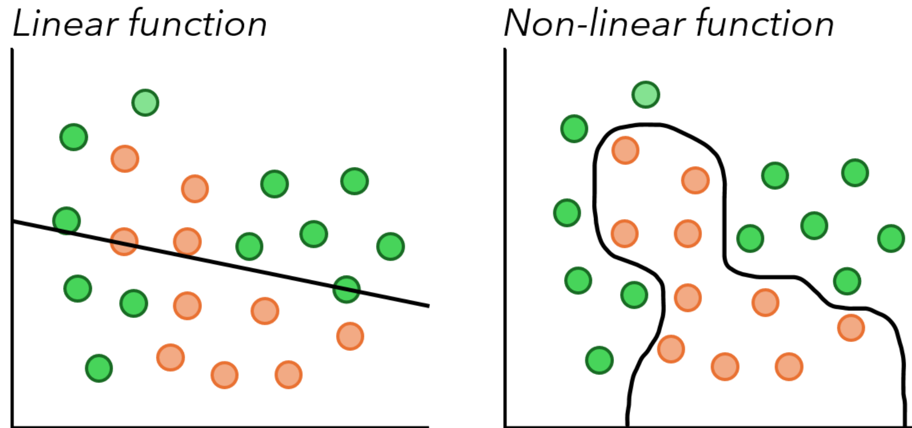

FIGURE 2.7: Linear vs non-linear function, own illustrations

$$\hat{y} = g(w_0 + \sum_{i=1}^{m} x_i w_i), \tag{2.84}$$

where the linear combination of the inputs are inside the brackets, and this is all passed through the non-linear activation function, g. To bring this equation closer to the language of ML, it can easily be rewritten in terms of dot-products and vectors:

$$\hat{y} = g(w_0 + \mathbf{x}^T \mathbf{w}), \tag{2.85}$$

where $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$ and $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$. To clarify further how the non-linearity can become necessary in mapping real-world data, see figure 2.7. In this simple example, the task is to separate green dots from orange dots. A linear function (left) has far from enough capacity to capture this complex task, however, increasing the capacity of the model to include non-linearity (right) seems to be able to solve this more complex separation. This represents generally the motivation for increasing model capacity in most real-world data sets.

Going forward, to simplify notations, let $h = (w_0 + \mathbf{x}^T \mathbf{w})$. This also means the process of a perceptron is noted as: $\hat{y} = g(h)$, which makes it more simple to extend a single perceptron into an NN.

2.9.2 From A Perceptron To An FNN

Now that a single perceptron has been introduced, an expansion into multiple perceptrons leads to what is called an NN (a network of neurons). Furthermore, it is also possible to expand into multiple different output units. A group of units is usually refereed to as a *layer* and deciding how many layers to include in the network is part of planning the *architecture* of an NN. Additionally, the amount of units in each layer is referred to as the *width*, which may very well vary from layer to layer.

Lets assume first a simpler case illustrated in figure 2.8.

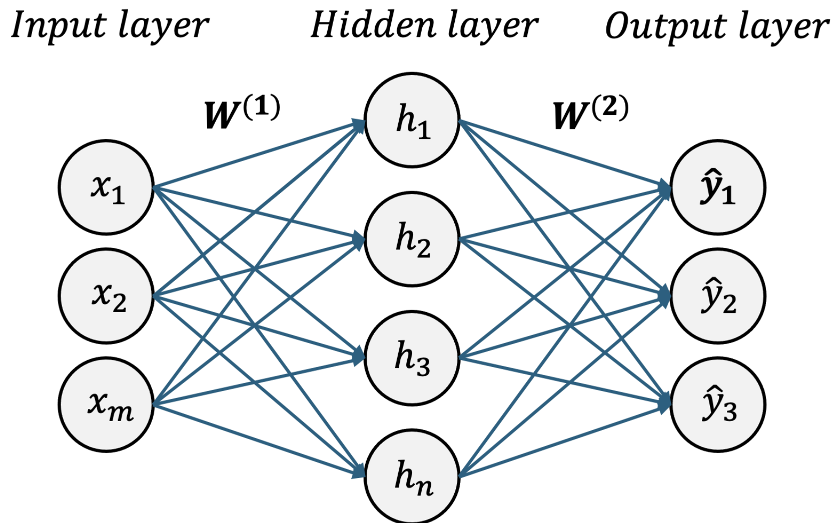

FIGURE 2.8: A neural network with one hidden layer. Own illustrations

Notice now, that a *hidden layer* has been introduced. The name *hidden* comes from the fact that what goes on inside this layer is not directly observed or controlled by the human as the training data only instructs what input and output values must be. It is therefore up to the training algorithm itself to utilize it optimally in order to end up with a reasonable output value that matches y. The hidden layer consists of multiple perceptrons that are all linked to each input and the process of going from the input layer to each perceptron in the hidden layer is exactly the process just described in relation to a single perceptron. But now, since there are more than one perceptron, the weight vector now becomes a matrix of weights. So the equation for the linear combination of inputs, $\{h = (w_0 + \mathbf{x}^T\mathbf{w})\}$, changes slightly to: $\{h = (w_0 + \mathbf{x}^T\mathbf{W})\}$.

In figure 2.8 the bias term cannot be seen for simplicity, but importantly; it is assumed to be present in all layers, which is also seen mathematically. Note also, in the figure, that since the hidden layer has now been introduced, the forward propagation now involves both a transition from input layer to hidden layer and a transition from hidden layer to output layer. This means, that two weight matrices are now needed; $\boldsymbol{W}^{(1)}$ and $\boldsymbol{W}^{(2)}$. Each transition in a network is defined by being a perceptron, which means it requires to have 1) a weight matrix, 2) a bias term and 3) an activation function.

Mathematically, going from the input layer to the hidden layer is given by:

$$h_i = (w_{0,i}^{(1)} + \mathbf{x}^T\boldsymbol{W}^{(1)}), \tag{2.86}$$

which is just the linear combination introduced in the former section. Inside the hidden layer, the linearity is passed through the activation function, g. At this point, the process is similar to the example of a single perceptron, but as just mentioned, a second transition with a new weight matrix, bias and activation function is needed to reach the output. The mathematical intuition is very similar here instead now, the input variable is the output of the first step: $g(w_{0,i}^{(1)} + \mathbf{x}^T\boldsymbol{W}^{(2)}) \rightarrow g(h_i)$. Each of these new inputs are element-wise multiplied by the new weight matrix, $\boldsymbol{W}^{(2)}$, and a new

bias is added. Lastly, this value is again passed through an activation function to reach the output:

$$\hat{y}_i = g(w_{0,i}^{(2)} + \boldsymbol{g}(\boldsymbol{h_i})^T \boldsymbol{W^{(2)}}). \tag{2.87}$$

It becomes obvious that expanding these neural networks into multiple layers quickly enlarges the amount of parameters and activation functions, requiring massive computational efforts. Adding more layers means increasing capacity of the model, specifically because each layer has an activation function (which may change functional form through each layer as well). Remember again, the amount of layers is called the depth, which is where the name *deep* neural networks arose from. Adding more layers allows the model to fit extremely complex mappings, but one has to be careful: at some point the model capacity becomes excessive, it performs excellent on the training data, however, it may then generalize poorly on unseen data, hence it is easy to produce a model which overfits. The exact depth of the model and the width of each layer is a hard task to decide on and there is no universal answer. The method of deciding these amounts is through experimentation; repeatedly training the model on new architectures and testing on the validation set is they way to go, though it might require a lot of time.

2.9.3 Back-propagation

Up until this point, the weights and biases have simply been assumed to be given - nothing has been mentioned in regards to how they are actually obtained. At the moment, the network is taking a vector of inputs, $\mathbf{x}$, and producing a value or vector of outputs, $\hat{y}$, hence the information is flowing forwards. This process is called forward-propagation, and it continues until a loss function is produced - but importantly, weights and biases are randomly chosen at this point. The next, and arguably the most important, step in the learning algorithm is to then use this loss to flow *backwards* through the weights and biases in order to produce a gradient - *the gradient optimization was described in Section 2.8*. This process is then called *back-propagation*, and luckily this algorithm can be used for practically any function, and hence in all kinds of deep neural networks. The algorithm itself is quite simple as it is just taking the derivatives of the loss function w.r.t the parameters, however, for neural networks, it can quickly become a computationally expensive process as the amount of layers and examples increase.

Now, since the back-propagation algorithm computes the gradient by flowing backwards through an entire network consisting of multiple layers, it needs to take derivatives of functions chained to each other (e.g three functions composed together creating a network: $\{f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(x)))\}$). This also means, that the back-propagation algorithm is required to use the chain rule quite a lot. The chain rule can be described and generalized quite easily. Suppose a function $y = g(x)$ which is the inner function of another: $z = (f(g(x)) = f(y)$. Then using the chain rule, one can find the derivative of z with respect to x:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.88}$$

Since the two functions, $y = g(x)$ and $z = f(y)$, are dependent of each other, then z also depends on x through the intermediate function, y, and this connection can

be expanded to many more layers, which is often apparent in neural networks, and so computing the gradient becomes a product of multiple derivatives. Equation 2.88 is simplified to a single scalar input for x and y, but can be generalized to multiple inputs and functions:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \tag{2.89}$$

which can then also be written in matrix form:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z. \tag{2.90}$$

An important note regarding the back-propagation algorithm is that it does nothing more than using the chain-rule to compute the gradient. Deciding what to do with this gradient is not part of back-propagating itself - this is where the stochastic gradient decent algorithm takes over and learns in which direction to take the weights and biases towards. When this is done with some given learning rate, the back-propagation is once again used to compute a new gradient. This is mentioned, because the back-propagation algorithm is often misunderstood in also doing the SGD for example, which is not the case. To clarify, the process of the learning algorithm can generally be described as:

1. *Forward-propagation:* from input through hidden layers to output
2. *Back-propagation:* computing gradient of loss function w.r.t weights and biases
3. *SGD:* update the weights and biases according to the learning rate
4. Repeat back-propagation and SGD until desired output is achieved

So, to sum up, back-propagation is quite simple intuitively, yet arguable the central most important part of training a neural network. It is also used in most expansions of the above feedforward version of a neural network. The issue now, is that while the feedforward network seems very logical and can indeed be useful in certain scenarios, it will not succeed in helping with the issue of forecasting exchange rates. This is due to the fact, that it does not handle sequential data at all. That is, if working with a time series, the feedforward model will never accomplish anything useful. Luckily, it requires only a few additional features to expand the current feedforward version into one that handles sequential data. The majority of the above descriptions share the same methods, which is why expanding is not such a huge step. In what comes later, it is shown that the *recurrent neural network* (RNN) model family (specifically the *long-short term memory* (LSTM) model) will provide massive improvements to modelling and training on a time series data set, which is required in this case. It fills the last gap required to finally start the practical training process.

2.9.4 Activation functions

Before diving into the more advanced expansions of the model, it is essential to first introduce and go through another central decision-making in designing the NN architecture, which is choosing the specific activation functions to use. This step is quite important as it affects both the type of output provided as well as how well the model trains on some specific data set.

Through the years, a lot of different activation functions have been introduced in the context of NNs (Brownlee, 2021). Activation functions are used both in hidden layers and output layers, but act has different roles in the two - in the hidden layers, the activation function is central to how well the model learns, and in the output layer, it defines the type of prediction the model is desired to provide. Throughout the hidden layers, it is most common to use the same activation function in all layers, but it is not necessary. Lastly, the hidden layer activation is often different from the output layer activation. The below will describe some of the most used and efficient activation functions for the two layers. A description of their usefulness will help guide which activation functions will later be optimal for the given model at hand.

Necessary for all activation functions is that they are differentiable. If an activation function is non-differentiable, then its not possible to do back-propagation through the network as the first-order problem cannot be solved. Below are some of the most commonly used activation functions in practice:

1. ReLU
2. Logistic (Sigmoid)
3. Hyperbolic Tangent (Tanh)
4. Softmax

ReLU

The ReLu function has become among the most used activation function in hidden layers. This is primarily the case since it is computationally very light and models train very efficiently and fast on this activation. The function itself is very simple, and almost looks linear in its nature, however it still implements a non-linearity at values around 0. The mathematical expression is given as:

$$ReLU(x) = max(0, x), \tag{2.91}$$

and so, if the input value, x, is negative, it returns 0 (and the input is not activated), and if x is 0 or positive, the function returns x. The model extracts some desirable features from the linear function, as it returns a linear output for all positive values of x, but always returns 0 for negative values. The latter part is what makes it nonlinear, and the simple characteristics make it very efficient when performing the back-propagation. It can also be helpful to see the function on a graph - for this, see figure 2.9. One issue that often might arise in using the ReLU activation is that it might cause what is called *dead* neurons. This arises when the function ends up returning 0 values too often and hence a lot of important neurons are deactivated (or *killed*). A common fix to this is to modify it into a *leaky* ReLU, where instead of all negatives values returning 0, they return small negative numbers, and hence keep the activation going. This requires only a small additional feature, but will not be given here.

Sigmoid (Logistic) Activation

The sigmoid activation was the primary function used in the early stages of deep learning before the ReLU came into the picture. With that said, it still serves as a great option today, as it can be very useful especially in the output layer. The sigmoid

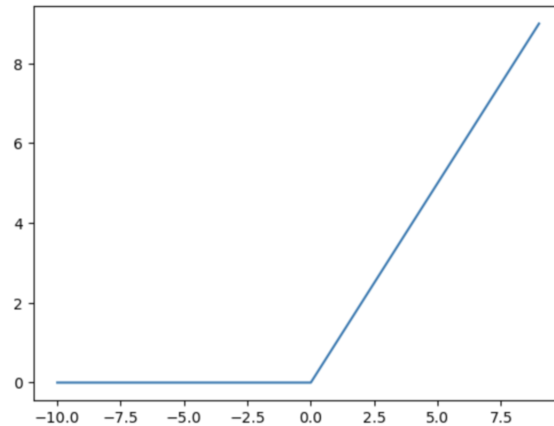

FIGURE 2.9: **Rectified Linear Activation Unit** (ReLU). Own illustration

activation *squashes* all input values into an interval between 0 and 1, and for this reason it is arguable the best activation if the output prediction is some probability (since all probabilities lie between 0 and 1). The nature of this function is quite obviously nonlinear, and for input values close to 0, it has a very steep return curve, whereas for most input values in either positive or negative directions, it returns values close to 0 or 1. So, the sigmoid activation is very useful to use in the output layer if the desired prediction is some binary classification or in general probability. In the case of binary classification, an example could be that all output values below 0.5 are classified as *false*, and all output values above 0.5 are classified as *true*.

The mathematical expression for the logistic sigmoid activation is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.92}$$

and an illustration of this is shown in figure 2.10. However, this sigmoid function also comes with some heavy downsides, which become apparent when doing the back-propagation. Even though the sigmoid function is differentiable, the derivative of such a function often provides very small numbers. This can cause the earlier mentioned issue of a vanishing gradient - as the gradient approaches values close to 0, the training will become weak and stuck at astronomically small steps. The issue can in fact be illustrated well by taking the derivative of the sigmoid function (which is exactly what is done during back-propagation), and can be seen in figure 2.11. As it is shown, the derivative is only significant with input values close to 0, but very quickly approaches 0 as the input values become bigger. This issue is mainly what causes the literature to shift away from using the sigmoid function (Baheti, 2021).

Hyperbolic Tangent (tanh)

The tanh function is very similar to the sigmoid, and in fact, it is also a sigmoidal function in that it follows an *s-shape* curvature. It differs, however, from the sigmoid logistic function by providing an output interval of -1 to 1 (instead of 0 to 1). This difference alone actually makes the tanh a great candidate for *hidden* layers. This is due to the fact that the activation will always center the data around 0 (the expected value of the ouput is 0). One advantage of this is that it easily maps the output values

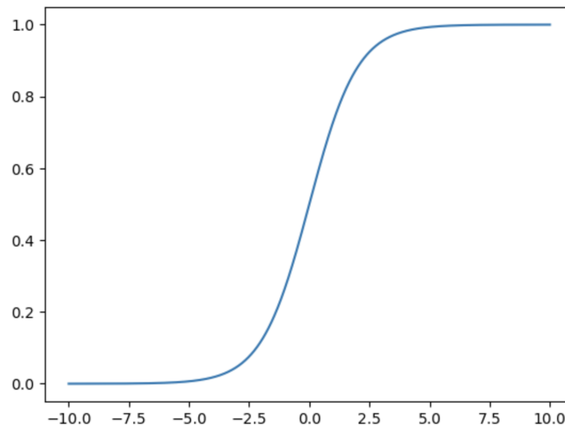

FIGURE 2.10: Sigmoid Activation Function (logistic). Own illustration

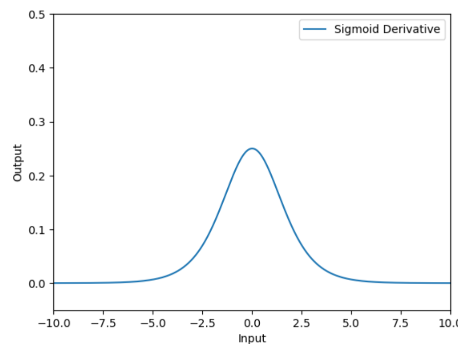

FIGURE 2.11: Derivative of sigmoid function. Own illustration

as either strongly positive, neutral or strongly negative. But much more importantly; by centering the data around and close to 0, it *standardizes* the values which makes learning in the next layer much easier, and that explains why it is great in hidden layers.

Besides that, it is very similar to the logistic sigmoid function, and unfortunately also suffers from the vanishing gradient risk, as gradients will only be significant for input values very close to 0. The mathematical expression for tanh is:

$$tanh(x) = \frac{e^x - e^{(-x)}}{e^x + e^{(-x)}}, \tag{2.93}$$

and an illustration of this is presented in figure 2.12, which clearly shows the similarities and differences from the sigmoid logistic function.

Softmax Activation

The softmax activation function is one of a kind. It does not follow the same patterns as the other functions but is proven to be great at what it does. First of all, the softmax function is build upon the logic of the logistic sigmoid function - hence it outputs some sort of probability. But suppose now, that instead of producing a binary classification, one wishes to compute the probabilities of multiple output classes - this is what the softmax activation helps provide. It is the optimal choice for

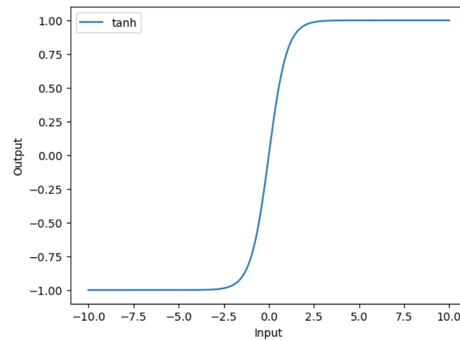

FIGURE 2.12: The tanh activation function. Own illustration

multi-classification outputs (Baheti, 2021). This also implies, that it does not follow the idea of outputting a single number, but instead a vector of numbers, which makes it impossible to illustrate on a graph. With that said, it is still possible to provide a mathematical expression:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}. \tag{2.94}$$

The expression itself already hints towards of the intuition; the function is dealing with some *relative* probability over a sum. This is best explained through a simple example. Suppose one is interested in determining the probability of three different possible outcomes. This requires the output layer to have three neurons. Let the neurons provide the following vector of outputs: [1.8, 0.9, 0.68]. Letting these values pass through the softmax activation function yields relative probabilities of each value: [0.58, 0.23, 0.19]. Notice here, that the sum of this vector is 1, and each of the values in the vector shows the relative probability of observing output i. Lastly, the softmax function returns 1 for the largest probability index and 0 for the remaining. In this case, it returns 1 for the first index and 0 for the two others, indicating which classification the model predicts.

As mentioned, there is not direct graph or curvature to illustrate in regards to softmax functions, however, purely for the sake of consistency, figure 2.13 shows the process of passing data through the softmax function, specifically for the example just gone through.

FIGURE 2.13: Softmax activation transfer process. Example values given in text section. Own illustration

2.10 Recurrent Neural Network

In an FNN all nodes pass in one direction, however, in an RNN, loops are used to encode previous information of the model into the current output. The feature that allows the RNN model to remember previous outputs is called the *hidden state*.

The RNN model divides the data into sequences of mini batches with the size τ. This is done for the model to analyse these mini batches as one whole sequence. So if τ is 30, each mini batch will contain 30 days of data, and so, it remembers every 30 rolling sequences.

2.10.1 Unfolding a Recurrent Neural Network

When taking a deeper look into the hidden state; unfolding the sequencing function gives an intuitive understanding of how the RNN model works.

Consider the following classical form of a sequencing system:

$$\mathbf{h}^t = f(\mathbf{h}^{t-1}, \mathbf{x}^t; \boldsymbol{\theta}), \tag{2.95}$$

where $\mathbf{h}^t$ is the hidden state of the system, where t describes which sequential iteration the system is at. $\mathbf{x}^t$ is the input variable and $\boldsymbol{\theta}$ is the parameter set.

If equation (2.95) has 3 sequences, $\tau = 3$, the system could be unfolded in the following way:

$$\mathbf{h}^3 = f(\mathbf{h}^2, \mathbf{x}^t; \boldsymbol{\theta}) = f(f(\mathbf{h}^1, \mathbf{x}^{t-1}; \boldsymbol{\theta}), \mathbf{x}^t; \boldsymbol{\theta}). \tag{2.96}$$

The first hidden state, $\mathbf{h}^1$, is computed just as a standard feedforward network, $f(\mathbf{x}; \boldsymbol{\theta})$ with no previous hidden state involved. The output of the first hidden state is then included as input into the calculation of the second state giving the function $f(\mathbf{h}^1, \mathbf{x}^{t-1}; \boldsymbol{\theta})$ to calculate the second state $\mathbf{b}^2$. By using the same method as for $\mathbf{h}^2$ the third state, $\mathbf{h}^3$, is calculated by using the second state as an input giving the function $f(f(\mathbf{h}^1, \mathbf{x}^{t-1}; \boldsymbol{\theta}), \mathbf{x}^t; \boldsymbol{\theta})$ which is the same as $f(\mathbf{h}^2, \mathbf{x}^t; \boldsymbol{\theta})$. The system contains all the information about the 2 previous periods and can be adjusted depending on τ.

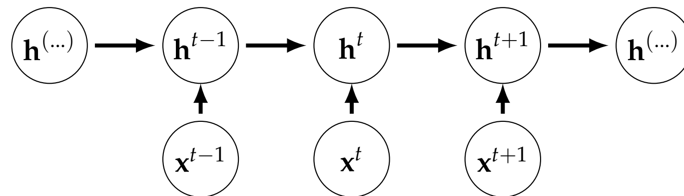

FIGURE 2.14: Sequencing system for the hidden state - own illustration.

In figure 2.14 a sequencing system can be seen. Firstly, the system uses the previous output as well as the input values for $t-1$ noted by $\mathbf{x}^{t-1}$ to calculate $\mathbf{h}^{t-1}$. Afterwards it uses the value for $\mathbf{h}^{t-1}$ and input values time t noted by $\mathbf{x}^t$ to calculate $\mathbf{h}^t$. The system will continue this cycle until there are no more input values.

2.10.2 The mechanics of a Recurrent Neural Network

With the knowledge of how a sequential system works the RNN model can be defined. The design has some significant inclusions from just being a sequential system. It combines the sequential system with a feedforward network to create a model that can optimize the values of the parameter set, $\boldsymbol{\theta}$, and find sequential dependencies of the data generation process. Some different types of RNN models have been developed, however, for the purpose of this study a *many-to-many* RNN model is defined which produces an output at each time step.

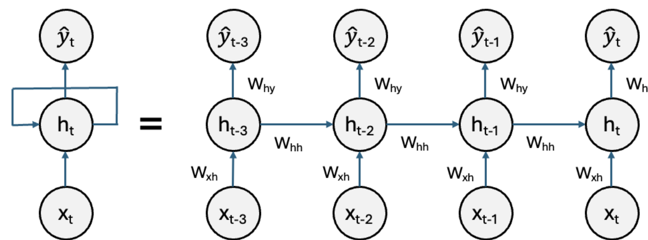

FIGURE 2.15: Basic recurrent neural network with 3 iterations. Own illustration

In figure 2.15 a basic RNN model with 3 iterations is illustrated. Where W_{xh} is the weight between the input and the hidden state, W_{hy} is the weight between the hidden state and the individual time step output and finally W_{hh} is the weight between the hidden states in each time step.

Further explaining the RNN model it can be observed that each input variable, $\mathbf{x}$, leads into a hidden state, $\mathbf{h}$, that uses forward propagation on the inputs to calculate the weights for a single time step resulting in the output, $\mathbf{y}$. However, the main difference from an FNN stems from the hidden state feeding an input to the next time step. It should be noted that the weights are the same for each time step which is central in calculating the loss function of the model. Note that there is a weight between each hidden state which either increases or decreases the sequential dependencies of the hidden state.

Calculating the training loss for such a model is done by taking the individual training loss of each time step and summarizing it into a grouped training loss. This concept is illustrated in figure 2.16.

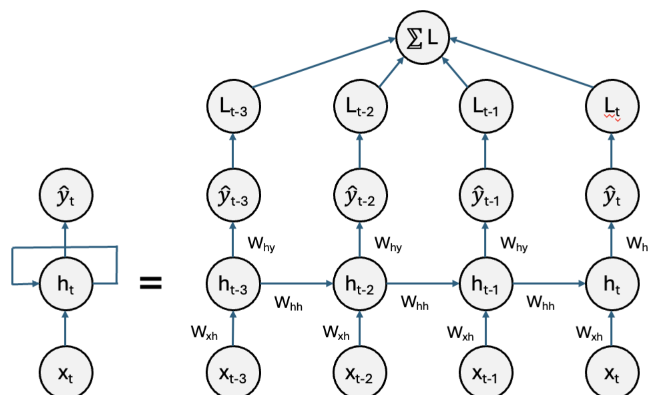

FIGURE 2.16: Basic recurrent neural network with 3 iterations - own illustration

2.10.3 Back-propagation Through Time

When training an RNN model, back-propagation is used to learn the optimal weights of the model, however, with a small difference in order to handle the sequential information. When using back-propagation through time, the algorithm is computing gradients to gradually adjust both the input and output weights as well as the addition of the hidden state weights between time periods to find the optimal parameter set.

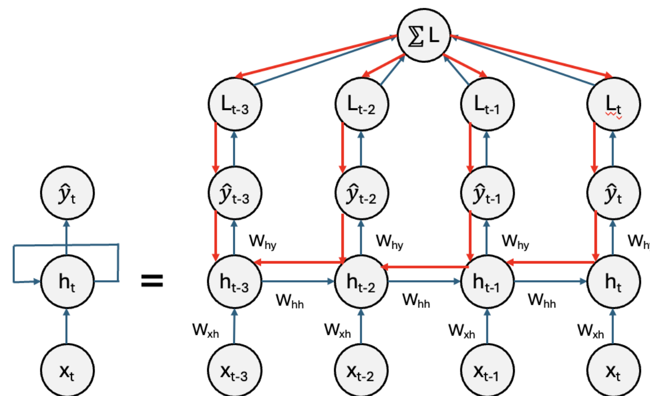

FIGURE 2.17: Basic recurrent neural network with 3 iterations - own illustration

In figure 2.17 the process of back-propagation through time is illustrated. The model show the standard back-propagation for each time step, however, it also shows a back-propagation between time steps which separate back-propagation through time from a standard back-propagation algorithm.

2.10.4 The vanishing and exploding Gradient Problem

The RNN often deals easily with short-term dependencies, however, the central issue with the basic RNN model is that it has issues dealing with long-term dependencies. The core issue stems from what is often referred to as the vanishing and exploding gradient problem.

Both issues depend on the value of the weight between the hidden states, W_{hh}. If the value of the weight is below 1, the long-term dependencies will experience the vanishing gradient problem as the value of a important long-term hidden state will diminish over time since it is multiplied by a value below 1 at every time step, multiple times. The result of this diminishing value causes the gradient to become close to zero. However, if the weight between the hidden states is above 1, a different problem for the exploding gradient arises. In this case long-term effects will be increasing at each time step as it will be multiplied by a constant. This increases computation time as well as increases the value of the gradient which leads to the algorithm for back-propagation through time having issues with gradually taking small steps as the value of the gradient is large.

Because of problem with vanishing or exploding gradient, the basic RNN is not used widely, however, the idea behind the model can be used to make a more advanced versions like the Long short-term memory model.

2.11 The Long Short-term Network Model

In the following section the LSTM model will be presented. The section is mainly built from the theoretical presentation of (colah, 2022), (GeeksforGeeks, 2023) and (Starmer, 2022) who all give a very thorough and clear description of the model. The section starts with the basics dynamics in the model slowly unravels the system behind the LSTM.

Because of the gradient problem a basic RNN model has limited applications in a practical sense, however, if a more advanced RNN model is used, the problem with the vanishing and exploding gradient can be mitigated. One model that accomplishes this and therefore is used broadly in practical applications is the *LSTM* model.

The LSTM model utilises the concept of *gates* to determine the short- and long-term effects of inputs by altering what is called a *cell state*. It is built on the loops from the RNN model, however, uses them in a more advanced method where it constantly evaluates what should be included in the short- and long-term memory in terms of what should be added or forgotten from the short- and long-term memory. The LSTM model is visually represented in figure 2.18.

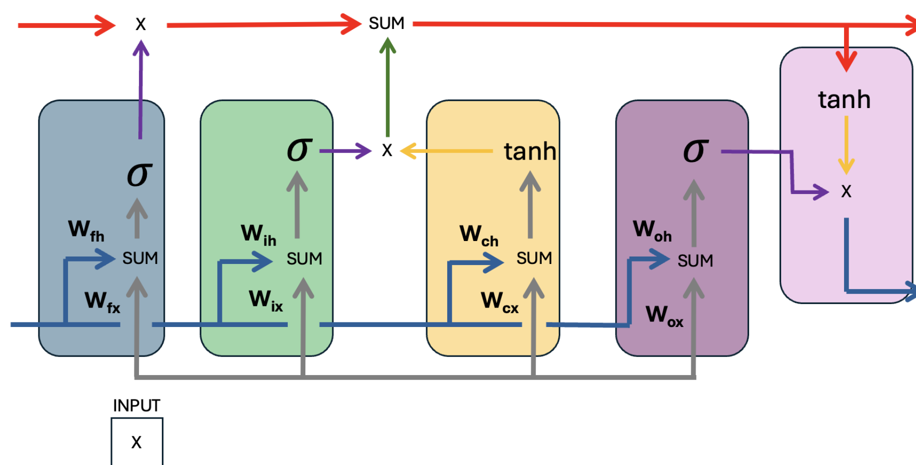

FIGURE 2.18: The long-term memory for one time step in an LSTM model - own illustration

At first look the LSTM model can be rather complicated to interpret as it has many parts and memory components that requires further elaboration. To give a clear presentation of the model each component will be presented individually and at last put together to provide a full picture of the model. Firstly, the two memory types that the model operates with will be defined, and secondly, the three stages of the model will be untangled to understand the mechanics and interactions between the short- and long-term memory.

2.11.1 The short- and long-term memories of the model

The first memory component of the LSTM model is the long-term memory that describes the continuous long-term effects for the model by the variable cell state. The variable is affected by two gates; the first gate affects the cell state by multiplication in the first stage noted as "**x**" and the second gate that affects the cell state by addition in the second stage noted as "*SUM*". The long-term memory is visually represented

for one time step in figure 2.19 by a horizontal line in the top part of the illustration. The long-term memory is represented by the red line and the rest of the components and stages of the model is blurred out for visual clarification.

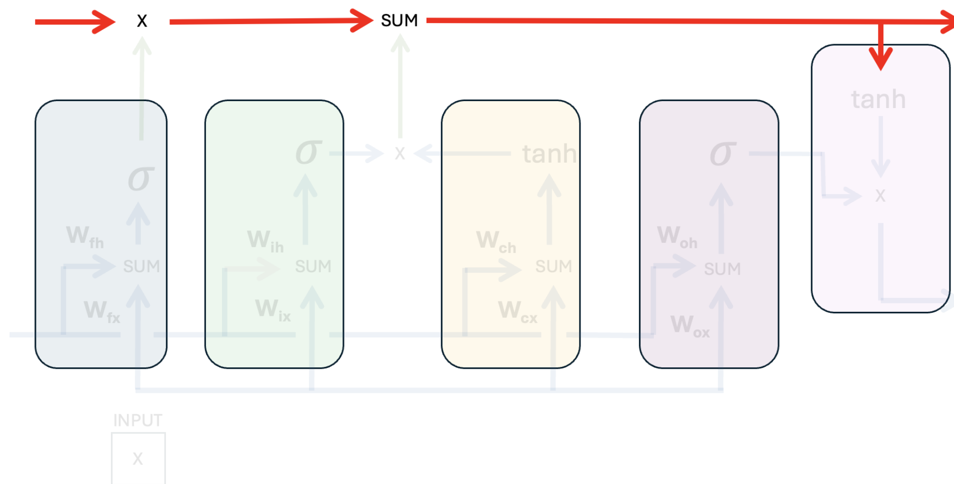

FIGURE 2.19: The long-term memory for one time step in an LSTM model

The cell state for the long-term memory is only affected by a *forget gate* which multiplies a value between 0 and 1 to the cell state and by an *input gate* which adds a value stemming from the current short-term memory. Since the memory is only affected by these gates, the problem of the vanishing and exploding gradient is accounted for. The design of the LSTM allows the cell state to go through multiple time steps without being scaled by a weight or bias that either makes the gradient explode or disappear. The LSTM model can therefore include long-term memories in the cell state without having the typical problems of the basic RNN model by more effectively flowing it through.

The second memory component of the LSTM model is the short-term memory which is called the hidden state. The hidden state in many ways works as in the RNN model as it is an input scaled by a given weight, however, unlike in the RNN it is not just a single input but an input for each of the three stages which the LSTM model contains in a single time step. The short-term memory is visually represented in figure 2.20 by a horizontal line in the bottom of the model - the rest of the model is blurred.

The hidden state is affecting all the three stages of the model by a single input in the first and third stage and a two inputs in second stage. The inputs that the hidden state gives are scaled by a weight coefficient for each gate that it affects. The last part of the short-term memory to the far right in the model is the value that the model determines the new short-term memory should be and will also be the final result or prediction of the model.

The final memory component to the LSTM model is the input variables. The input variables affect the same gates as short-term memory, however, with different weights than the hidden state weights. The variable inputs are visually represented in figure 2.21.

To better understand how each components of the LSTM are interacting in the model each stage of the LSTM need to be reviewed.

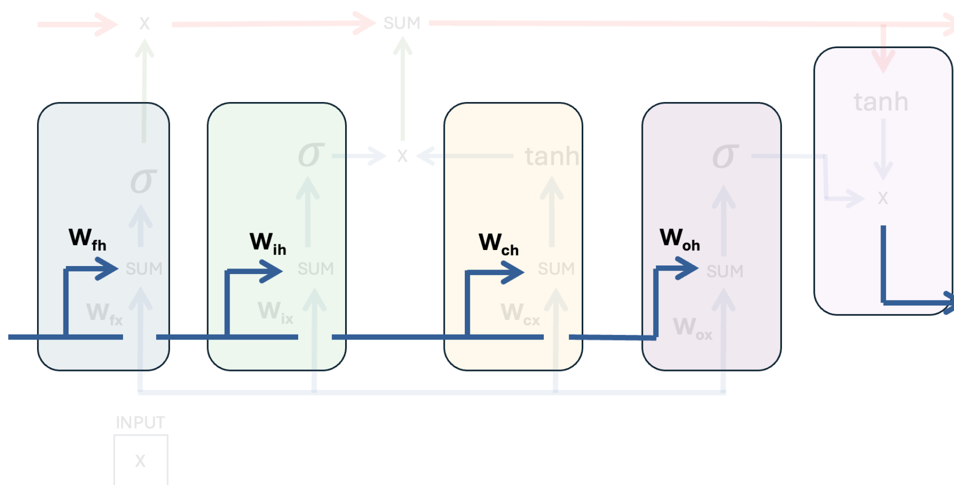

FIGURE 2.20: The short-term memory and hidden state weights for one time step in an LSTM model.

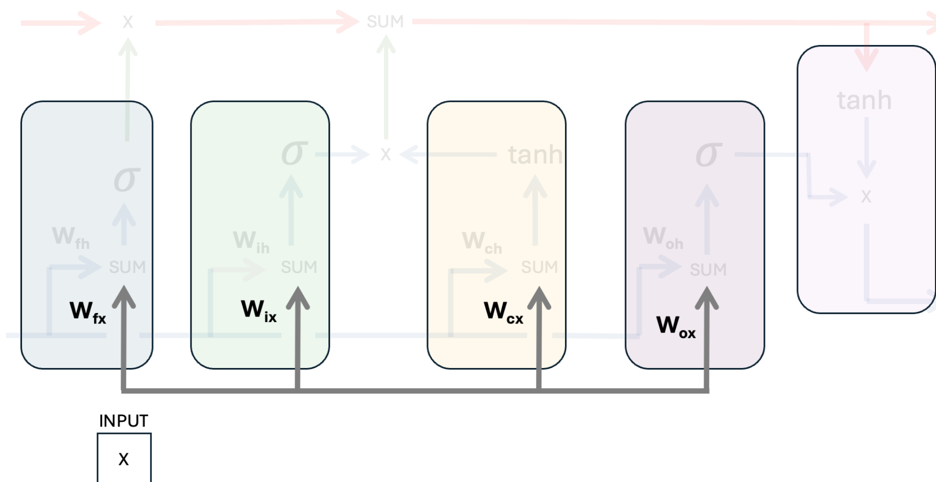

FIGURE 2.21: The input variables and input weights for one time step in an LSTM model - own illustration.

2.11.2 The Three Stages of the LSTM Model

The LSTM model's flow of information is moving through three stages that updates the values of the cell state or the hidden state. The first stage has the purpose of forgetting information from the cell state based on the values and weights of the hidden state and input variable. The second stage is the stage that adds new information to the cell state based on the values and weights of the hidden state and input variable. The third and last stage is updating the value of the hidden state given the value and weights of all three memory components.

The first stage is the forgetting stage and adds the values in the first gate noted as SUM of the hidden state multiplied by the weight matrix $\mathbf{W}_{fh}$ and the input variables multiplied by the weight matrix $\mathbf{W}_{fx}$. It is important to know that a bias term also is added in the SUM gate. After the values have been added together the matrix is then input into the activation function sigmoid to gain a value between 0 and 1. The value is afterwards multiplied to the cell state in order to forget part of the long-term

effects that has become redundant. The first stage is visually represented in figure 2.22. The function that executes the first stage can be expressed mathematically by the following equation:

$$\mathbf{f}_t = \sigma(W_{fh} * \mathbf{h}_{t-1} + W_{fx} * \mathbf{x}_t + \mathbf{b}_f) \tag{2.97}$$

Where f_t is the function that tells the percent of the cell state that is remembered, σ is the sigmoid function, W_{fh} is the weight matrix of the hidden state, $\mathbf{h}_{t-1}$ is the value of the hidden state from the output of the last time step, W_{fh} is the weight matrix of the input variables, $\mathbf{x}_t$ is the input variables for the current time step and $\mathbf{b}_f$ is the bias of the the first stage.

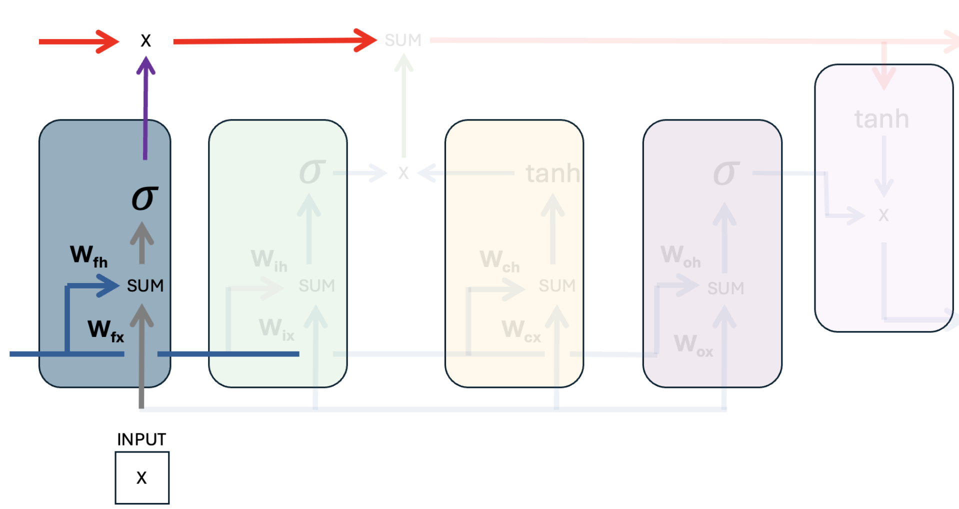

FIGURE 2.22: The first stage for one time step in an LSTM model.

The second stage is the input stage that adds new information to the cell state. This is done through two separate gates; the sigmoid gate that gives the percent of short-term memory to be remembered and the tanh gate that gives the value that should be remembered.

The sigmoid gate is the gate that estimates how much of the short-term memory that should be included in the cell state. The approach is the same as the first stage where the hidden state and input variables are summarised with respect to each individual weight matrix and input into the sigmoid function to give a value between 0 and 1. The output of the sigmoid gate is hereafter multiplied with the second gate. The sigmoid gate can be mathematically expressed by the following equation:

$$\mathbf{i}_t = \sigma(W_{ih} * \mathbf{h}_{t-1} + W_{ix} * \mathbf{x}_t + \mathbf{b}_i) \tag{2.98}$$

The tanh gate is the second gate that determines the value that should be added to the cell state. The hidden state and input variables are added together with a bias with respect to the individual matrix weight, however, it is then input into by the tanh activation function before being multiplied with the sigmoid gate's value. The tanh gate or cell state can be mathematically expressed by the following equation:

$$\mathbf{c}_t = tanh(W_{ch} * \mathbf{h}_{t-1} + W_{ch} * \mathbf{x}_t + \mathbf{b}_c) \tag{2.99}$$

Finally, the result of the second stage is added to the cell state. Note that the maximum value that can be added or subtracted from the cell state is 1 or -1. This would be given that the sigmoid function has the value of 1 and that the tanh function has the value of 1 or -1. The second stage is visualised in figure 2.23.

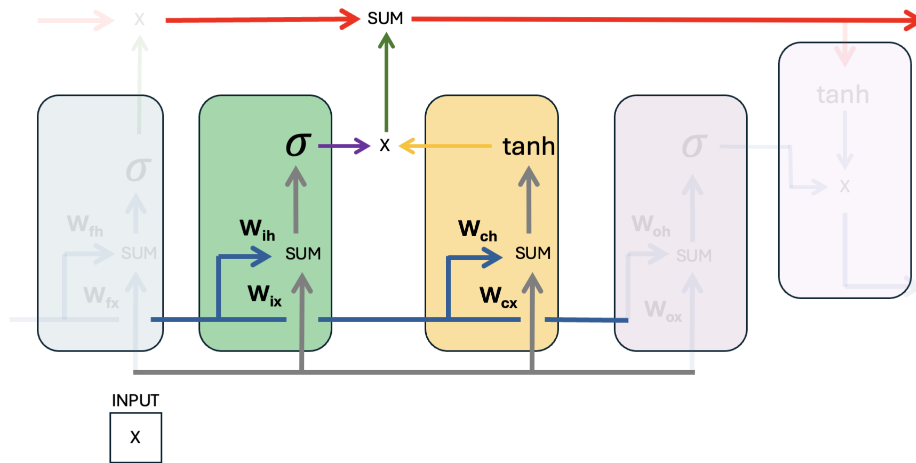

FIGURE 2.23: The second stage for one time step in an LSTM model.

The third stage is the output stage that collects the information from the cell state, hidden state and input variables to update the hidden state value. The stages has to gates as well a sigmoid gate and a tanh gate. The sigmoid gate work exactly like the other sigmoid gates with the hidden state and input variables having an individual weight matrix with a bias. The sigmoid gate can be mathematically defined by the following equation:

$$o_t = \sigma(W_{oh} * \mathbf{h}_{t-1} + W_{ox} * \mathbf{x}_t + \mathbf{b}_o) \tag{2.100}$$

The tanh gate has a single input of the cell state noted as $\mathbf{c}_t$. Thereby the value of the cell state is input into the tanh function and multiplied with the sigmoid gate. The final update to the hidden state and the output variable is therefore:

$$\mathbf{h}_t = \mathbf{o}_t * tanh(\mathbf{c}_t) \tag{2.101}$$

The third stage is visually represented in figure 2.24 and shows that the cell state and hidden state moves on the the next time step. It should be noted that at the final time step the value for the hidden state, $\mathbf{h}_t$, will become the prediction of the model.

2.11.3 The Mechanics of the LSTM Model

Now that the whole LSTM has been presented, the mechanics and operations of the model can be explained and discussed.

The three stages of the LSTM has the effect of, firstly, removing information from the long-term cell state, then secondly, adding information from the short-term hidden state and feature inputs and, finally, the hidden state is updated from the components; cell state, previous hidden state and input variables. This process which the LSTM performs is all done in the $\mathbf{h}_t$ node that was previously discussed in the basic RNN model. Thereby a whole LSTM network still has the same structure as a basic

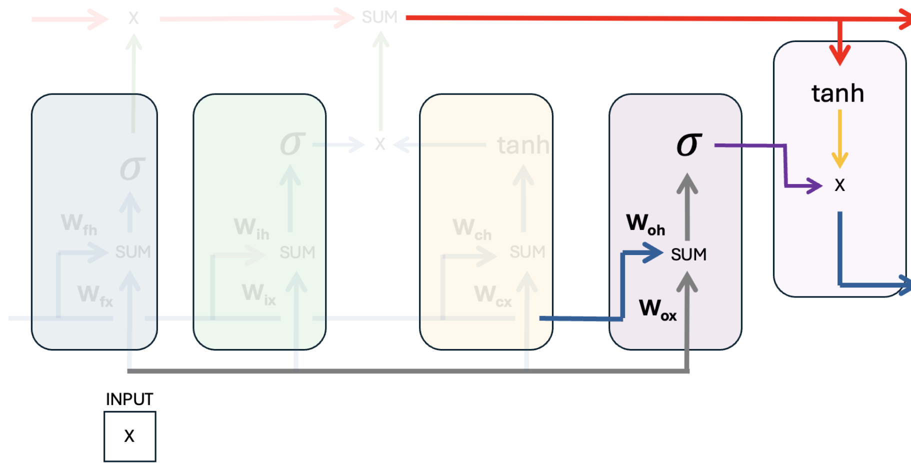

FIGURE 2.24: The second stage for one time step in an LSTM model.

RNN model as described in figure 2.16 in subsection 2.10.2. The difference is that the node $\mathbf{h}_t$ is changed from a standard feed forward network that uses a weight matrix and activation function to the structure of the LSTM with 3 stages for each h_t node. Revisiting the basic RNN model in figure 2.16, the model can be expanded to an LSTM network by substituting the hidden layer units, $\mathbf{h}_t$, with this structure. A representation of the neural network of a LSTM model is visually shown in figure 2.18.

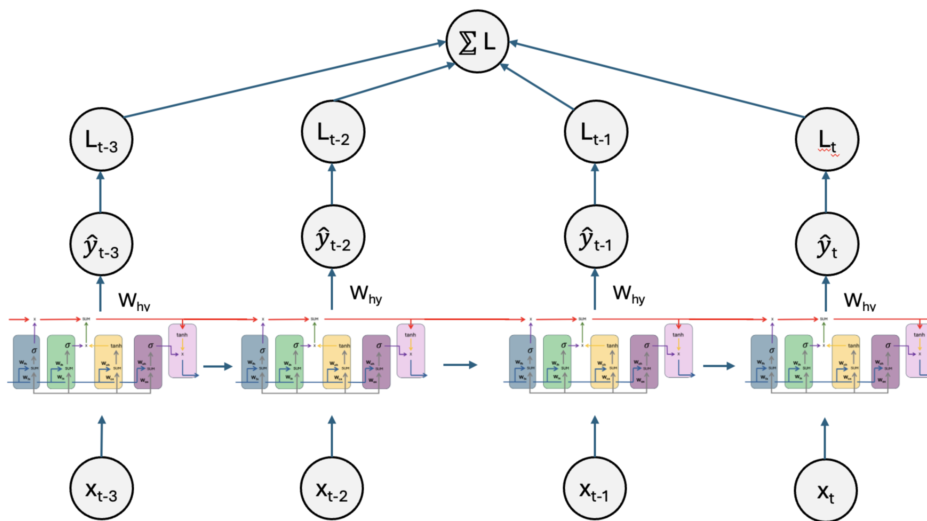

FIGURE 2.25: The second stage for one time step in an LSTM model.

As seen in figure 2.25 the cell state is consistent in all time steps and is only influenced by the forget stage and the input stage in each time step. It is not possible for the gradient to explode as there is no constant augmentation that increases the value exponentially. At the same time the model also significantly improves the issue of the diminishing gradient. If there is no long-term effects the cell state should go towards zero.

The hidden state is updated for every time step as the value of the previous hidden state, cell state and input variables adjust the hidden state in each time step.

In conclusion, the LSTM model has a structure that can accurately describe short- and long-term effects which can be applied to predict an output value. The LSTM model can like a feedforward network be used to with additional layers which raises the complexity of the model.

2.12 Exchange Rate Theories

The topic of exchange rates is very complex and different models and theories strive to explain how they work and are determined, however, it seems that the majority of theoretical suggestions are very inadequate in understanding the relationship between two currencies. Some argue that exchange rates are determined through traditional economic theory, others on fundamental macroeconomic variables and others purely on technical price movement analysis. The issue might not be that those suggestions themselves are wrong, but more that it is probably wrong to assume that one of those suggestions can single-handedly explain the relation between two currencies, hence if one can in some way combine theoretical models with fundamental and technical indicative variables, the explanatory framework might be one step closer to predicting movements of exchange rates - this is where the LSTM model might come in handy. This section will briefly provide some examples of central theoretical frameworks introduced in the literature in regards to exchange rate determination. Going through different and even opposing theories is intended in order to create a neutralized motivation of different ideas which could be implemented later, and so, the section should be seen as a starting point in determining variable candidates to implement in the modelling stage.

Definition of an exchange rate Before diving further into the theories evolving exchange rates, a quick definition is helpful; *an exchange rate is the rate at which one currency can be exchanged for another currency* (Chen, 2024). Most exchange rates are floating, i.e. they can freely depreciate or appreciate. The rates are affected both by domestic currency and foreign currency, and therefore it is a relative term. To evaluate a currency, it is compared to some other currency, and is often denoted with an acronym; e.g. USD is the acronym for the US Dollar just as EUR is for the euro. So the exchange rate between USD and EUR is denoted as EUR/USD.

Theories on exchange rates

Purchasing Power Parity (PPP) The PPP theory focuses specifically on the internal price levels in different countries as a determinant for exchange rates between them. Though it has ideas dating back to Ricardo, the official developer of this theory is the Swedish economist, Gustav Cassel (*Gustav Cassel - Econlib* 2022). The main idea is that in equilibrium, the exchange rate between two countries is set so the purchasing power is equal in each country. This equilibrium also requires *the law of one price* to be true, which might be subject to criticism. In the long run, this relation will be kept according to differences in inflation in the two countries, and so, according to this theory, exchange rates are determined almost purely by relative price levels. While there is plenty of logic in stating that inflation is a huge factor in determining exchange rates, there are most likely a lot of other factors needed to be included. The PPP is especially criticized by the Keynesian theories, in that it completely ignores the aspect of capital movements in between countries, which could arguably be just as important. Additionally, it is naive to claim that the purchasing power is equal in all countries and that there are no trade barriers and fees. To sum up, inflation seems to be an important factor, but additional features are required.

Interest Rate Parity (IRP) As the name suggests, the IRP lays focus on the interest rates in different countries as a driver for price determination. It connects interest rates with spot and forward exchange rates. It is called a *parity* since it deals with a no-arbitrage rule of exchange rates, and so, it relates largely to the trading aspects

of determining exchange rates. In general terms, it says that one should not be able to buy one currency and sell another at a future date (forward exchange rate) - and generate a risk-free profit. As such, it connects spot- and forward exchange rates with the interest rate of the given countries, and the mathematical parity is given as:

$$F_0 = S_0(\frac{1 + r_a}{1 + r_b}), \tag{2.102}$$

where F_0 is the *current* forward exchange rate, S_0 is the spot exchange rate, and r_i is the interest for country i. Note here, that this theory does not directly argue for the fundamental variables affecting exchange rates - rather it builds a central relationship, or parity, between the three variables, which should always be held true. Where PPP related exchange rates to the real economy, the IRP does so in the financial markets. As with the first theory, IRP also has its share of limitations and gaps; as an example, it is based on the assumption, that markets can freely flow capital in between each other with no regulation, fees or transaction costs. Furthermore, it assumes efficient markets, and that expected future exchange rates are known and precise. So as with the PPP theory, there is plenty of logic in the theory, and it argues for the inclusion of interest rates as an explanatory variable in determining the exchange rates.

Technical indicators In a more practical aspect, it has also become a centrally discussed topic whether or not technical indicators are better in explaining exchange rate fluctuations than the traditional fundamental ones. In fact, (Gehrig and Menkhoff, 2006) finds evidence that technical analysis has become the new "workhouse" on the foreign exchange markets. Specifically, they find that the majority of FOREX traders and fund managers primarily focus on technical indicators rather than traditional macroeconomic ones. While this *chartist* approach on pricing a currency might seem ridiculous to most economists, it simply cannot be denied to have *some* influence, when it is used by so many and thereby creates observed self-fulfilling outcomes. (Hsu, Taylor, and Wang, 2016) argue in this relation that technical indicators can exploit irrationality in trading markets, and so it is able to capture movements that are not so much related to economic sense, but rather psychological ones. This theory builds upon behavioural economics and it states that agents bear a lot of biases (e.g cognitive biases, rules of thumb, herding behavior, overconfidence etc.). So, this suggests quite significantly, that besides including economic, fundamental data, the model can gain further strength from also including technical indicators. Specifically which indicators to include, will be touched on in Section 4.4.

Chapter 3

Literature Review

In this chapter focus will be on reviewing the literature that concerns financial modelling in ML. The primary focus will be on how the literature is applying fundamental- and technical indicators to financial data in the FOREX market, primarily concerning LSTM models. The knowledge gained from this chapter should be a key element in comparing the performance of this study's LSTM model. The chapter begins with a short introduction to the fundamental- and technical analysis. Then it proceeds to review articles regarding choices and discussions for fundamental- and technical indicators in financial modelling. At last there will be a more specific focus on hybrid indicator LSTM models in the FOREX market. Fundamental analysis could be described as the cornerstone of investing as it is the analysis which centralises the core system surrounding a market. For FOREX trading it is the analysis of the core macroeconomic variables, both quantitative and qualitative, which affects the currency relationship. Fundamentals are in their simplicity all aspects of the future development of a currency - thereby making it the perfect indicator for long-term investing. However, in practical terms, the market is not always efficient as many participants view the market from different perspectives and the irrationality and behavior of participants in the market can vary drastically. For this reason, the ability to analyse and quantify behavior and irrationality has become useful and many, especially on short-term, are favouring a more technical analysis with the use of technical indicators (Drakopoulou, 2015).

Technical indicators are mathematical calculations on historic data which has the objective of predicting future market behaviour in order to further predict future price movements more accurately. It is inherently different from fundamental analysis as it does not predict prices from macroeconomic variables or the core basics behind a market but purely the behaviour of the market. Whether technical indicators will be able to predict future patterns and behavior in the market or not is a difficult question to answer, however, it is a widely used technique by traders. Technical analysis rests on the three assumptions which are 1; "Market action discounts everything", 2; "Price moves in trends" and 3; "History repeats itself" (Drakopoulou, 2015).

The optimal variables and parameters of the fundamental and technical analysis can be difficult to select for the model, however, one can look at the past literature as an adequate measure of which indicators are performing well on financial data in ML. In the following review, the amount of periods used in an indicator is marked in parenthesis.

For fundamental indicators in FOREX trading, many cases revolve around the growth and stability of the countries' currency. Often FOREX traders look at macroeconomic indicators as GDP, inflation, trade balance, the interest rate, manufacturing index

and the producer price index - some of which match well with the exchange rate theories earlier mentioned. It does not not revolve around the day-to-day price movements and is therefore usually considered better for medium to long-term trading/investing (Snow, 2024). (AbuHamad, 2013) found that including macroeconomic signals in a FOREX trading model that only used lagged closing prices yielded significant improvements in its predictive power. (Yıldırım, Toroslu, and Fiore, 2021) uses fundamental indicators such as interest rates, inflation and stock indices such as S&P 500 and DAX to support finding the relationship for EUR/USD. They found that using macroeconomic indicators yielded a model hit rate of 50.69%, however, with a relative higher standard deviation of 3.72 percentage points, they conclude that a model of purely macroeconomic variables would not be optimal to trade on. (Abednego, 2018) made a fundamental and technical trading algorithm that both yielded profit, however, the technical trading robot was superior due to its more stable profit curve. They tested the two algorithms for 30 days and concluded that the fundamental model required a longer time frame to be effective.

Looking at the technical indicators, there is no limit to the number of indicators or adjustments which can be made to optimise them. Unlike fundamental indicators, technical indicators can be used in the short term on small periods of data, however, also performs well on medium and long-term data (Snow, 2024). (Drakopoulou, 2015) describes moving averages as one of the most recognised indicator and useful in both validating past trend and future momentum of price movements. They further propose that that 20 days or less are a fine short-term momentum indicator while a 100 periods or more are good for analysing long-term momentum. They also promote the *MACD* indicator as it has proven to be one of the most simple yet still reliable indicators. As per standard, it calculates the difference between the 12 period and 26 period EMA to estimate if there is an upward or downwards momentum, however, it struggles with over-bought and under-bought market. (Yıldırım, Toroslu, and Fiore, 2021) uses technical indicators indicators on a daily frequency to forecast the price of the EUR/USD relationship. They divide their indicators into three groups; lagging, leading and volatility indicators. The lagging indicators are MA(10) and MACD(12, 26) which analyse past trends in hope that trends will continue. The leading indicators are ROC(2), Momentum(4), CCI(20) and RSI(10) which are indicators for predicting momentum of short-term price movements. The volatility indicator is Bollinger Bands(20) which measures the volatility of the price. The model achieves an average hit rate of 52.18 % with a standard deviation of 1.96 percentage points.

The best performing models according to the reviewed literature are models that have the combination of both fundamental and technical analysis. The idea is that they support each other in the that the fundamental drive the long-term price and that the technical indicator can identify the behavior on the short term (Snow, 2024). (Yıldırım, Toroslu, and Fiore, 2021) makes two of such models. The first model combines fundamental indicators and technical indicators in a single hybrid LSTM model which collectively outperforms the single fundamental and single technical indicator models when they are separated. It achieves a trading model that generates a hit rate of 53.05% with a relative high standard deviation of 7.42 percentage points. The second model that they propose is a model which is a rule based model and which only combines the output of the technical and fundamental indicator making it take action only when the models have the same signal and staying passive if the signals are opposing. The model achieved the highest hit rate of 79.23% with a standard deviation of 15.06 percentage points. Even though the the rule based model

outperformed the hybrid model the hybrid model had double the amount of trades as the rule based model - making it a more active model that would execute orders more often.

Chapter 4

Data description

4.1 Preprocessing and Feature Engineering

Having a clear and consistent view of the data characteristics and properties before delving into the modelling is a fundamental and important step. In standard econometrics, what one often cares about is correlation, multicollinearity, and stationarity, both for their intrinsic properties for reliable estimates but also as a clear requisite for constructing models such as *ARIMA*. For NN's, this is a different story; therefore, this section will highlight some of the key differences between these two pillars of statistics in how they handle and process data. Next, some key data transformations are both described in terms of how they transform and what the impact of the transformation is. The section will conclude with date time handling and the *look ahead bias*, which can massively influence model performance.

4.1.1 Two Pillars of Statistics

A core property of financial data is that it captures intrinsically complex relationships between variables, which, more often than not, are non-linear. Specifically, this property can be challenging to model as it requires strong assumptions to be imposed on the model if standard econometric methodology is applied. On top of this non-linearity, the data tends to be high-dimensional. In a lot of situations, one can be placed in a scenario with more variables than observations. On top of this, financial data has a high degree of *noisiness*, specifically; one can extract the return of an asset using realised returns from market data, but knowing what the real expected return is can be a challenge. In most circumstances, it is difficult to attribute a specific event to a change in the returns of an asset, even more so when macroeconomic variables are introduced to explain financial data. From economic theory, it has been researched and theorised for decades what affects asset returns and macroeconomic variables, but even so, it can be difficult to provide a comprehensive theory to explain them as they are noisy (Lezmi and Xu, 2023).

Highlighting these properties of financial data is not a problem akin to econometrics but also ML. Being that as it may, ML's theory is built to withstand these data properties and actually outperform most traditional statistical methods on noisy, non-linear, and high-dimensional data, which makes it a very attractive choice (De Prado Marcos, 2018).

Correlation and Multicollinearity

As known from classical statistics, the presence of multicollinearity, in which two or more features exhibit a high degree of correlation can pose problems in inference. This manifests itself in the interpretability of the coefficients, since the researcher no longer is able to determine the effect of a specific independent variable on the dependent variable. This problem does not translate itself over to ML methods. Already in the naming of the variables it is evident that econometrics and standard statistics has a different view, *independent and dependent variables* rather than *target and feature variables*. Within ML, the goal is to achieve high predictive power of the model created over inference, which comes at the cost of interpreting the results of weights and biases, which are updated during the training process. These results no longer tell us anything about which feature has the largest effect on the predictions. Therefore, if interpretability is not a concern or area of interest, it will not be beneficial to remove any columns as ML will find the underlying structure of the data (Ghanoum, 2022). This brings forth yet another important distinction between econometrics and ML; econometrics wish to impose a specific structure on the data and ML wishes to unveil the structure of the data, *parametric* and *non-parametric*. This has important implications in how the models are evaluated, and in later sections in which the models are built, the procedure of the performance evaluations will further highlight this key difference.

Stationarity

Achieving stationarity within econometrics is an absolutely vital part of preprocessing. Achieving I.I.D. series is important for two reasons: i) it produces reliable estimates as the series have the same properties over time; and ii) models such as ARIMA impose these specific assumptions upon the model, therefore making it a necessity. The fact that stationary time series produce reliable estimates can also be a desirable property for a LSTM model. Browsing through the existing literature for FOREX predictions using LSTM, often times the authors display a plot of the actual against the predicted values over the time series. Here, it can generally be seen that many LSTM models make predictions that are simply an AR(1) process. This is not necessarily a recurrent event across the literature, but a potential pitfall to beware of. As a best practice, it can be beneficial to achieve stationarity, as is the case with econometrics.

Even though stationarity exhibits these desirable properties, (De Prado, 2018) highlights that achieving stationarity comes at the expense of wiping out all memory from the time series. What is meant by memory in this context? Time series has a long history of prior levels, which has fundamentally changed its mean; achieving stationarity through traditional methods wipes out these properties of the series, thereby reducing the memory. (De Prado, 2018) argues that this trade-off is not an attractive one. Instead, it argues for a different method, *Fractional Differentiation*, which has the purpose of achieving stationarity through differencing the data, but just enough to achieve stationarity validated through the *Augmented Dickey-Fuller test*. Again, one is confronted with a trade-off: achieve stationarity and produce reliable estimates, or achieve weak stationarity with a minimum amount of differencing and maximise memory of the time series.

Some papers, including (Walasek and Gajda, 2021), argue for the usage of fractional differentiation and highlight increased performance by applying this method. Even

though this is the case, LSTM are quite robust to the data fed into them, and on the contrary, they do not require the data to be stationary before usage. This comes at a very specific expense: interpretability. Deep learning is often referred to as a black box, meaning that what is done in terms of computation and fed out through the output layer often remains a mystery since a human wouldn't be able to make the same amount of computations, which often are in the millions for deep learning methods.

4.1.2 Look Ahead Bias

Often, when researchers collect data to perform statistical analysis, a fundamental concept is regularly overlooked, but it can introduce bias and an augmented reality of the historical performance of a security or an economy. Collecting data through OECD, FRED, ECB, and other data sources, the macroeconomic indicators are marked with the first day of the month in which the variable value is displayed. This is of course correct when you look at it from a historical perspective, but when conducting research that captures daily effects, this would lead to corrupt or biased estimates, why? i) By applying this technique, the researcher uses information that was not readily available at that specific time; more accurately, the inflation data for January in an arbitrary year cannot be published on the 1st of January but rather after that month has passed and is usually published in the middle of the following month. Researchers are therefore subject to *Look Ahead Bias*; by applying the inflation data for January to the 1st rather than its publishing date, it creates a distorted reality of what traders, market actors, and policymakers had at that point in time. We try our best in avoiding this by manually adjusting data according to publication dates, whenever possible.

4.2 The output variable - EUR/USD

The following section introduces the most central variable in the model, namely the exchange rate between USD and EUR. Firstly, some descriptive statistics will be walked through to identify the basic behaviours of the raw time series. Secondly, in order to use it properly in the final modelling, quite a few transformations and changes has to be made, which will be discussed. This includes converting the exchange rate into daily changes which then helps create some classification categories for when the price is moving up, staying the same or decreasing. To properly define these classification categories, some threshold values has to be determined. This will be done through the help of some statistical properties of the daily change time series. Note also, that the model will take three classifiers as outputs: one that classifies an increase, one that classifies the price change staying neutral and lastly one that classifies a decrease.

4.2.1 EUR/USD - raw data review

The daily closing price for the exchange rate, EUR/USD, ranging from 2005-2024 is fetched from the reliable database, *Federal Reserve Economic Data*(Louis, 2024). The variable specifically denotes US dollars to one Euro, hence a decrease in the rate indicates a weaker EUR. The entire raw time series from 2005-2024 can be observed in figure 4.1. One can quickly notice how the exchange rate went through some volatility during the financial crisis and the following years when EU and US fought hardly against very threatening recessions - and they did so in quite different ways. Finally

around year 2015, global economies seemed to be stabilising again. This surprisingly ended very quickly again in 2020 when the Covid-19 pandemic started and not long after a huge energy and war crisis, that sent inflation levels to the roof. In response, US very aggressively lifted interest rates, which lead to huge demand for the USD, resulting in a historically strong USD (in fact reached parity for a while). EU eventually also lifted interest rates, and the exchange rate retraced back again.(plus500, n.d.)

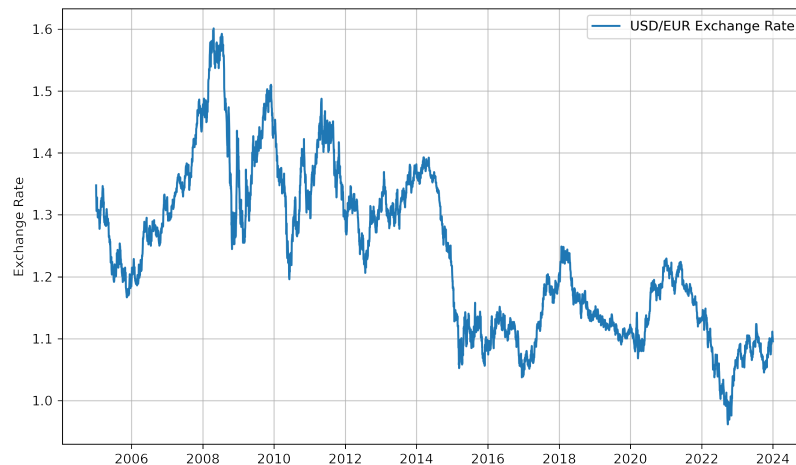

FIGURE 4.1: EUR/USD Exchange rate. Daily frequency. 2005-2024. Data: FED St. Louis. Own illustration

Additionally, the daily *highs* and *lows* are fetched from Yahoo Finance in order to produce certain technical indicators used as inputs in the final NN model.

4.2.2 EUR/USD transformations and preparation

The model in this paper does not, like a lot of others, try to predict the direct price of the exchange rate each day - rather it tries to predict the *direction*. After all, that seems much more realistic to predict, yet arguably still just as useful as the alternative. So in order to achieve this, a lot of transformation and engineering has to be made on the raw time series.

First of all, in order to convert the time series from being daily prices into daily changes, a difference between current exchange rate closure and the previous exchange rate closure is taken (i.e. $EUR/USD_t - EUR/USD_{t-1}$). It is deeply interesting to dive into the statistics of this newly created time series. Because it is now the daily changes, one can from these statistics figure out how these have behaved over the last 20 years - specifically, the distribution of price changes and some features in this context. But very importantly; when modeling the exchange rate later on, in order to determine the direction each day, it is necessary to convert the variable into classification categories. These categories will be split up by certain price change thresholds, and so when going through the statistics of the daily price changes, these thresholds can more easily be determined. *(to clarify on thresholds: let three classes be defined by 1) increase in price, 2) no change in price and 3) decrease in price. Then to split up these classes, it is necessary to determine two thresholds that define what price change intervals are identified by being price increases increases, price neutrality or price decreases)*. The idea behind this converting is that the final LSTM model is supposed to classify the correct category for as many days as possible through a softmax activation function.

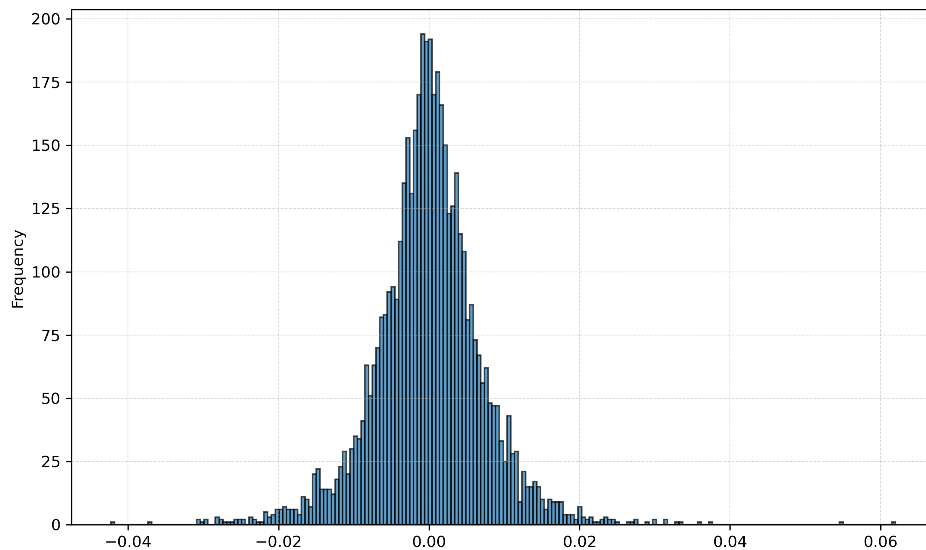

FIGURE 4.2: Distribution of daily changes in EUR/USD. Own illustration

Now, lets start by inspecting the distribution of the time series containing exchange rate changes by observing figure 4.2. One will very quickly notice how the distribution seems to be very close to a Gaussian one with no obvious skewness in either direction and a mean price change around zero. This already now indicates that there might not need to be any skewness in the chosen thresholds (this creates the expectation that the amount of buy signals equals that of sell signals when thresholds are set symmetrically around zero). To confirm these observations it is useful to extract the values of some summary statistics, and indeed, these values are simply too perfect; the mean of price changes is *-0.0001*, the median ≈ 0, and the 25th/75th percentiles are *-0.004/0.004* respectively. Remembering that this data is real world data, it is very rare to see a distribution having such perfect symmetry around its center - which the center itself being very close to 0. The median also suggests that the amount of daily price decreases are very close to that of increases - and to confirm this; the exact number of decreasing price change days is $2,364$ while the number of increasing price change days is $2,353$. So over a period of 20 years, the exchange rate has experience almost exactly as many increasing changes as decreasing. It almost seems as if some model generated this distribution.

So up until now it is expected that the thresholds will approximately be symmetric around 0 given the distribution and not being wrongly biased in either direction. Next step is deciding how distanced the thresholds should be around the mean. One starting point could be to set the two thresholds on some quantiles of the distribution. Lets first set the two threshold values as the 25th/75th percentiles (this way all price changes that fall below the 25th percentile are categorized as decreases, all price changes that fall above the 75th percentile are categorized as increases and the remaining categorized as staying the same). To look at another example as well, lets also assume two threshold values on the 40th/60th percentiles, which will be much narrower in comparison. To compare the two threshold examples visually, see figure 4.3. The left distribution illustrates the first example and the right illustrates the

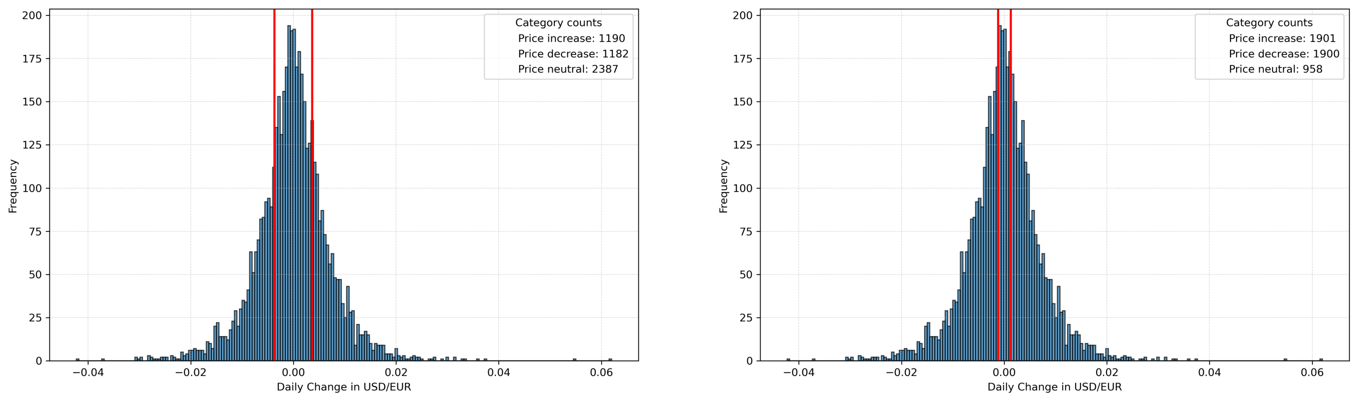

FIGURE 4.3: Two different threshold sets. *Left: 25th/75th percentiles. Right: 40th/60th percentile*. Own illustration

second. Note here, that the legends indicate, over the last 20 years, how many days the exchange rate category would be set to *increase*, *neutral* or *decrease* in price for the two examples. In the 25th/75th percentile example, the model seems to categorize the majority of outcomes as neutral, and hence it will be less active in taking positions. So, without knowing anything regarding the optimal threshold values, it seems more *"exciting"* in the other example, where the big majority of category counts fall into buy/sell signals, but it is unknown whether this is more efficient or not. The central point is here, that the optimal threshold values cannot theoretically be found, it is a decision that needs to be made when iterating the model several times with different thresholds.

When the correct threshold values have been found, the time series will go through a final transformation; all the daily price changes are categorized into the three mentioned classes, i.e. the variable is converted from numeric to categorical. Specifically the model shall output 0 when price falls, 1 when it stays the same and 2 when price increases. This format is also fed into the training set, in order to train the model on the same relationships as it is later supposed to predict on.

4.3 Macroeconomic and fundamental data - inputs

The following section introduces the macroeconomic and fundamental datasets used as inputs in some of the models computed. A short motivation for including the variables is described, then some very basic presentation of the raw data is highlighted. If anything noteworthy occurs during the time period, this will also be highlighted. Then, the transformations made to each time series in order to prepare it for modelling will be explained. Lastly, providing sources for each dataset ensures that all inputs are reliable and correct. As the section includes an extensive list of figures the visual illustrations can be found in Appendix A.2-A.7

4.3.1 Inflation, CPI announcements

Consumer Price Indices (CPI) for the US and EU from 2005 to 2024 are fetched from their respective central banks; Federal Reserve St, Louis and ECB. Due to limitations on data availability, the rate is *monthly* for the US and *yearly* for EU. While these do

not match, it is assumed that the effects will be equal. The two time series are displayed in figures A.1 and A.2. Inflation rates are included to hopefully catch effects from relative purchasing powers in the areas. As seen in both figures, inflation rates took some huge hikes during the energy crisis following the war in Ukraine, during which time the exchange rate was also more volatile, so at least some effect can be expected. Now, in order to account for the look-ahead bias mentioned earlier, the data is adjusted to match all announcement dates. This ensures that all information is fed into the model at the same time it did so in the real world and this should help catch shock effects better. Lastly, the data series are taken through stationarity tests and scaled down to 0-1 which is preferable for the LSTM computation.

4.3.2 Unemployment rate announcements

Monthly unemployment rates from 2005-2024 are fetched from Federal Reserve St. Louis for the US and ECB for EU. The two time series are seen in figures A.3 and A.4. The overall pattern of the two series match to some degree, however, there are some larger deviations in terms of magnitude. The years following the financial crisis seem to be hit the EU hard for longer which might be due to the economic distress in some of the southern EU countries like Greece and Spain (Picardo, 2024). On the other hand, the Covid-19 pandemic hit the US much harder than that for EU, but was very quickly recovered again. Overall, unemployment rates are good indications of economic activity and central banks might use these rates to evaluate monetary policies which might directly affect exchange rates (Clay, 2024). As with CPI, the data here is adjusted to announcement dates to better catch the reaction on the exact days of publication. The announcements dates are also drawn from FED and ECB.

4.3.3 Yield curves

The 3-month, 10-year and 30-year yields between 2005 and 2024 are fetched from Alpha Vantage API for US and ECB for EU. The yields stem from US Treasury Bonds and AAA official bonds for the Eurozone. Both correspond to what is often referred to as *risk-free* bonds. All 6 yield series are displayed in figures A.5 and A.6. Again, one can observe that there are some correlations both between the different yields and the two areas. The yield curves tell a story about the interest rates as they are very closely related. Interest rates can be seen as the price for a currency, and should thereby have a close relation to the currency exchange rate. The theory of interest rate parity also suggests that its an important predictor.

4.3.4 Short-term interest rates

The Federal Funds Rate (FFR) for US and Main Refinancing Operations (MRO) for EU are included as well from FED St. Louis and ECB respectively. The argumentation for these are quite similar to the one for the yields. They are direct representations of the interest rate situations and are set solely by the central banks, and so, they should have some impact on exchange rates. The illustrations of FFR and MRO are displayed in

4.3.5 Stock Indices (Euro50 and S&P500)

To also include financial assets and indicators for the overall entire financial markets, S&P500 for US and Euro Stoxx 50 for EU are fetched for 2005-2024 from Fed St.

Louis and Google Finance respectfully. The frequency is daily. SP500 contains the 500 biggest listed companies in the US while Euro Stoxx 50 contains the 50 biggest listed companies in EU. The development of the two is illustrated in figures A.7 and A.8. Note here that the index reference for SP500 is shifted to a later date than the original, and so the value is lower. This does not have any effect though. The overall stock indices indicate well the optimism in markets and could therefore have quite a strong relation to the capital flows into the two currencies. Financial data is in general very noisy and exponential in its nature, so preparing these carefully before modelling is crucial.

4.3.6 Volatility Index, VIX

To also include some market volatility measurement, the volatility indices for S&P 500 and Euro Stoxx 50 are also brought in as inputs. These are famously known indices that are released by trustworthy sources (in fact, it's STOXX themselves that produce the one for Euro Stoxx 50). The VIX is included to account for periods of higher uncertainty on the financial markets which might spill over on some fluctuations in the exchange rate market as well. The two VIX indices are displayed in figures A.9 and A.10. As can be observed, the two developments are very closely correlated with primarily two periods of high volatility; the financial crisis and the Covid-19 pandemic.

4.3.7 Brent Crude Oil

Lastly, to account for the commodity markets, which have been in the highlight the last couple of years, Brent Crude Oil daily prices have also been included. The variable acts as a proxy for the general commodity market, which seems reasonable as it seems to be correlated to some degree with alternative commodities. The development is given in figure A.11 where one can observe that it has had multiple periods of big ups and downs. Oil is a direct contributor to inflation, and so it becomes an important factor in monetary policies and expectations. Furthermore, US has become quite a strong oil exporter while EU has become a strong oil importer, so price changes in oil must have some larger effect on the trade balances and flows between the zones, thereby also affecting exchange rates.

4.4 Technical indicators - inputs

The following section introduces the technical indicators used in the LSTM model. A short reasoning and motivation is given for including the indicators. Afterwards, the formula and mathematical calculation for the given indicator will be shown. At last, a visual representation of the indicator for the chosen time period will be shown and interesting periods as well as features be commented on. The technical indicators - momentum, ROC parabolic SAR and Bollinger Bands - were excluded from the study as there has been a focus on reducing the dimensionality of the data set to reduce computation. Additionally, these indicators were less commonly used according to the literature review and caused the LSTM model to become overparameterized, leading to overfitting. Additionally, we tested the model with these for confirmation, and indeed, they did not seem to increase the performance in significant ways.

4.4.1 Simple Moving Average, SMA

The SMA is a lagging indicator that can help identify previous trends from the output value (Fernando, 2023). In this study's LSTM model it is expected to assist the model by making previous trends more clear. If this variable was not included, the model could have issues in finding previous trends as there is a lot of volatility and white noise in most of the variables chosen. Both the SMA(10) and SMA(20) were chosen as to give the model two different short-term trends. The reasoning for choosing two short-term SMA's is that the trading frequency is daily so long-term indicators become less relevant. As for the medium-term indicator other indicators have been chosen to capture these effects.

The simple moving average is calculated by the average price over a chosen period. The SMA(n) is calculated for any n-period by the sum of the last n observations divided by n:

$$SMA(n) = \frac{A_1 + A_2 ... A_n}{n} \tag{4.1}$$

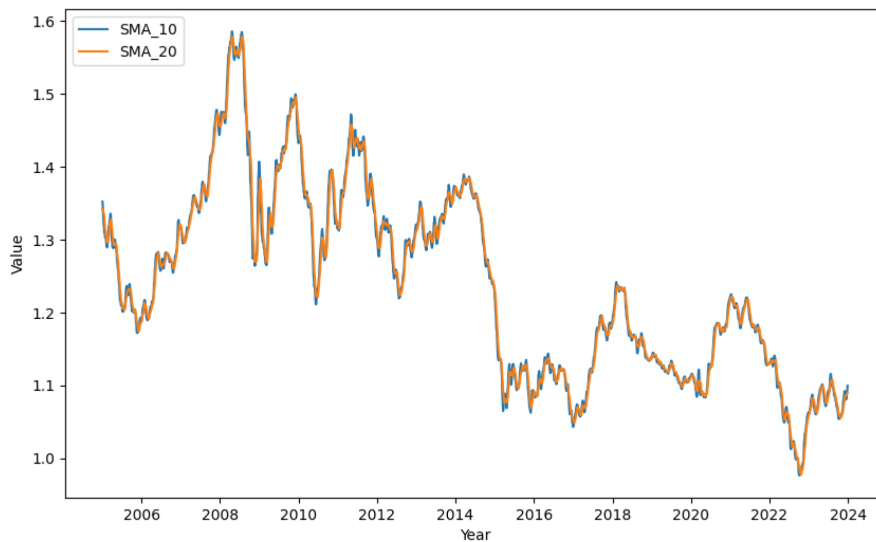

FIGURE 4.4: **SMA(10) and SMA(20)**. 10 and 20 days simple moving average of EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration

The two SMAs are very similar as they are derived from the same value. However, the SMA(20) shows a more smoothed curvature which shows longer trends when compared to the SMA(10). The LSTM model should be able to utilise the two technical indicator to analyse and navigate in two different short-term trends. Lastly, the indicators are taken through stationary tests and scaled down to 0-1 like the rest of inputs.

4.4.2 Moving average convergence/divergence, MACD

MACD is a lagging indicator that identifies price trends, momentum and entry points. It is expected that the MACD(26,12) will assist the LSTM in differentiating

between short-term and medium-term price trends and momentum. The MACD(26,12) indicator consists of the difference between two exponential moving averages(EMA). EMA is a weighted moving average that reacts more significantly to recent price changes compared to the SMA. The formula for the exponential moving average is:

$$EMA(t) = (EUR/USD_t * \frac{Smoothing}{(1 + Days)}) + EUR/USD_EMA_{t-1} * (1 - \frac{Smoothing}{1 + Days}) \tag{4.2}$$

Note that for a EMA(12) the process of analysing the previous $EMA_t - 1$ is done for 10 periods. Smoothing is, as standard, set to 2 in this study, emphasising that new inputs are more important without giving previous input too little weight(Chen, 2024).

The reasoning for using the MACD and not including any EMA indicator is that the single EMA is known to create a bias that produces false trade signals by wanting to mitigate this as much as possible the MACD is used as a substitute too some degree. However, the MACD also has some insight full analysis on its own as it is great for trade executing and analysing reversion of trends (Dolan, 2024).

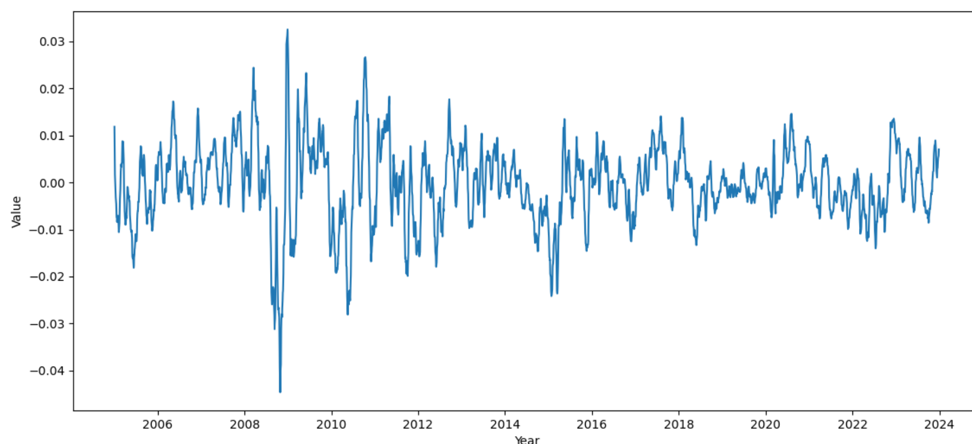

FIGURE 4.5: **MACD(26,12)**. 26-12 days moving average convergence/divergence of EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration

The MACD show values moving around zero as it only show the difference between the EMA(12) ans EMA(26). It shows the clear trend and momentum of the price especial in periods of shock to the economy. it was consider to have involved a EMA(9) as a signal indicator for the MACD, however, with the idea of minimising variables and noise this was disregarded.

4.4.3 Relative Strength Index, RSI

The RSI is a short-term leading indicator that is used to measure momentum and if a price is under- or overvalued in regards to recent trading patterns. The indicator has a value between 0 to 100 where 30 is considered an oversold price that leads to a buy signal and 70 is considered an 70 oversold price that leads to a sell signal. The model

will not know if when the price is considered under- or oversold but is expected to find patterns in the movements of the RSI.

The RSI(10) is calculated by the following formula:

$$RSI = \frac{100}{1 + \frac{9*\text{Previous average gain}+\text{Current gain}}{9*\text{Previous average loss}+\text{Current loss}}} \tag{4.3}$$

The losses are recorded as absolute numbers. Any period without a gain or loss is assigned a value of zero. If a period has a loss, it is recorded as zero for that period's gains and a gain is recorded as a zero in the loss for the period. The purpose is to smooth the results so that only strong trends in either direction come close to 0 or 100.

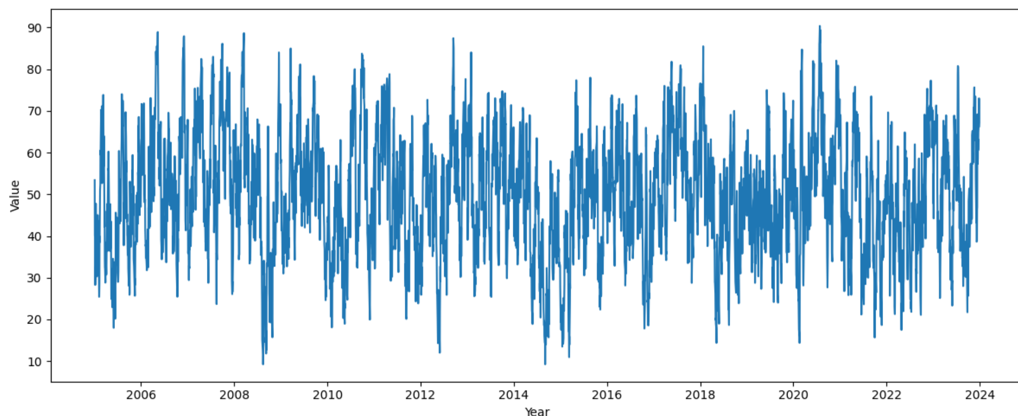

FIGURE 4.6: **RSI(10)**. 10 days relative strength index of the EUR/USD exchange rate. Data: FED St. Louis. Own calculation and illustration

In this study the RSI(10) is used which has a shorter time frame than the standard RSI and thereby identifies over- and undersold price movements quicker. This is done as the frequency is daily and the LSTM model is expected to need a relatively faster reaction time in order to capitalise on momentum opportunities in the FOREX market (Fernando, 2024).

Chapter 5

LSTM Predictive Trading Strategy

Now that the theoretical foundation together with a description of the input features has been established, it is time to build the actual LSTM predictive model. This chapter will begin with a walk-through of the framework from which each model was built, followed by providing an overview of the methodology and reasoning behind each step before computing the models.

5.1 Model Framework

In the subsequent section, the results for each model will be presented; therefore, it is essential to have a solid grasp of how each of these was built, including the reasoning behind several factors that are determined solely by the modellers.

All of the models that will be presented were trained using *Google Colab* for increased computational ability. Using this enables the utilisation of a *Graphics Processing Unit*, (GPU), which is tailored for tasks that require many computations. Specifically the L4 from NVIDIA, which was used during the training process in this thesis. When training a single iteration of a model, it will not be necessary to use this type of unit; rather, it can be trained using a normal CPU from a personal device. But as will be shown later, not just one model has been trained during this process, but multiple, in order to optimise these networks as much as possible.

Preparing the Data

Before feeding the data described in Sections 4.2 and 4.3 into the models, they need to be prepared specifically for this type of network. The process starts by splitting the target variable from the input variables. Next, a seed is set as a global variable for each model, ensuring reproducible results for continuous iteration optimization. Additionally, the dates from the data set are excluded entirely; these are only used to establish the correct sequencing across the entire time period; furthermore, LSTM's cannot handle values in string format, only numerical.

As mentioned in Section 4.2, the target variable approximately has a Gaussian distribution, so it is essential to select an appropriate threshold value for the target variable, as suboptimal thresholds can lead to biased results. The strategy for splitting the data into the three classes: sell, buy, and no action is to create an even distribution across the classes. The rationale for this is that having a specific class over-represented will force the model to predict that class in almost all circumstances, since it produces a lower loss in the cost function than predictions that spread out across the classes. This is not equivalent to a model actually learning from the data,

but rather finding a loophole that leads to uninteresting results that cannot be used to formulate an actual trading strategy. The percentiles chosen to achieve this even distribution were: 34.1th and 66th. This created class representations with 1601 increases, 1596 decreases, and 1559 no action observations. These class distributions are not entirely evenly distributed; this problem arises from the property of a Gaussian distribution: towards the mean, the frequency of observations increases drastically, and even small incremental changes can lead to highly varying class distributions. Furthermore, many iterations of different models were tested using different thresholds to further determine the optimal threshold values, leading to this conclusion as other values lead to over-representations of some classes. These thresholds were then used to create three classes that are interpretable for a classification problem such as this thesis', namely the values 0, 1, and 2, representing sell, no action, and buy, respectively. Combining the threshold values and the class representations, the entire target variable was looped through to categorise each of the differenced values of the exchange rate into its respective bucket. This leads to a vector of 0's, 1's, and 2's. In order for the model to effectively learn from these using the input data, an additional transformation was applied, *one-hot encoding*, transforming each observation into a vector with 1 and 0's. If the true class label is 0, a sell label, the one-hot encoding would equivalently be the vector $[1, 0, 0]$. The choice of one-hot encoding can at first glance be a somewhat heuristic choice, but further investigation actually shows this can matter quite significantly. Instead of having the pure class labels, which have the range $[0, 2]$ consisting of natural numbers, which the model can mistakenly misinterpret as orders of magnitude, This is a problem since it can introduce bias into the model; rather, one-hot encoding ensures that each class is treated equally and cannot be mistaken for orders of magnitude. However, this is not a one-sided story of one-hot encoding only providing improvements to the model; since it creates a vector rather than a single numerical value, the dimensionality increases, which increases the complexity of the model, potentially leading to overfitting. Given the target variables binary vector representation that includes zero values in two of the elements in the vector, the memory usage of the models also increases, leading to increased computation for large datasets, although this is not a direct issue with the dataset used for the modelling in this thesis, as it is inherently small compared to other models, which are computed using up to millions of observations. For financial data, one is often faced with the issue of having a small number of observations compared to other tasks such as image processing and natural language processing.

The next step for the models is scaling the data into a specific range; the range chosen in almost all circumstances is $[0, 1]$. This is an important step since unscaled data has the disadvantage that some inputs are treated disproportionately to others based on their values and methods for computing them.

In order for the LSTM model to interpret and process the data, it is now sorted into a *tensor*. A tensor is a generalisation of a 2D matrix in 3D space; this is visualised in Appendix A.12. As the figure shows, instead of only having columns and rows indicating features and observations, respectively, a third dimension is now present. Specifically, the data is structured into the number of samples, the time steps that the LSTM processes in a sequence, and the number of features used in the model. This is essential for the model to read the data properly and ensure that it can iterate over each time step in a sequence. After structuring the data into a tensor, the data is split between a training and testing set with 80% and 20%, respectively. Importantly, there now exist four distinct datasets: X_{train}, X_{test}, y_{train}, and y_{test}. As mentioned in Section

2.4 the training sets are, as the name indicates, used to train the network and improve upon the predictions. The test sets are then used to evaluate the performance and generalization. As mentioned repeatedly, the sequencing in an LSTM network is of high importance; therefore, the data between these sets is not shuffled but rather processed linearly, so that the sequence is kept in tact.

Model Architecture

Having prepared and ensured correct presentations of the data, the model architecture can be defined for each of the models created. This acts as the shell of the model, and the task of filling it is left to the modeller. The individual components of this architecture will be explained in part, and the specific values set for these components in each model will be further elaborated in Sections 5.2, 5.3, 5.4, and 5.5.

For each of the models, the following potential parameters are defined: *nodes*, *the dropout rate*, *batch size*, and *regularization value* (either L1 or L2). The nodes specify how many nodes are included in each hidden layer; depending on the number of hidden layers, there can be several different values. The same applies for the dropout, which can be applied to each hidden layer added to the network. The batch size determines the number of samples to process before updating the parameters, which in this case are the weights and biases corresponding to each node. To prevent overfitting, both L1 and L2 regularization can be applied; in the case of the models created in this thesis, L2 regularization is applied over L1. L1 has the undesired property that some coefficients are set to zero, thus negating their impact on the model entirely. This can potentially force the model to be biased heavily towards a specific category, as the coefficients chosen could potentially be helpful in predictions for the other classes. L2, on the other hand, sets the coefficients towards zero, so their impact is diminished but not erased entirely, which creates predictions that are more evenly distributed across classes.

The number of layers determines the complexity of the model to a large degree, and increasing it can lead to cases of overfitting. Both dropout and L2 assist in diminishing the effect of this increased complexity, but even so, it can be important to vary in increasing this drastically, not only as the probability of overfitting increases, but it additionally increases the computation time.

As a final layer for each of the models, there are three nodes representing each of the classes that the model can predict. Here, the softmax function from Section 2.9.4 is applied to construct a probability vector of three elements. The element in the vector with the highest probability is selected as the prediction, which is then fed into the *categorical cross-entropy loss function*, which computes the loss for that specific prediction. During this thesis, cost functions have been introduced that were specific for their respective purposes, such as the binary cross-entropy loss in Section 2.3 for binary classification or the MSE in Section 2.1.1 for regression. Since this is a multi-class classification problem, the categorical cross-entropy loss function is the most suitable for this general case with more than two classes. Its formal definition is defined as follows:

$$-\sum_{c=1}^{N} y_c \log(p_c) \tag{5.1}$$

Equation 5.1 measures the error between two probability distributions if combined with the softmax activation function. The predicted probability p_c, which is for each class, is the value of the loss. If p_c is high, the model is rewarded for making a correct prediction, since the probability of observing this class is also high. Conversely, if the value of p_c is low, the model will be punished for making a prediction with a low probability of occurring. As always, these values are summed up to represent the cost function, whereas the aforementioned definition is per training example and is namely the loss function (Pykes, 2024).

Lastly, for the model architecture, the numerical optimisation method, Adam, from Section 2.8 is defined. In this section, it was mentioned that the starting weights and biases can play a pivotal role in determining the path towards the minima of a function; this importance is not diminished when incorporating them into a complex model, such as the LSTM. The package in Python from which the model is defined has default values that initialise the weights and biases within each hidden layer. The Keras package that was used during this thesis utilises specific distributions, such that the weights and biases can only take values within a specified range but keep the variance in tact between the training of each model. As will be shown later in the performance metrics of the models, each iteration of the same model can vary in its performance quite significantly, which these weight and bias initializations are responsible for. This effectively means that reproducing the results is an impossible task, and the model outcomes cannot be taken at face value; instead, several iterations of the same model are required to evaluate its performance (Keras, n.d.). Finally, as a continuous training process metric, accuracy is also measured, which sums the total number of predictions and calculates the percentage of correct predictions for each iteration. Importantly, each iteration is referred to in the literature as an *epoch*, which refers to an iteration in which the model has gone through the entire training set, updating the parameters each time a batch size is processed.

Hyperparameter Tuning

As mentioned repeatedly throughout the thesis, ML has many parameters that are up to the modeller to determine; luckily, this need not be the case. Instead, what is often done in practice is to define a *parameter grid*. The formulation of this grid is still handpicked by the modellers, but it allows the model to test out several different combinations and select the one with the lowest loss, subsequently producing the best model. For each of the models performed, this parameter grid is defined; depending on the type of model, this grid can vary from three to six parameters. So instead of blindly selecting values for these parameters, each possible combination of the parameter grid is searched and evaluated, which can potentially lead to many hundreds of combinations. It is possible to test many different values, but the trade-off is again reflected in the computation time. How are each of these possible combinations then evaluated? Here the principle of splitting the data reenters through the method *k-fold cross validation*, (k-fold CV). This method further splits the data, but specifically the training data is split again using the same distribution as earlier and 80% and 20%. Therefore, the actual amount of the original data set used for training becomes 64%. The splitting of the data into a further training and testing set is reflected further in the method in which k-fold CV calculates the error during the training process. From the original split, the training set is then split into five different folds, where the model is trained on each fold and the error is estimated. After having completed training for each of the five folds, the average error is computed.

The purpose of using k-fold CV is that averaging each fold error estimate is to lower the variance of the loss for both training and validation. This is essential since a high variance in the model's performance means that it is significantly dependent upon the subset of data used, which makes for unreliable results. During the parameter grid search, another regularisation technique is applied: *early stopping*. This method can drastically reduce the training time but also act as a brake so that the model does not begin to diverge to a suboptimal solution with a large generalisation gap. Specifically, this method stops the training if the validation loss has not improved for n epochs; it then restores the weights and biases that were present 10 epochs ago and uses those as the final model.

After having found the optimal parameters through the grid search and k-fold CV, the best and final model is trained and is ready to be evaluated. The first and simplest method is to plot the training and validation losses to see their evolution through the epochs used for training. This plot can be seen as a mere snapshot of the model's performance; therefore, it is good practice to test the final model through several iterations to evaluate it more accurately. This again ties back to weight and bias initializations, which are drawn from the same distributions but can have different values, leading to different results.

Performance Metric Through Several Iterations

When evaluating the performance of a model mainly four metrics are measured; the average hit rate, average portfolio value, standard deviation and the average accuracy. Consider the metrics separately can give a indicator for if the model is a well-performing model, however, when the the metrics are considered as an unity a more clear picture of the performance can be given.

The average hit rate describes the proportion of correct predictions made by the model out of the total predictions. This proportion is calculated by the amount of profitable trades divided by the amount of unprofitable trades. Note that periods where the model choose not to trade will not have any effect on the hit rate as the model takes no action. If the model chooses to make no action through all the periods the hit rate will be noted as 0. The metric indicates if the model often finds the correct direction of the market. Given the normal distribution of the EUR/USD exchange rate a hit rate of 50 percent would be the same as letting a stochastic normal distribution find the direction.

The average portfolio value is calculated by assuming a value of 100 that starting at each iteration of the model then trading the portfolio value throughout each period and finally taking the average of each models last portfolio value to see how the portfolio has traded on average over the period. The period tested is the testing set which is the last 20 percent of the data going from 2020 to 2024. As a metric the average portfolio value can give an indication if the model is trading at a profit or loss. Even though the hit rate is positive the model might still lose money as it is trading on high volatility days in the wrong direction.

The standard deviation is the average standard deviation of the average portfolio value. It is essential to know as it tells how much of a risk a given trading model has. It will also be one of the key factors in determining if the model is a good trading strategy as a model with slight profit might not be desirable if the standard deviation is too high.

The accuracy is how often the model chooses the correct labeled category. This metric is important as it indicates how good of a fit the model has. Even tough the model accuracy is low the model could still be well formed model if the model is able to predict shocks scoring a lower accuracy but can create high value for the trader as shocks are periods of large losses and gains. When looking at the accuracy it is important to have in mind that if a normal distribution based on the threshold were too choose every signal the accuracy would be between 33-34 percent. Thereby, if a model has this accuracy it cannot be certain that it has learned any of the underlying patterns unless it performs well on the other performance metrics.

By analysing the four metrics together it should give an adequate look at if the model is performing well. Some metrics will have more impact than other, however, this will be determined on an ad-hoc basis as it is difficult to determined the values of the metrics from a rule-based approach but most be done from a principal-based approach.

5.2 Single-Layer LSTM - Model 1

In the following section, the first model will be constructed for a single-layer LSTM model with all the selected variables mentioned in the data description. First, the construction and architecture of the LSTM model will be described including considerations of performance and issues that might be present in the model. Secondly, the models results will be presented and interpreted. At last errors and improvement will be discussed so that the next iteration in the second model can learn from the mistakes of the first.

The architecture of the for first model is a simple single-layer LSTM model that uses dropout as the only regularization tool. The architecture is visually illustrated in figure 5.1.

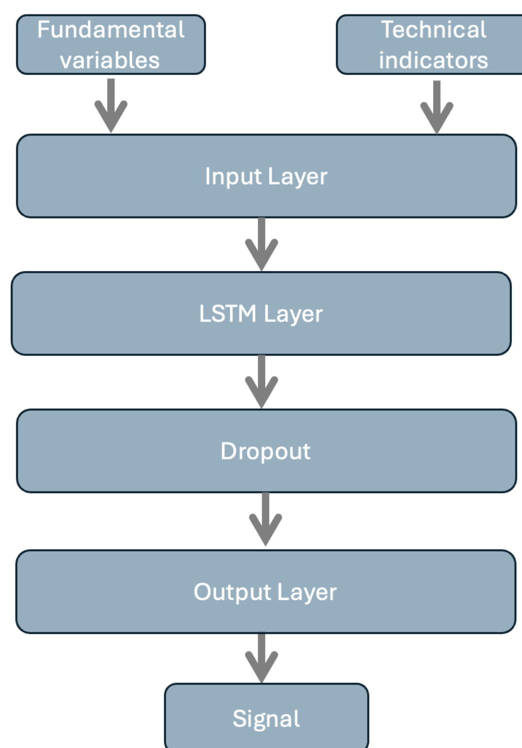

FIGURE 5.1: The architecture of model 1

One of the considerations that went into the model was that there was a large number of inputs with higher complexity. The inputs are all derived from macroeconomic or financial variables that are known to have high amount of noise and volatility. The question is how the model will react to this kind of data input. The LSTM model could have difficulties in lowering the generalisation error and gap as the noise makes the gradient converge against a sub optimal result. Furthermore, In the first model there is no regularization term like ridge regularization that actively reduces the generalization gap which could lead to a divergence in the generalization gap. The model will therefore be a initial test of how a standard LSTM processes the data.

Here, 36 models were run to test for the lowest validation error. The models were run for the following hyperparameters; LSTM nodes $\{32, 64, 128\}$, dropout rates {0.1, 0.2, 0.3, 0.4}, batch size $\{32, 64, 128\}$. The chosen model with the lowest validation

error was with the following hyperparameters: LSTM nodes = 32, dropout rate = 0.1, batch size = 128.

5.2.1 The results of model 1

The performance metric results of the model are shown in table 5.1. The result show that the model performs close to that of a random walk. Note that the standard deviation is 9.99 which indicates that the model has fluctuations between 89.44 - 109.42. The accuracy of the model was 34% which indicates that the model is very close to a random walk as the model has 33% chance of hitting the correct category which a slight bias towards buy and sell - thereby making 34% close to a stochastic normal distribution.

Performance Metric	Value
Average Hit Rate	0.4994
Average Final Portfolio Value	99.34
Standard Deviation of Final Values	9.99
Average Accuracy	0.34
Average No. of Trades	568

TABLE 5.1: Summary of model 1's performance metrics

Overall, the performance of the model did not satisfy any of the performance metrics and came close to having the same performance as a random walk which would indicate that the model has not understood the trend patterns of the dataset. Furthermore, when looking closer at the development of the model loss, it can be observed that the model overfits the data as it has an increasing validation loss and generalisation gap. A single iteration out of several tested is shown in figure 5.2 for model 1 showing clear overfitting.

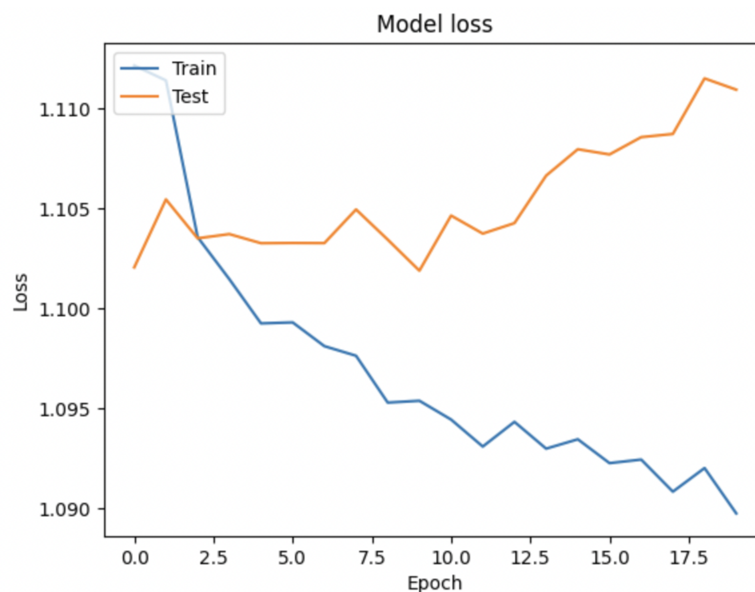

FIGURE 5.2: The model loss of model 1

For the next iteration of the model, a regularization term should be added to the model as not to make the model overfit the data as heavily as it has done in model 1. As model 1 shows problems dealing with the volatility of the data, it will be

interesting to find out if regularization will be able to let the model find long-term and short-term dependencies of the input variables for the the EUR/USD exchange rate.

5.3 Single-Layer LSTM With Regularization - Model 2

In the following section the second model will be constructed. Firstly, the architecture for the second model will be described and the initial expectations for the model will be explain. Afterwards the results of the model will be reviewed and at last the changes for the next iteration will be discussed.

The architecture of the for second model is a single-layer LSTM model that has a ridge regularization term to generalise the results of the model. The architecture is the same as in figure 5.1, however, with the difference that each weight is affected by the ridge regularization to reduce generalization error. The ridge regularization term can, however, also have the negative effect of increasing the training error - making the model worse at fitting the data.

For model 2 one of the interesting point of focus will be if the model is able to retain its accuracy while also lowering its generalization error thereby not overfitting. Furthermore, if the generalization gap falls will the model be able to forecast the market more enough accurately to make a positive hit rate and portfolio value.

81 models were run to test for the lowest validation error. The models were run for the following hyperparameters; LSTM nodes $\{32, 64, 128\}$, dropout rate $\{0.2, 0.3, 0.4\}$, batch size $\{32, 64, 128\}$. The chosen model with the lowest validation error was for the following hyperparameters: LSTM nodes = 128, dropout rate = 0.4, batch size = 128.

5.3.1 The results of model 2

The performance metric results of the model are shown in table 5.3. The results shows that the models performance in terms of profit and hit rate has not improved with a hit rate close to the same as model 1 and an average portfolio value below at 96.50 percent. One notable observation is that the model has a significant better standard deviation which is preferable to model 1. Indicating that the model has lowered its risk/reward in trading. The accuracy of the portfolio was also raised to 35% which indicates that the model has improved from a random walk, however, not by a significant amount.

Metric	**Value**
Average Hit Rate	0.4953
Average Final Portfolio Value	96.50
Standard Deviation of Final Values	6.19
Average Accuracy	0.35
Average No. of Trades	638

TABLE 5.2: Summary of model 2's performance metrics

Overall the model has improved in the second iteration of the model as there was a lower standard deviation and higher accuracy. However, the trading performance of the model is still not satisfactory - as the model trades at a loss and below 50% hit

rate. When looking at the model loss for several iterations it is clear that the regularization has been effective in preventing overfitting. A low generalisation cap makes the result of the model more consistent and is a good indication that the model has become model capable of generalising. The model loss is visually illustrated in figure 5.3.

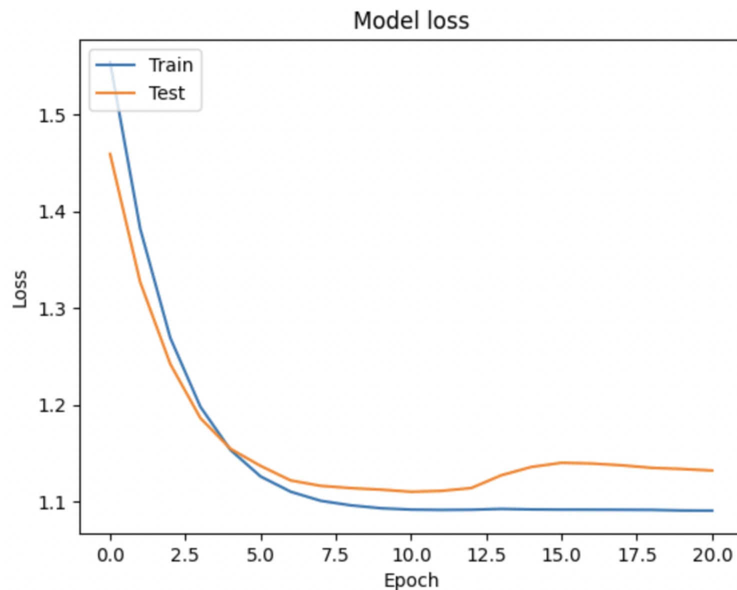

FIGURE 5.3: The model loss of model 2

For the next model iteration a focus should be on further boosting the models performance. The LSTM model does not seem to find a trend pattern for the variables to fit the output variable, this might be due to the simple and lower complexity of the first and second model. Therefore the model might perform better at a higher complexity which could be done by adding another LSTM layer.

5.4 Two-layer LSTM - Model 3

In the following section the third model will be constructed. Firstly, the architecture for the third model will be described and the initial expectations for the model will be explain. Afterwards the results of the third model will be reviewed. at last the changes for the next and final iteration will be discussed.

The architecture of the for third model is a two-layer LSTM model with a ridge regularization term. The architecture is more complex structure and visually illustrated in figure 5.4.

For model 3 the expectation is that the model will be able to unveil a better trend pattern between the variables and the EUR/USD exchange rates. The more complex structure of the model should able to capture more complex pattern in the dataset. However, the question is if the model will be effective enough as a trading strategy as the model should have a significant profit and low standard deviation in order to be able to be traded on.

144 models were run to test for the lowest validation error. The models were run for the following hyperparameters; first LSTM layer nodes $\{64, 128\}$, second LSTM layer nodes $\{64, 128\}$, dropout rate1 $\{0.2, 0.3, 0.4\}$, dropout rate1 $\{0.2, 0.3, 0.4\}$, dropout

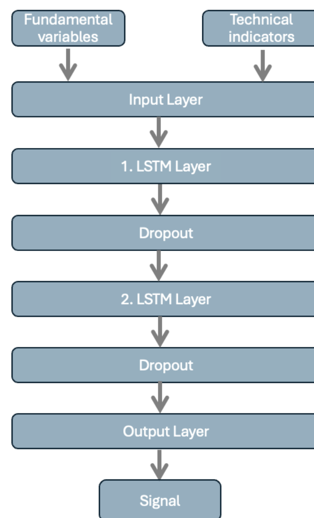

FIGURE 5.4: The architecture of model 3

rate2 $\{0.2, 0.3, 0.4\}$, batch size $\{64, 128\}$ and the delta for the l2 term $\{0.1, 0.01\}$. The hyperparameters were in this iteration lowered in order for the computation time not to rise exponentially. The chosen model with the lowest validation error was for the following hyperparameters: first LSTM layer nodes = 128, second LSTM layer nodes = 64, second LSTM layer nodes = 128, dropout rate1 = 0.4, dropout rate2 = 0.3, batch size = 128.

5.4.1 The results of model 3

The performance metric results of the model are shown in table 5.3. The results shows that the models performance has improved significantly as the model has generated a positive profit of 100.11% and hit rate above 50% with an even lower standard deviation of 4.66 compared to model 2. The trade accuracy has also improved which indicates the the model has improved in spotting trend patterns in the dataset. As the previous model were close to having the same performanve as a random walk to confirm the new results an additional 100 iterations of model 3 were run. The results showed that the second iteration of model 3 performed similarly with a profit of 100.59 % and a hit rate of 50.2% - confirming that the model indeed has improved.

Metric	Value
Average Hit Rate	0.5016
Average Final Portfolio Value	100.11
Standard Deviation of Final Values	4.66
Average Accuracy	0.36
Average No. of Trades	568

TABLE 5.3: Summary of model 3's performance metrics

Overall implementing the second layer LSTM proved to be an great improvement for the model. The generalisation error of the model is very similar to model 2 and is shown in figure 5.5. When putting the model in relation to a trading strategy the performance is still not satisfactory as the model only produces a slight profit and

hit rate above 50%. Additionally, taking the standard deviation into consideration the risk/reward of the model is not satisfactory. The model would therefore not be considered good for trading or forecasting.

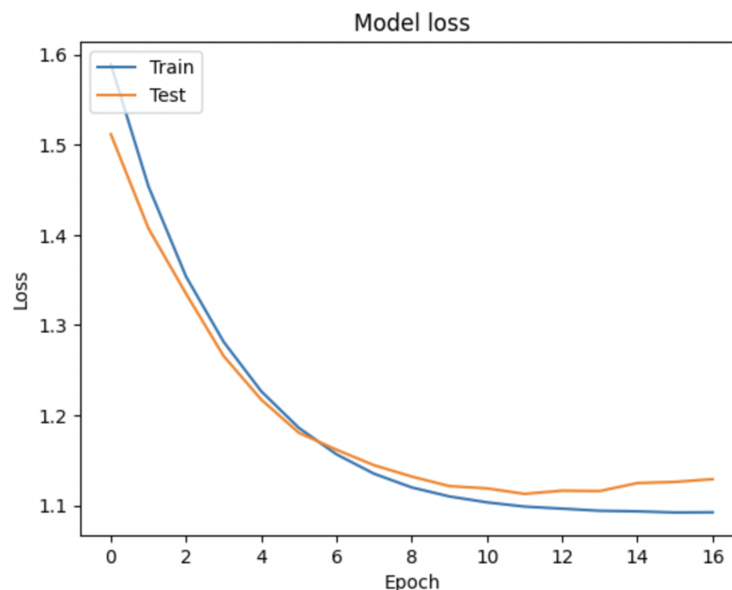

FIGURE 5.5: The model loss of model 3

An additional model was run for a three-layer model with selectively chosen hyperparameters, however, the performance of the three-layer model drastically decreased the performance metrics and it is concluded that raising the complexity of the model would not benefit the model's performance, therefore it was scrapped. Therefore for the next iteration another problem with the model is addressed. One of the original concerns for the model was that the volatility and noise of the dataset might cause the model to not be able to distinguish between the noise and correct signals in the dataset. To correct for this, the next iteration of the model will therefore focus on lowering the noise in the dataset by removing highly noisy variable that has low or medium signalling for the pattern of the EUR/USD exchange rate.

5.5 LSTM with restricted data - Model 4

The following section will differ from the other modelling sections as this model will include the final model with all iterations in it. Firstly, the restriction of the data set will be constructed and discussed. Afterwards, the first two iterations for model 1 and 2 will be made for the restricted dataset to evaluate the general performance of the dataset in the LSTM model. When the performance has been evaluated the final model will be constructed and results discussed. At last the section will be concluded and a reflective look will be taken for the modelling process.

For this model the dataset is restricted to only include variables that has high signalling value as well as a low noise. First of all the technical variables are included as all variables are a smoothed indicator that describe important short-term movements in the trend pattern. When choosing the macroeconomics variables a low dimensionality set of variables is preferred. The most central and key macroeconomic variables are determined to be the inflation, unemployment and interest rate for both the US and EU. These variable are monthly variables that all directly highly effect the

EUR/USD exchange rate. Given that the variables are monthly updated the shocks doesn't have noise. A final variable is chosen as a indicator for the volatility in the financial markets; the VIX index for the US and EU. In conclusion the variables for the final model are: SMA(10), SMA(20), MACD(12,26), RSI(10), inflation for USA and EU, unemploymnent for USA and EU, interest rate for USA and EU, and finally the VIX indices for S&P500 and EURO50. Now that the dataset has been set up for the new models the modelling of the same iterations as model 1, 2 and 3 can begin.

The results of the first model 4a, where the new dataset is applied on a simple single-layer LSTM, is shown in table 5.4. Surprisingly enough even though the model overfits, the profit performance metrics perform well. The profit of the model nets an average portfolio value of 102.75% and a hit rate of 50.3%. However, the model still has a high standard deviation of 10.48 and even lower accuracy score at 31.4%. As the average profitability is the highest for now the model is run for another 100 iterations to test if the profitability was an abnormal distribution. The new test showed that the model still attained a positive profit of 100.54 and hit rate of 50.04% with a standard deviation of 10.64 and accuracy of 0.31. in conclusion, it seems that the models profitability has improved, however, the overall fit of the model is the same as the first model and the model would therefore again not be suitable for a trading strategy.

Metric	**Value**
Average Hit Rate	0.5033
Average Final Portfolio Value	102.75
Standard Deviation of Final Values	10.48
Average Accuracy	0.31
Average No. of Trades	920

TABLE 5.4: Summary of model 4a's performance metrics

The results of the second model 4b, where the new dataset is applied on a single-layer LSTM with ridge regularization, is shown in table 5.5. The results shows to be quite improved with an average portfolio profit of 100.72% and hit rate of 50.33%. The standard deviation remains low at 5.02 and the accuracy is the highest achieved yet at 37%. It would seem that simplifying the data inputs makes a single-layer LSTM more sufficient in analysing the trend patterns for the dataset. This could be considered in a trading strategy as the profit is significant, however, the standard deviation is at the same time, relatively to the expected profit high, making the strategy only suitable for risk-seeking traders. It is notable that the trading strategy only has an average trade of 499 trades which makes it less active than the other models.

Metric	**Value**
Average Hit Rate	0.5037
Average Final Portfolio Value	100.72
Standard Deviation of Final Values	5.02
Average Accuracy	0.37
Average No. of Trades	499

TABLE 5.5: Summary of model 4b's performance metrics

For the final model 4c, the new data is applied on a two-layer LSTM model with regularization. This was the model that previously performed best, however, by

lowering the data complexity this model could lead to the modeling being too complex for the new dataset has happened when a three-layer LSTM was tested on the old dataset. The results of the final model are displayed in table 5.6. The results show that the structure of the model is likely too complex as the average accuracy is 30.46%. The rest of the performance metrics are additionally underwhelming as the average profit of the portfolio is 89.78% with a hit rate of 49.37%. Even though the standard deviation is low the model is simply not predicting any patterns in the EUR/USD exchange rate.

Metric	**Value**
Average Hit Rate	0.4937
Average Final Portfolio Value	89.78
Standard Deviation of Final Values	4.61
Average Accuracy	0.30
Average No. of Trades	809

TABLE 5.6: Summary of model 4c's performance metrics

In conclusion, model 4b ended up being the best model for a trading strategy as it has the best profit at the lowest standard deviation. Even though model 4a yielded a better profit, the overfitting of the model and high standard deviation made it more unstable and hence unreliable. For model 3, there was a lower standard deviation than model 4b, however, the profit of the model was simply too low.

Given the results of the models, it can be concluded that modelling a FOREX trading strategy can be a very difficult task, as it is one of the most complex markets to model. If the modelling should be optimised looking into more advanced methods is needed such as methods for dealing with noise, like principal component analysis (PCA). Furthermore, obtaining more advanced and better data quality is also a necessity - as using open-source data is harder to create an edge as other traders also has access to it. More advanced data could either be a new collected data variable or unique components that are constructed from already available data.

Chapter 6

The role of AI in future financial markets and asset pricing

6.1 The evolution of prediction in financial market

Modelling, forecasting and understanding economic and financial data has for a very long time been of interest among economists and governments. Theories, models and methods have continuously been developed just to be proven wrong through times of booms and busts, and so, the evolution of predicting financial market behaviours has been changed dramatically (Ebrahimi, 2023).

Going all the way back to the 17th century, market analysis was already in focus and it was mainly driven by qualitative information and rumors. Take as an example the very famously known crisis; *Tulip Mania* in 1637 in Netherlands that was caused primarily by emotional behaviour in the pricing market (Hayes, 2022). Before the crisis hit, the value of tulips reached as high as six average annual salaries. This led to an extreme urge to understand and model pricing of such markets. Moving forward into the early 20th century, Benjamin Graham and David Dodd built the foundation for fundamental analysis through their book, *Security Analysis*, which focused primarily on pricing assets and firms based on their *intrinsic* value and nothing else. While this was a milestone in forecasting future valuations, it was lacking explanatory power in that the markets were probably not as efficient as they assumed and behavioural aspects of market participants should also be included. Now, taking into account all different factors that potentially explain the price movements of assets, it is likely impossible to build a theoretical framework or model, that explains it all.

This is where data comes into the picture. What if one can forecast future tendencies of an asset purely based on historical numbers? Quantitative analysis in econometrics has profoundly impacted the prediction of all types of financial data, including exchange rates. The popularity of econometric modelling likely stems from its objective nature; it is unbiased in that it belongs to no specific economic ideology and it simply tells a neutral story by purely combining numbers in certain ways. With the development of computers and increasing availability of data, the traditional econometric models have been used by traders, hedge funds, businesses and policymakers for decades - and it still proves as a remarkable tool in modelling some financial data.

The world is now at a point where the competition and comparison between traditional econometric methods of forecasting financial assets with machine learning algorithms is at a peaking point. Some prefer one over the other while others are

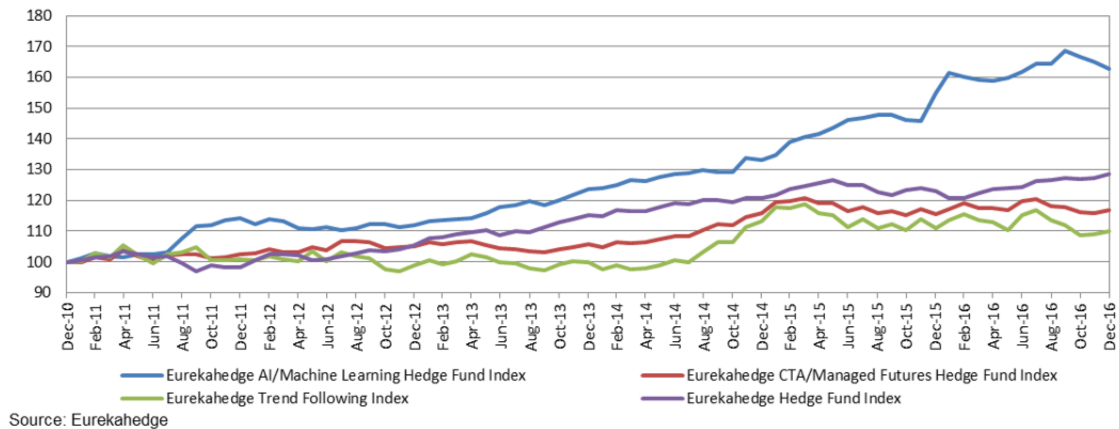

FIGURE 6.1: AI-driven hedge funds versus traditional hedge funds. 2010-2016 return index. Data and illustration by Eurekahedge Investment research firm.

looking to merge the two into some hybrid model. (Pérez-Pons et al., 2021) released a research article summarizing 52 papers comparing traditional econometric models against machine learning models in predicting different financial or economic variables. They found that a huge majority of papers found ML predictions to outperform econometric ones. The predictions included price forecasts on stocks, future demand forecasts, insurance performance etc. Only as little as two papers found econometric models to outperform ML, which was specifically a regression model and a support vector regression respectively. It should also be noted, that quite a lot of papers found that combining ML with econometric models yielded the optimal results in forecasting tasks, which is quite interesting. These hybrid models might be showing some promising potential for future modelling and forecasting.

Furthermore, the investment research firm, Eurekahedge, performed an in-depth analysis of the performance of four different hedge fund types during 2010-2016; some were AI driven while the three other were traditional types of hedge funds. The resulting returns are displayed in figure 6.1. As can be seen in the graph, the AI-driven hedge funds significantly outperform all others with an annualized return of 8.44% in comparison to 2.62%, 1.62% and 4.27%. Additionally, they found that the risk-adjusted returns (sharpe ratio) was better as well. These results indicate some strong potential that are convincing. In fact, the machine learning area is so convincing that the six biggest Wall Street banks (Goldman Sachs, JPMorgan Chase, Bank of America Merrill Lynch, Morgan Stanley, Citigroup, and Wells Fargo) have all participated heavily in the funding of the famous machine learning platform, Kensho, which provides data analytics based on AI technology.

Now, it is important to also highlight the fact that some studies still find evidence for the opposing to be true; namely that econometric models can still outperform ML in some tasks. As a great example, (Jang and Lee, 2019) thoroughly analysed the performance of calibrating, predicting and domain adaptation for the S&P 100 put option pricing through econometric jump models against several ML branches. First, the price calibration performance by an econometric model using only the previous day as input was similar to that of ML models using the past seven days as inputs. Second, the econometric jump model predicted prices significantly better

than all ML models, which only showed to be more evident as the period to prediction was increased. Lastly, they tested domain adaptation performances by letting the put options randomly switch between American and European versions of the same option, where only the econometric model could recognize and adapt to this mechanism. The ML models could not separate the two versions from each other, which was a surprising result given the large compliments ML has received in terms of adapting to dynamic structures.

As an example on some of the hybrid models combining ML and econometric models, (Saâdaoui and Rabbouch, 2019), conducts a comparative analysis between multiple different model types in predicting short-term electricity prices. Now, it is important to appreciate the extreme volatility of electricity prices, which arguably is higher than most other markets - at least much more than the financial markets. Their analysis finds that by combining a very traditional econometric model; the auto-regressive integrated moving average (ARIMA) model with a quite simple feedforward network, they are able to catch patterns, non-linearities and volatility that outperform four strong benchmark models, and so their hybrid model predicts day-ahead power prices very accurately. The main step for their success in prediction is wavelet transformation, which allows them to separate components of the time series, and then dedicate the different models for the components they are each best at estimating on - thereby pulling strengths from both worlds. Their work motivates for a very interesting potential combination of two pillars of statistics. If they are able to build a model that efficiently predicts such volatile and irregular time series, it must have some promising potential in most other types of time series forecasting challenges.

6.2 The computational challenges of an AI future

AI and machine learning has for a long time been a thing in the world of research and modelling, however, the computational power demanded by proper ML models has simply never been fully satisfied and this could be a serious bottleneck in the future development potential. All the way back to 1960 when Frank Rosenblatt, known as the father of neural networks, constructed a three-layered NN, he quickly came to the conclusion that *"as the number of connections in the network increases, the burden on a conventional digital computer soon becomes excessive"*, and in 1969, Minsky and Papert also developed a more efficient deep neural network, but their framework was quickly abandoned, as the computational needs were simply not met. Ever since the beginning, it seems that the development of machine learning models has been haltered by the computational capacity available. Today, computational power has become extremely strong and yet still struggles to keep up with the modern and recent suggestions of ML models. It has become popular to think that ML models can now be fully utilized given our rapidly increasing computational capacity, however, an MIT initiative, specifically dives into this exact issue, and surprisingly finds that the gap between computational capacity and demand needed for the full ML potential has increased even more during the last decades (Thompson et al., 2020). This is not because computational performances are not increasing, but more because the complexity in potential DL models is increasing much faster. It can further indicate something quite unsustainable; if computation of advancing models gets more and more expensive, then the institutions with the most money will dominate on the AI-front while individuals and small businesses will struggle to keep up.

To demonstrate how computational capacity, time and available wealth is paramount in how deep learning performs, lets compare the models in this paper with another, way more extreme case; ChatGPT-4 by OpenAI. The *GPT* stands for *Generative Pre-trained Transformer*, and so ChatGPT is build on a form of deep learning model, namely transformer, which is often compared to and competing with the LSTM - also in the context of currency exchange rate predictions. In building the latest ChatGPT-4, the amount of parameters to be estimated was around *1.8 trillion* parameters across 120 layers. The models used in this analysis never came near half a million parameters, and the most advanced model included just three layers. Furthermore, the batch sizes that OpenAI used during training reached a shocking 16 million compared to just 128 here. While this study spent a small amount of money renting a single remote GPU (which was a huge upgrade compared to what was else at hand), OpenAI had to utilize 25,000 A100 GPUs, which are among the most powerful GPUs in the world. The cost of training their is unknown but must be running up millions of dollars. While the most complex model in this study took between 2-3 hours to train, it took 90-100 days for 25,000 GPUs to train ChatGPT-4.

Now, these comparisons are extreme, and one should appreciate that ChatGPT-4 is among the most advanced field of AI the world has ever seen, but it serves to highlight the significant range of difference in performance that AI can deliver. And it surely also shows that the computational effort and amount of money invested one is willing to sacrifice is paramount in how deep learning performs. The above presented facts on the training process of ChatGPT-4 also helps explain why one cannot expect to simply sit down and build a model that perfectly predicts EUR/USD "just because its artificial intelligence". No, much more is needed and it takes a lot of time, expenses and effort to produce results that are both consistent and outperforming. The question is then to what degree future developments of AI and ML will be accessible to the public.

Chapter 7

Conclusion

This study sets out to explore the application of LSTM networks in predicting the EUR/USD to determine if utilising a machine learning network can generate a profitable and usable trading strategy.

To support the problem formulation, a theoretical framework was constructed in order to help understand the mechanics of the LSTM model. Firstly, the capacity of a machine learning showed that it is necessary to fit the model to the correct complexity, else the model will under- or overfit. Then looking at the regularization and optimization of a neutral network, the study concluded that it is necessary to implement a regularization for lowering the generalisation error and an optimizer to find a numerical optimal solution. The regularization method used in the project is the well-known Ridge regularization and Dropout and the optimization method is ADAM. In the construction of a neutral network, choosing an LSTM model has had several advances that differ from other models. 1; It can distinguish between short and long term effects, 2; it can handle non-linear data and 3; it has previously proven results in the financial markets.

The literature review gave an in-depth insight into the variable needed in constructing a LSTM model to predict exchange rates. The review was split into a fundamental and technical section where the fundamental review found macroeconomic variables that directly explained the exchange rate the EUR/USD and the technical review found that several indicators could be used to determine and examine the behavioural patterns in the exchange rate. At last, a look was taken at a hybrid model example from the literature which incorporated technical and fundamental indicators in order to create predictive FOREX model of the EUR/USD exchange rate. The review model gained a profit of 53.05 percent with a standard deviation of 7.42. Afterwards, the analysis and set up of the model began by a data description of the chosen variables. The variables were chosen collectively from the theoretical framework for exchange rate theories and the literature review.

In the development of the framework for each iteration of the models presented in the analysis, it was shown that careful preparation of both the target and input variable was paramount, as it can have significant impacts on the training outcomes. Especially the preparation of the target variable showed to be very sensitive to class distributions, which called for a more even distribution between each class in order for the model to not be biased towards a specific class, and thus leading to inconsistent results in the back-testing evaluation. When the model architecture were defined, the weight and bias initializations showed that each iteration of the same model specifications could lead to prediction and back-testing outcomes which varied in performance. This tied together with the numerical optimization theory in

which it was shown that the initialization values for the weights and biases proved to have a significant effect for the path towards the minima of the functions. Therefore it was concluded that each model could not be taken at face value, rather it was essential to evaluate it over several iterations to determine its validity.

The modelling generally improved with the amount of iterations of the models. Looking at model 1, a clear overfitting could be seen. When this was corrected for in model 2 by implementing regularization the model improved, however, it was not accurately able to discern between the trend patterns in the dataset. A more complex model was introduced with a two-layer LSTM, which seemed to improve at disguising between the signals which the data inputs provided, however, ultimately did not do so satisfyingly as the profit of the model was close to the initial portfolio value of 100, hence it did not generate significant results. The fourth and final model was constructed where the issue of white noise in the data was addressed through reducing the dimensionality of the dataset. The model was run similarly to all of the previous three models and resulted in a overall better modelling when taking white noise into account. The first iteration for model 4a was very similar to the first as it overfit the data, however, gave a significantly higher profit. The second model 4b had the best fit and proved to be a trading model suitable for risk-seeking traders. However, the model results did not yield well enough profit for a practical use as it gained a 100.72 percent portfolio value with a standard deviation of 5.02. The third iteration of model 4c ended up being an overly complex model as the complexity of the two-layer LSTM did not fit the more simple structure of the new dataset. Therefore model 4b ended up being the best model for a trading strategy. It was concluded that to build a better and practically sufficient model for trading, a more in-depth look needs to be taken into the advanced methods of noise reduction and more advanced datasets need to be obtained. Alternative to LSTM is the well-known Transformer model, which might be able to perform.

At last, a look was taken at the methodology of modelling from historical data. Throughout history many models and theories has been set up and hypothesised, however, the one remaining constant is that shocks and structural changes occurs and changes the dynamic of the economy. The ability to adapt and change distinguishes the best model from the worst. If frameworks can adapt to changes by incorporating data in quantitative analysis, the most accurate prediction will become unbiased of the structural period that a economy is in. Previously, econometrics has been the main tool for economists in analysing the real world, however, with the emerge of machine learning it could replace the role of econometrics in analysing shocks and structural changes. It seems that when looking at the literature, machine learning has a overweight in outperforming econometrics, however, at the same time many are looking towards a hybrid incorporation of both econometrics and machine learning in one model. It is at the same time important to highlight that econometric models at some tasks outperform machine learning models.

Finally, the computational challenges of machine learning can be argued to be an unsustainable practice as the computational capacity is exponentially rising. The amount of hardware and cost required to run comprehensive and large scale model is very high. This indicates that only large corporations are able to perform the calculations and retrieve the required data in order to make the artificial intelligence models which can predict the financial markets. A final question is asked whether the future development of AI and ML will become accessible to the public or not.

This is currently left as an open question ready to be answered within the coming years.

Appendix A

Appendix

A.1 Appendix A

A.1.1 Least squares and normal equations

$$\nabla_{\mathbf{w}} MSE_{train} = 0 \tag{A.1}$$

Plugging the representation of MSE_{train} from equation 2.3 into equation A.1:

$$\nabla_{\mathbf{w}} \frac{1}{m} \mid\mid \hat{\mathbf{y}}^{train} - \mathbf{y}^{(train)} \mid\mid_2^2 = 0 \tag{A.2}$$

Earlier in equation 2.1, $\hat{y}$ was defined, this representation will be used for $\hat{\mathbf{y}}^{(train)}$ and yields:

$$\nabla_{\mathbf{w}} \frac{1}{m} \mid\mid \mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)} \mid\mid_2^2 = 0 \tag{A.3}$$

Realizing that linear algebra notation can substitute having the entire term squared:

$$\nabla_{\mathbf{w}} (\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)})^T - (\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)}) = 0 \tag{A.4}$$

Remembering to transpose the expression on the left to abide dimensions when doing matrix multiplication. Now multiplying the two expressions together yields:

$$\nabla_{\mathbf{w}}((\mathbf{X}^{(train)}\mathbf{w})^T\mathbf{X}^{(train)}\mathbf{w} - \underbrace{(\mathbf{X}^{(train)}\mathbf{w})^T\mathbf{y}^{(train)} - \mathbf{y}^{(train)T}\mathbf{X}^{(train)}\mathbf{w}}_{\text{Equality}} + \mathbf{y}^{(train)T}\mathbf{y}) \tag{A.5}$$

$$\Rightarrow \nabla_{\mathbf{w}}((\mathbf{X}^{(train)}\mathbf{w})^T\mathbf{X}^{(train)}\mathbf{w} - 2(\mathbf{X}^{(train)}\mathbf{w})^T\mathbf{y}^{(train)} + \mathbf{y}^{(train)T}\mathbf{y}) \tag{A.6}$$

After having simplified the expression, the gradient of the expression wrt. $\mathbf{w}$ is computed:

$$2\mathbf{X}^{(train)T}\mathbf{X}^{(train)}\mathbf{w} - 2\mathbf{X}^{(train)T}\mathbf{y}^{(train)} = 0 \tag{A.7}$$

Rearranging and rewriting:

$$\mathbf{X}^{(train)T}\mathbf{X}^{(train)}\mathbf{w} = \mathbf{X}^{(train)T}\mathbf{y}^{(train)} \tag{A.8}$$

Isolating for $\mathbf{w}$ by taking the inverse of $\mathbf{X}^{(train)T}\mathbf{X}^{(train)}$:

$$\mathbf{w} = (\mathbf{X}^{(train)T}\mathbf{X}^{(train)})^{-1}\mathbf{X}^{(train)T}\mathbf{y}^{train} \tag{A.9}$$

The system of equations whose solution is given by equation A.9 is known as the *normal equations*.

A.1.2 Derivation of Decision Boundary

$$g(\mathbf{X}^{(train)}) = 1 - g(\mathbf{X}^{(train)}) \tag{A.10}$$

The solution to this equation are the points, for which the two classes are equally probable, these points are placed exactly on top of the decision boundary. For the binary logistic regression this yields:

$$\frac{\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})} = 1 - \frac{\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})} \tag{A.11}$$

Rearranging and simplifying:

$$\frac{\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})} = \frac{1}{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})} \tag{A.12}$$

Multiplying both sides by $\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})$:

$$\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)}) = \frac{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})}{1+\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})} \tag{A.13}$$

Simplifying and taking the logarithm of both sides:

$$ln(\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)})) = ln(1) \tag{A.14}$$

Using the property $\ln(exp(a) = a$ and knowing that $ln(1) = 0$:

$$\exp(\boldsymbol{\theta}^T\mathbf{X}^{(train)}) = 0 \tag{A.15}$$

A.2 Appendix B

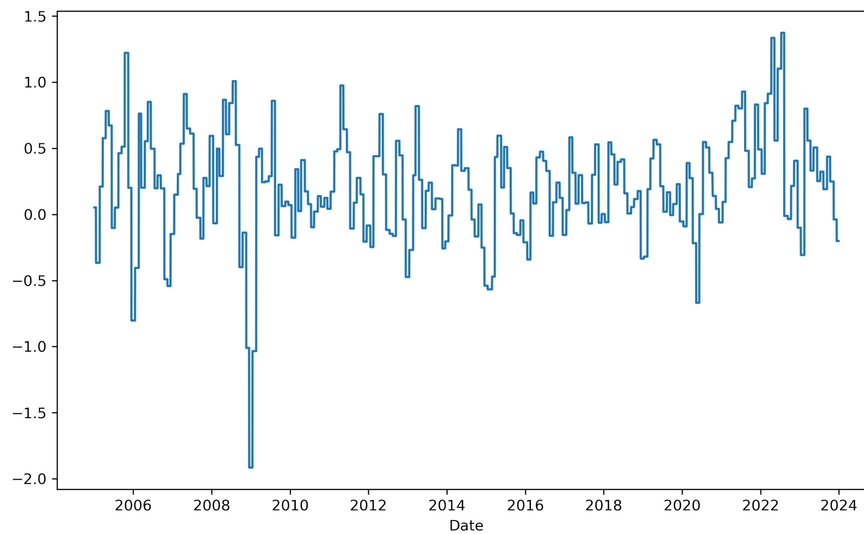

FIGURE A.1: **US CPI**. Month-to-month inflation rates. Data: FED St. Louis. Own illustration

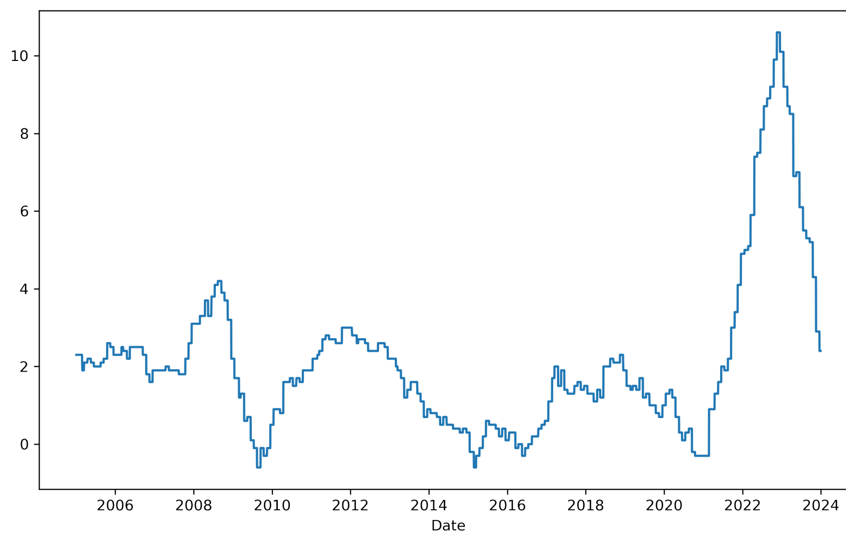

FIGURE A.2: **EU CPI**. Yearly inflation rates, monthly frequency. Data: ECB. Own illustration

A.3 Appendix C

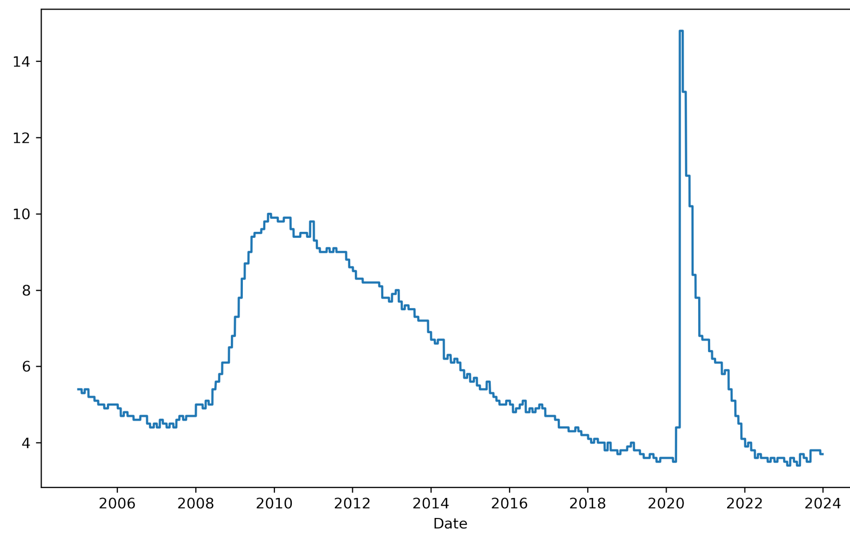

FIGURE A.3: **US Unemployment rates**. Data: FED St. Louis. Own illustration

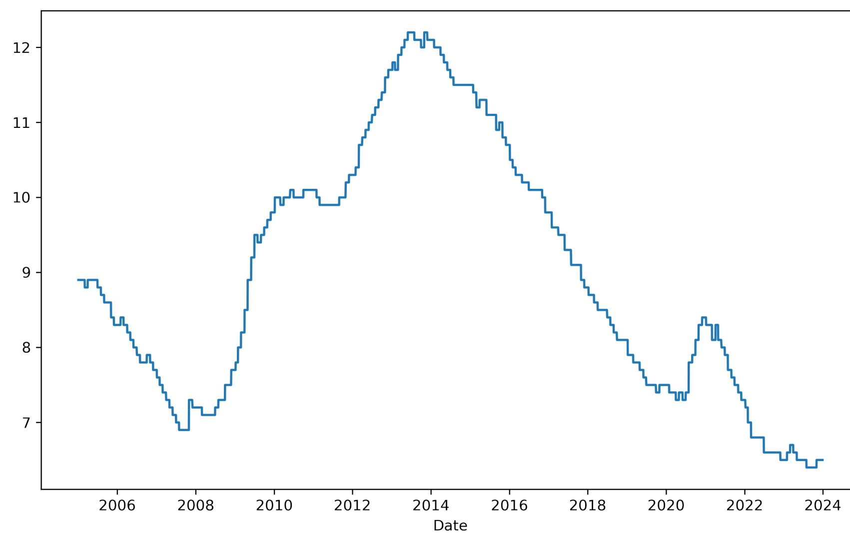

FIGURE A.4: **EU unemployment rates**. Data: ECB. Own illustration

A.4 Appendix D

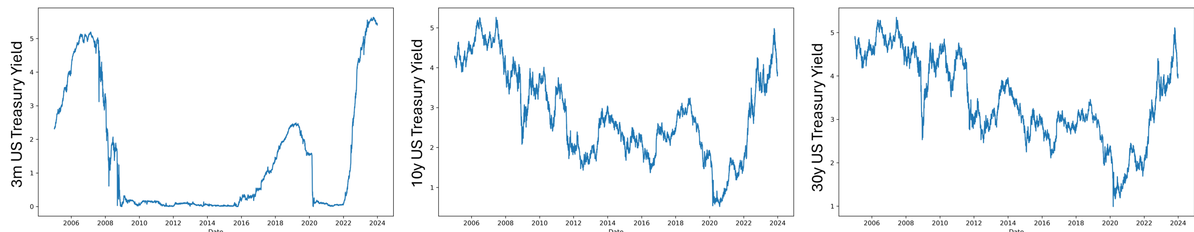

FIGURE A.5: **US Treasury Yields**. From left to right: 3m, 10y and 30y yields. 2005-2024. Data: Alpha Vantage API. Own illustrations

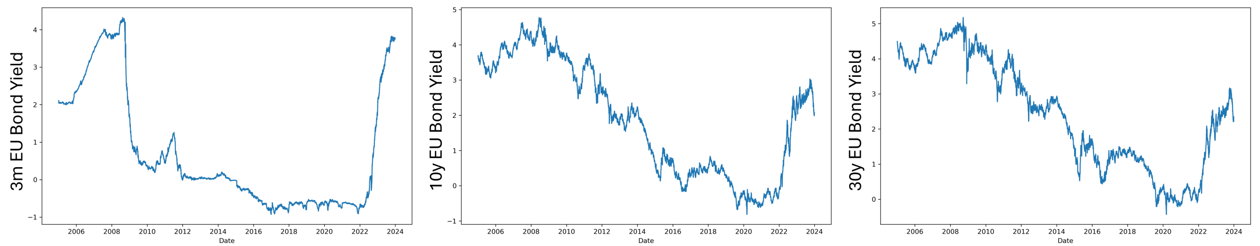

FIGURE A.6: **Euro Area AAA Bond Yields**. From left to right: 3m, 10y and 30y yields. 2005-2024. Data: ECB. Own illustrations

A.5 Appendix E

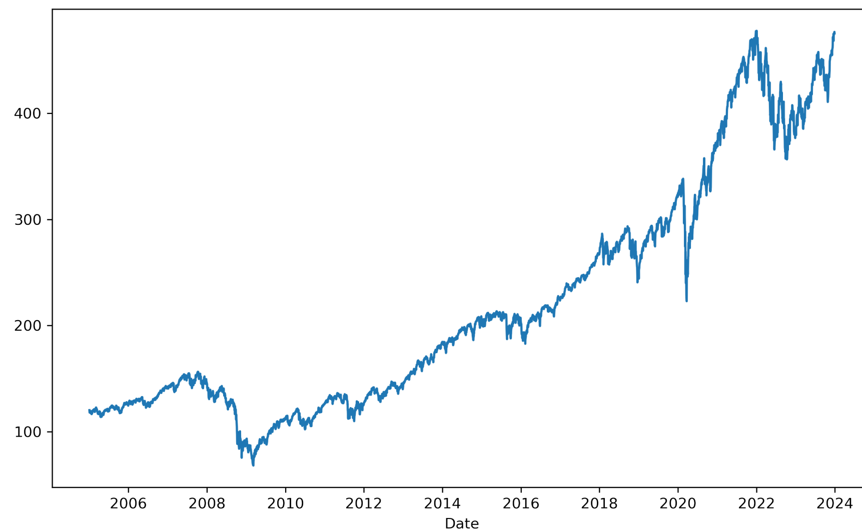

FIGURE A.7: **S&P500 Index**. 2005-2024. Data: FED St. Louis. Own illustrations

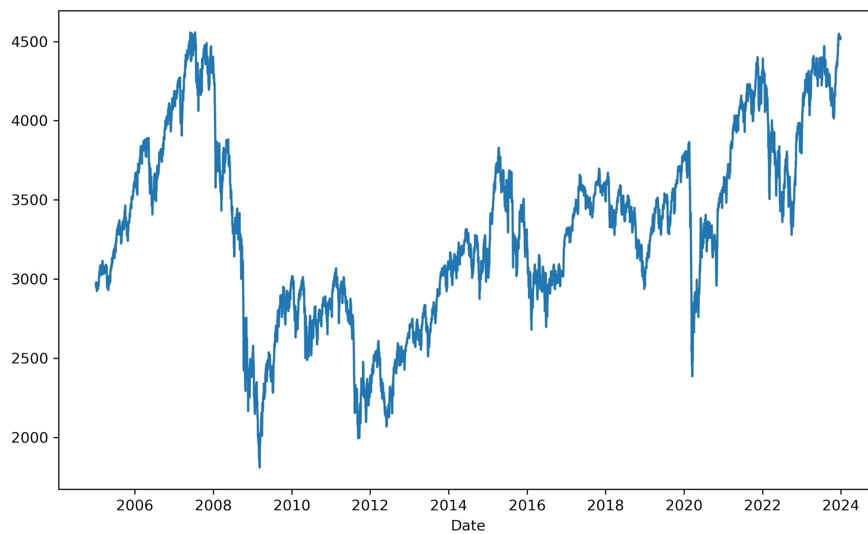

FIGURE A.8: **Euro STOXX 50 Index**. 2005-2024. Data: Google Finance. Own illustrations

A.6 Appendix F

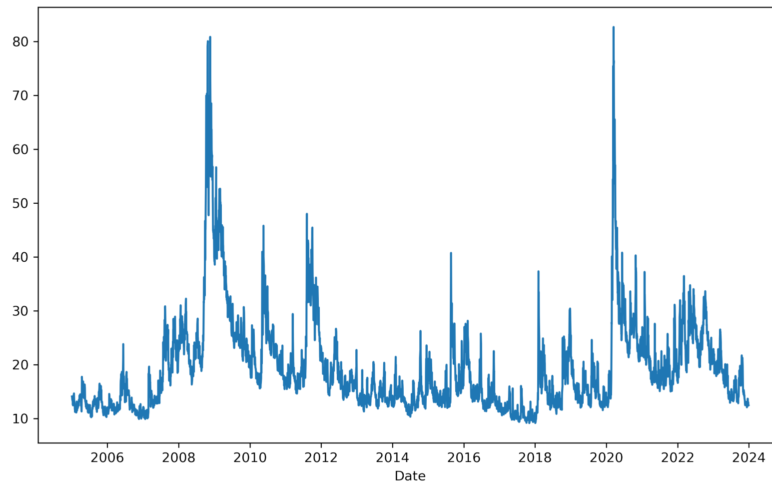

FIGURE A.9: **US Volatility Index, VIX**. For S&P500. 2005-2024. Data: Cboe. Own illustrations

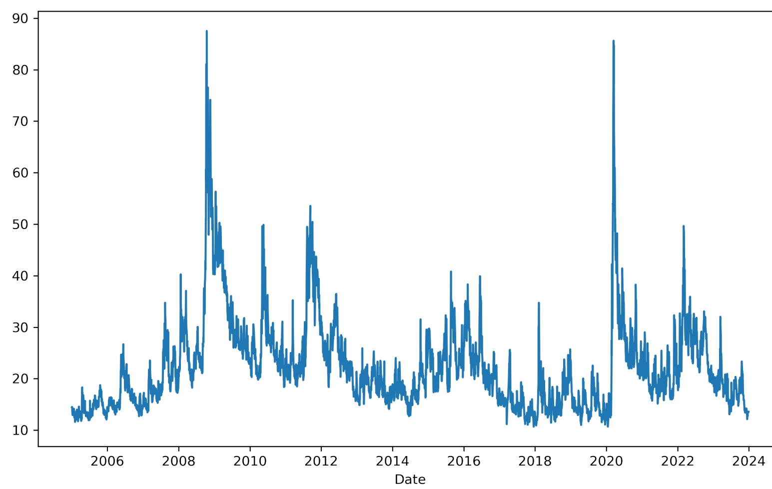

FIGURE A.10: **EU Volatility Index, VIX**. For EURO STOXX 50. 2005-2024. Data: STOXX. Own illustrations

A.7 Appendix G

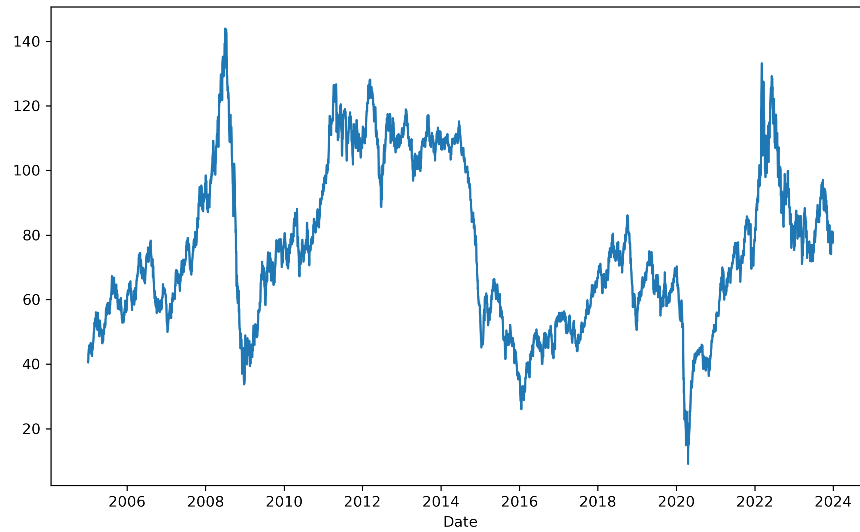

FIGURE A.11: **Brent Crude Oil**. 2005-2024. Data: Alpha Vantage API. Own illustrations

A.8 Appendix H

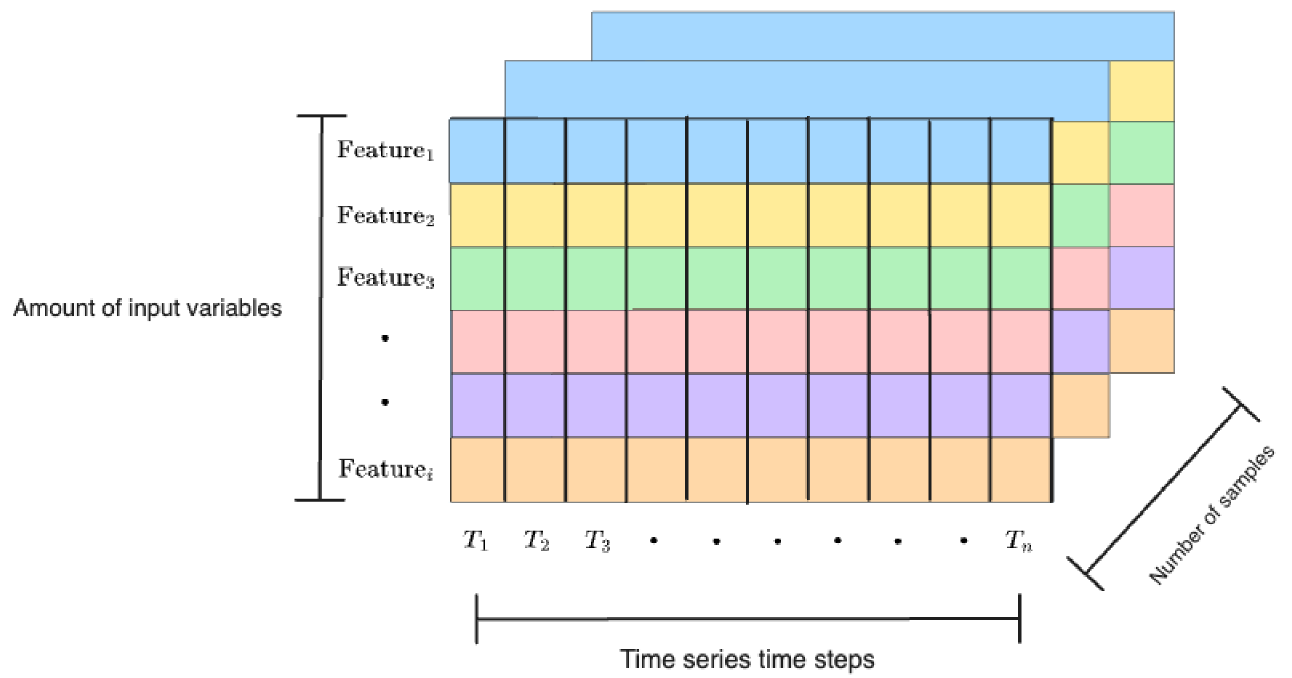

FIGURE A.12: Visualization of a tensor used for the LSTM model. Own illustration

Bibliography

Abednego, Luciana (Jan. 2018). “Forex Trading Robot with Technical and Fundamental Analysis”. In: *Journal of computers*, pp. 1089–1097. DOI: 10.17706/jcp.13.9.1089-1097. URL: https://doi.org/10.17706/jcp.13.9.1089-1097.

AbuHamad, None (Apr. 2013). “EVENT-DRIVEN BUSINESS INTELLIGENCE APPROACH FOR REAL-TIME INTEGRATION OF TECHNICAL AND FUNDAMENTAL ANALYSIS IN FOREX MARKET”. In: *Journal of computer sciences/Journal of computer science* 9.4, pp. 488–499. DOI: 10.3844/jcssp.2013.488.499. URL: https://doi.org/10.3844/jcssp.2013.488.499.

Baheti, Pragati (2021). *Activation functions in neural networks [12 types use cases]*. URL: https://www.v7labs.com/blog/neural-networks-activation-functions#h3.

Beattie, Andrew (Feb. 2022). *Forex: World39;s Biggest Market a Relative Newcomer*. URL: https://www.investopedia.com/articles/forex/10/forex-market-history.asp.

Brownlee, Jason (2021). *How to choose an activation function for deep learning*. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/?fbclid=IwZXh0bgNhZW0CMTAAAR0SRrG9HfYAKj_pbBpuKeyxXvtKTqXJr-q5MN2JaNsuSfJMTlgXWPFHSSk_aem_AeFUyevMr7GJq52RPY5gGdo7rxR8yMsap02EJDgeNx73qqShEsC98bdPhIv

Chen, James (Apr. 2024). *What is EMA? How to Use Exponential Moving Average With Formula*. URL: https://www.investopedia.com/terms/e/ema.asp.

Clay, Ben (Apr. 2024). *Unemployment impact on Forex | Blueberry Markets*. URL: https://blueberrymarkets.com/market-analysis/news/how-unemployment-rates-affect-the-forex-market/.

colah (2022). *Understanding LSTM Networks – colah’s blog*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

De Prado, Marcos Lopez (Jan. 2018). *Advances in financial machine learning*. John Wiley Sons.

– (Jan. 2018). “Advances in Financial Machine Learning: Lecture 1/10”. In: *Social Science Research Network*. DOI: 10.2139/ssrn.3270329. URL: https://doi.org/10.2139/ssrn.3270329.

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong (Apr. 2020). *Mathematics for machine learning*. Cambridge University Press.

Dolan, Brian (Mar. 2024). *What is MACD?* URL: https://www.investopedia.com/terms/m/macd.asp.

Drakopoulou, Veliota (2015). “A Review of Fundamental and Technical Stock Analysis Techniques”. In: *Journal of Stock Forex Trading*. DOI: 10.4172/2168-9458.1000163.

Duchi, John, Elad Hazan, and Yoram Singer (Feb. 2011). “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research 12* 12.61, pp. 2121–2159. URL: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2010-24.pdf.

Ebrahimi, Reza (Aug. 2023). *The Evolution of Financial Market Analysis: a journey through time*. URL: https://www.linkedin.com/pulse/evolution-financial-market-analysis-journey-through-time-ebrahimi.

Essam, Bassem (Jan. 2022). "LASSO and Ridge regularization .. simply explained - nerd for tech - medium". In: URL: https://medium.com/nerd-for-tech/lasso-and-ridge-regularization-simply-explained-d551ee1e47b7.

Fernando, Jason (Mar. 2023). *Moving Average (MA): purpose, uses, formula, and examples*. URL: https://www.investopedia.com/terms/m/movingaverage.asp.

– (Apr. 2024). *Relative Strength Index (RSI) indicator explained with formula*. URL: https://www.investopedia.com/terms/r/rsi.asp.

GeeksforGeeks (Dec. 2023). *Deep Learning introduction to long short term memory*. URL: https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/.

Gehrig, Thomas and Lukas Menkhoff (Sept. 2006). "Extended evidence on the use of technical analysis in foreign exchange". In: *International journal of finance economics/International journal of finance and economics* 11.4, pp. 327–338. DOI: 10.1002/ijfe.301. URL: https://doi.org/10.1002/ijfe.301.

Ghanoum, Tarek (Mar. 2022). "Why multicollinearity isn't an issue in Machine Learning". In: URL: https://towardsdatascience.com/why-multicollinearity-isnt-an-issue-in-machine-learning-5c9aa2f1a83a.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (Nov. 2016). *Deep learning*. MIT Press.

Gustav Cassel - Econlib (Nov. 2022). URL: https://www.econlib.org/library/Enc/bios/Cassel.html.

Hayes, Adam (Nov. 2022). *Tulipmania: About the Dutch tulip bulb market bubble*. URL: https://www.investopedia.com/terms/d/dutch_tulip_bulb_market_bubble.asp.

Hsu, Po-Hsuan, Mark P. Taylor, and Zigan Wang (2016). "Technical trading: Is it still beating the foreign exchange market?" In: *Journal of International Economics* 102. DOI: 10.1016/J.JINTECO.2016.03.012.

Jang, H. and J. Lee (Jan. 2019). "Machine learning versus econometric jump models in predictability and domain adaptability of index options". In: *Physica. A* 513, pp. 74–86. DOI: 10.1016/j.physa.2018.08.091. URL: https://doi.org/10.1016/j.physa.2018.08.091.

Keras (n.d.). *Keras documentation: Layer weight initializers*. URL: https://keras.io/api/layers/initializers/.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (Dec. 2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *neural information processing systems* 25, pp. 1097–1105. URL: http://books.nips.cc/papers/files/nips25/NIPS2012_0534.pdf.

Lezmi, Edmond and Jiali Xu (Jan. 2023). "Time Series Forecasting with Transformer Models and Application to Asset Management". In: *Social Science Research Network*. DOI: 10.2139/ssrn.4375798. URL: https://doi.org/10.2139/ssrn.4375798.

Lindholm, Andreas et al. (Mar. 2022). *Machine learning*. Cambridge University Press.

Louis, FED. St. (May 2024). *U.S. dollars to Euro spot exchange rate*. URL: https://fred.stlouisfed.org/series/DEXUSEU.

P. Kingma, Diedrik and Jimmy Li Ba (2015). "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION". In: *Published as a conference paper at ICLR 2015*. URL: https://arxiv.org/pdf/1412.6980.

Picardo, Elvis (Mar. 2024). *The Origins of Greece's Debt Crisis*. URL: https://www.investopedia.com/articles/personal-finance/061115/origins-greeces-debt-crisis.asp.

plus500 (n.d.). *The history of the EUR/USD | Plus500*. URL: https://www.plus500.com/en-dk/instruments/eurusd/the-history-of-the-eurusd~4.

Pykes, Kurtis (Jan. 2024). *Cross-Entropy Loss Function in Machine Learning: Enhancing Model Accuracy*. URL: https://www.datacamp.com/tutorial/the-cross-entropy-loss-function-in-machine-learning.

Pérez-Pons, María E. et al. (Apr. 2021). "Machine Learning and Traditional Econometric Models: A Systematic Mapping study". In: *Journal of Artificial Intelligence and Soft Computing Research* 12.2, pp. 79–100. DOI: 10.2478/jaiscr-2022-0006. URL: https://doi.org/10.2478/jaiscr-2022-0006.

Ruder, Sebastian (Jan. 2016). "An overview of gradient descent optimization algorithms". In: *Insight Centre for Data Analytics*. URL: https://arxiv.org/pdf/1609.04747.

Saâdaoui, Foued and Hana Rabbouch (Apr. 2019). "A wavelet-based hybrid neural network for short-term electricity prices forecasting". In: *Artificial intelligence review* 52.1, pp. 649–669. DOI: 10.1007/s10462-019-09702-x. URL: https://doi.org/10.1007/s10462-019-09702-x.

Snow, Richard (Mar. 2024). "Technical vs Fundamental Analysis in Forex". In: URL: https://www.dailyfx.com/education/why-trade-forex/technical-vs-fundamental-analysis.html.

Starmer, Josh (Nov. 2022). *Long Short-Term Memory (LSTM), clearly explained*. URL: https://www.youtube.com/watch?v=YCzL96nL7j0.

Thompson, Neil C. et al. (Jan. 2020). "The computational limits of deep learning". In: *arXiv (Cornell University)*. DOI: 10.48550/arxiv.2007.05558. URL: https://arxiv.org/abs/2007.05558.

Walasek, Rafał and Janusz Gajda (Aug. 2021). "Fractional differentiation and its use in machine learning". In: *International journal of advances in engineering sciences and applied mathematics* 13.2-3, pp. 270–277. DOI: 10.1007/s12572-021-00299-5. URL: https://doi.org/10.1007/s12572-021-00299-5.

Wei, Yiming (May 2022). "Empirical analysis of Ridge, Lasso, and Dropout regularizations". In: *2nd International Conference on Applied Mathematics, Modelling, and Intelligent Computing (CAMMIC 2022)*. DOI: 10.1117/12.2639002. URL: https://doi.org/10.1117/12.2639002.

Yıldırım, Deniz Can, Ismail Hakkı Toroslu, and Ugo Fiore (Jan. 2021). "Forecasting directional movement of Forex data using LSTM with technical and macroeconomic indicators". In: *Financial innovation* 7.1. DOI: 10.1186/s40854-020-00220-2. URL: https://doi.org/10.1186/s40854-020-00220-2.