A Comparison of Machine Learning Methods for Financial Time Series Forecasting

MASTER THESIS for the degree of

Master of Science

presented at the Department IV
of Trier University

by

Moritz Ahl
Zuckerbergstraße 10
54290 Trier

Supervisor: Prof. Dr. Volker Schulz
Second supervisor: Dr. Christian Vollmann

Trier, January 27, 2022

Declaration for the master's thesis

I hereby declare that I have written this Master's thesis independently and that I have not used any other sources and aids other than those indicated and that the thoughts taken directly or indirectly directly or indirectly from external sources. The master thesis has not been submitted to any other examination office in the same or a comparable form. It has also not been published to date.

Erklärung zur Masterarbeit

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Masterarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

January 27, 2022

(Date) (Signature)

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The objective of this Master's thesis is to deliver an overview over several machine learning methods used in econometrics. This topic is a matter of interest for the financial industry since the requirements for reporting and risk management increased over the last two decades, for example with the entry into force of the BASEL accords, as a consequence of the 2007/2008 global financial crisis. Reliable and easily accessible forecasting of returns and trading volume based on data and machine learning methods allows participants of the markets to predict changes and can possibly help to avoid running into price shocks.

The author first came in touch with the topic through a seminar at Trier University in the winter term 2020/2021 that dealt with quantitative risk management and in particular with financial time series using several autoregressive models. Chapter 2 deals with these models and how they can be statistically tested and fitted. It is followed by an introduction to machine learning and the idea of neural networks. An example of a feedforward neural network is also given at the end of the chapter. Next is the presentation of several advanced neural networks culminating in Transformer networks. In the following chapter, the implementation of the different tuning methods for neural networks is discussed and the results for forecasting financial market data using advanced neural networks are compared. The part of the thesis discussing the theory of neural networks is inspired by the work of Dixon, Halperin and Bilokon ([48]) who provide an introduction to the application of machine learning to finance. To conclude, the last chapter includes a summary of what we have learned and provides the reader with an outlook on further research and applications in the financial industry.

This work has been written in such a way that it can be understood by as wide a readership as possible. All essential concepts will be introduced and defined in a mathematically correct way. Basic knowledge of linear algebra, statistics, measure theory, numerics, economics and finance will nevertheless contribute to a better understanding. The ultimate goal is to provide the reader with a broad overview over financial time series forecasting and the zoo of neural networks that can be used to forecast such series.

If not stated otherwise, all sketches of neural networks and their components have been created using the online drawing tool draw.io. All programming codes used to determine the results presented in the course of this thesis can be found in the attachment and have been tested successfully for Python (Version 3.9), Jupyter Notebook (Version 6.4.5) and TensorFlow (Version 2.7.0), all run on a Windows 10 operating system.

Nice introduction to technical analysis in [22], Section 2.2

Acknowledgements.
The author wants to thank all of those who contributed to this thesis, in particular Prof. Dr. Volker Schulz who supervised and supported the process of writing this work and Dr. Christian Vollmann who agreed on being the second supervisor. Further thanks go to my parents, who supported me in many ways during my studies.

2 Financial Time Series

The content in this chapter is following Chapter 4 of the work [53] in combination with Chapter 6 in [48]. More detailed introductions to time series and their analysis can be found in [30] and [180], of which the primer one serves as the reference for the proofs left out here and the latter is regarded as standard literature of the field of econometrics.

In this chapter, after defining what a (financial) time series is and what the most striking challenges are, we will take a look at different *autoregressive models*, smoothing techniques and methods for dimensionality reduction. Moreover, methods for the fitting of autoregressive models and the generation of prediction will be introduced. The chapter will be concluded by a short summary of what has been learned.

2.1 Basics and Motivation

Before the focus will be on different models and their application, we need to establish the theoretical basis and specify in mathematically correct terms what we will be talking about.

2.1.1 Financial Time Series Analysis

First of all, let us define what a time series is:

Definition 2.1 (Time Series)**.** *A* **stochastic process** *is a sequence of random variables* $X = (X_t)_{t\in\mathbb{R}}$ *defined on a probability space* $(\Omega, \mathcal{F}, P)$*. A sequence of observations at discrete times of a stochastic process* $X = (X_t)_{t\in I=\{1,\dots,n\}}$ *is called a* **time series**.

In the following, if not stated otherwise we assume that $I = 1, \dots n$ and that the filtration $\mathcal{F} = (\mathcal{F})_{t\in\mathbb{R}}$ to be the *natural filtration* of the process X defined by

$$\mathcal{F}_t = \sigma(\{X_s : s \leq t\}). \tag{2.1}$$

In other words, the filtration is the σ-algebra created by the market information up to time t. The process X is assumed to be $\mathcal{F}$-adapted, meaning that X_t is $\mathcal{F}_t$-measurable for all $t \in I$.

A **financial time series (FTS)** is a time series that contains data from the financial markets, for example the prices of a stock over a given period or the returns of a fund.

To analyze financial time series, we need to introduce the concepts of *autocovariance*, *autocorrelation* and *stationarity*.

Definition 2.2 (Autocovariance)**.** *The* **autocovariance function** *of a time series X is given by*

$$\gamma(t,s) = \mathbb{E}[(X_t - \mu_t)(X_s - \mu_s)] = Cov[X_s, X_t], \tag{2.2}$$

where $\mu_t := \mathbb{E}[X_t]$ and $h := |t-s|$ is referred to as the **lag**.

Definition 2.3 (Autocorrelation)**.** *The* **autocorrelation function (ACF)** *is defined as*

$$\rho_h := \rho(X_t, X_s) := \frac{\gamma(t,0)}{\gamma(s,0)}, \tag{2.3}$$

where $h := t - s$ denotes the lag once again.

Definition 2.4 (Partial Autocorrelation)**.** *The* **partial autocorrelation function (PACF)** *is defined as*

$$\pi := \Gamma_k^{-1}\delta_k, \tag{2.4}$$

where $\Gamma_k = (v_{ij})_{ij}$ is the $k \times k$ autocovariance matrix with $v_{ij} = \gamma(i,j)$ and δ_k is the k-dimensional column vector with $(\delta_k)_i = \rho_i$.

The autocovariance describes the relation between the covariances of a process at different points of time s and t. The autocorrelation function describes the correlation within a time series for a lag h. Finally, the partial autocorrelation function can be understood as the conditional correlation between X_s and X_t given the intermediate observations $X_{s+1}, \ldots, X_{t-1}$.

We are now in a position to proceed by defining two versions of stability, in statistics better known as *stationarity*, which can be interpreted as the characteristic that a time series behaves similar at all points of time.

Definition 2.5 (Stationarity)**.** *A time series $X = (X_t)_{t\in I}$ with index $I = \{1, \ldots, n\}$ is called* **strictly stationary** *if for all $t_1, \ldots, t_m, k \in I$ such that $t_{i+k} \leq n$ for all $i = 1, \ldots, m$ there holds*

$$(X_{t_1}, \ldots, X_{t_m}) \overset{d}{=} (X_{t_1+k}, \ldots, X_{t_m+k}),$$

where $\overset{d}{=}$ means equality in distribution.

X is called **(weakly) covariance stationary** *if the first two moments of X exist and for k such that $t + k \leq n$ there holds*

$$\mu_t = \mu, \quad t \in I,$$

$$\gamma(t,s) = \gamma(t+k, s+k), \quad t > s \in I \text{ and } k \text{ such that } t + k \leq n.$$

The latter form of stationarity means that the time series X has the same expected value for all t and the same autocovariance for all lags $h = t - s$ over time.

2.1.2 ARMA Processes

Our goal is to create an econometrical model that captures the market prices for stocks in the best way possible. In the literature, it is common to look at the logarithm of the returns, often called *log returns*.

Definition 2.6 (Log Returns)**.** *For a time series* $X = (X_t)_{t\in I}$ *the* **logarithmic** *or* **log returns** $r = (r_t)_{t\in\{1,\dots n-1\}}$ *are given by*

$$r_t = \log\left(\frac{X_{t+1}}{X_t}\right), \quad t = 1, \dots, n-1. \tag{2.5}$$

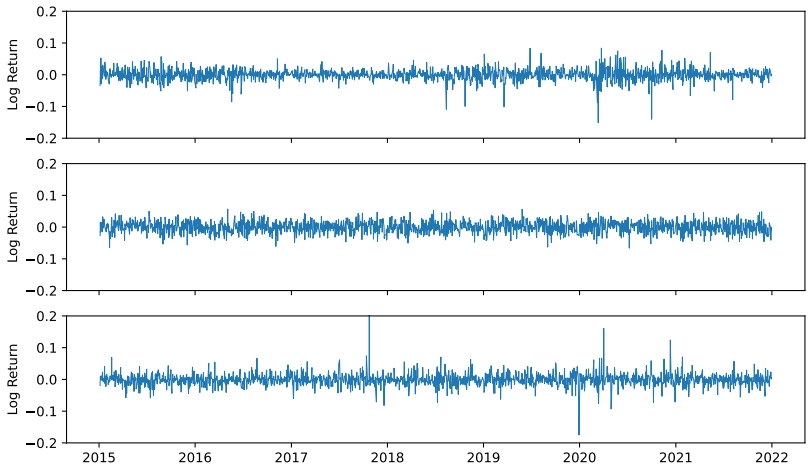


Figure 2.1: (Top) Daily log returns, (middle) simulated normally distributed log returns and (bottom) simulated t-distributed log returns for Bayer stock

Figure 2.1, displays the plots of daily log returns for the stock of the German Bayer company retrieved from Yahoo!Finance using the `yfinance` library for Python. A first approach could be to compare the returns to independent and identically distributed (iid) samples, in particular normally distributed with the expected value and standard deviation obtained in the historical data. If we run a simulation for this and plot the results (cf. middle of Figure 2.1), then we can see that the pattern is not very similar. The market data shows a clustering of periods with higher volatility and more extreme values than the normally distributed simulation. The latter one can be expressed in more technical terminology by: the real distribution has heavier tails or is leptokurtic. In a next step, a t-distribution to the data set and ran another simulation drawing the returns from this distribution. The bottom graph of Figure 2.1 shows that the returns now show more extreme values but the clustering is still not captured in a satisfactory manner.

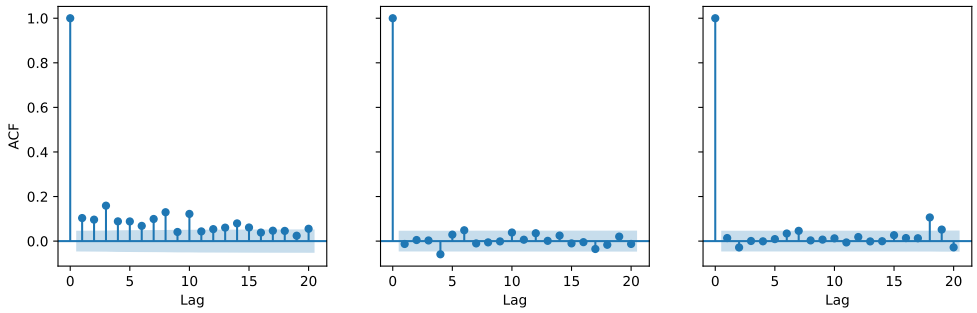


Figure 2.2: The autocorrelation functions for (left) the actual return data, (middle) the normal and (right) the t-simulation for absolute returns using the Python package statmodels

Furthermore, the autocorrelation functions for the absolute returns which can be found in Figure 2.2, where the blue area indicates the 95% confidence interval, indicate that there is a profound serial correlation within the return series and the same holds for squared return series. Both normally and t-distributed time series lack this property.

The code for implementating the computation and plotting of the log returns, as well as the samples drawn from fitted normal and t-distribution can be found in the Jupyter notebook `FTS_Intro` in the attachment.

The observations on the return series data from above are part of the so called **stylized facts** in econometrics. Embrechts, Frey and McNeil name six empirical observations in [53] that after long econometric experience have been elevated to status of facts. These facts are:

- Little serial correlation for return series but not iid,
- Profound serial correlation for absolute and squared return series,
- Conditional expected returns are close to zero,
- Varying volatility over time,
- Heavy-tailed or leptokurtic return series, and
- Clustering of extreme values.

To improve the modeling and account for the correlation between the data, a common approach is to introduce so called **autoregressive models** and **moving average models** in which the value of the time series in the last observation points is affecting the value today or weighted shocks drive the process respectively. A mathematical justification, why we should take a closer look at these types of models, is the famous **Wold representation theorem**, which states that certain time series can be represented by the sum of a deterministic and a weighted stochastic time series.

Theorem 2.7 (Wold)**.** *Every covariance stationary financial time series $X = (X_t)_{t\in I}$ can be written as*

$$X_t = \sum_{j=0}^{\infty} \psi_j Z_{t-j} + D_t, \tag{2.6}$$

where

- $\psi = (\psi_t)_{t\in\{0,\dots,\infty\}}$, $Z = (Z_t)_{t\in\{0,\dots,\infty\}}$, $D = (D_t)_{t\in\{0,\dots,\infty\}}$ *are unique,*
- $\psi_0 = 1$, $\sum_{j=0}^{\infty} \psi_j^2 < \infty$,
- $Z \sim \mathcal{N}(0, \sigma^2)$,
- *Z and D are uncorrelated, i.e. $Cov(Z_t, D_s) = 0$ for all $s, t \in I$, and*
- *D is deterministic.*

For further theory on the Wold representation, the reader is referred to the article by Iwok ([92]) and a proof of the representation theorem can be found in Theorem 7.6.7 in [10].

Arguably the most famous autoregressive model used in econometrics is the so called *autoregressive moving average* or *ARMA(p,q) model* for financial time series.

Definition 2.8 (ARMA Process)**.** *Let $p, q \in \mathbb{N}_0$, $\{\Phi_i\}_{i=1,\dots,p} \in \mathbb{R}^p$, $\{\theta_i\}_{i=1,\dots,q} \in \mathbb{R}^q$ and $I = \{1, \dots, n\}$ for $n \in \mathbb{N}$. A process $X = (X_t)_{t\in I}$ is called an* **zero-mean autoregressive moving-average process (ARMA(p,q))***, if X is covariance stationary and there holds*

$$\begin{aligned} X_t - \Phi_1 X_{t-1} - \dots - \Phi_p X_{t-p} &= \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q} \\ \Leftrightarrow X_t &= \sum_{i=1}^{p} \Phi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t, \quad \forall t \in I, \end{aligned} \tag{2.7}$$

where $\epsilon = (\epsilon_t)_{t\in I}$ is a **white-noise process***, i.e. for ϵ there holds*

- *$\mathbb{E}[\epsilon_t] = 0$ for all $t \in I$,*
- *$Var[\epsilon_t] = \sigma^2$ for all $t \in I$,*
- *ϵ_s and ϵ_t are independent for all $s, t \in I$ with $s \neq t$.*

For ARMA models, p and q denote the **order** *of the autoregressive and the moving average part of the model respectively.*

Remark 2.9. *A process X is called an* **autoregressive process (AR(p))** *of order p if $q = 0$ and an* **moving-average process (MA(q))** *of order q if $p = 0$. Combining these two models into one allows for a higher flexibility of the model (cf. [48], p.202).*
There also exist ARMA processes with mean μ. In this case one considers the centered process $(X_t - \mu)_{t\in I}$.

The AR-part of the model ($X_t = \epsilon_t + \sum_{i=1}^{p} \Phi_i X_{t-i}$) describes the process as a linear combination of past observations and adds some white-noise. On the other hand, the

MA-part ($X_t = \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i}$) can be regarded as the weighted average of (white-noise) shocks.

The order or lag p of an ARMA process can be determined by investigating the corresponding partial autocorrelation function. On the other hand, the autocorrelation function hints the order q of the moving average part.

Using a **lag operator** B, such that $BX_t = X_{t-1}$, we can rewrite the model in Equation (2.7) as

$$(1 - \Phi_1 B - \cdots - \Phi_p B^p) X_t = (1 + \theta_1 B + \cdots + \theta_q B^q) \epsilon_t. \tag{2.8}$$

From this, two polynomials can be retrieved for $z \in \mathbb{C}$, the **AR polynomial**

$$\tilde{\Phi}(z) := 1 - \Phi_1 z - \cdots - \Phi_p z^p \tag{2.9}$$

and the **MA polynomial**

$$\tilde{\theta}(z) := 1 + \theta_1 z + \cdots + \theta_q z^q, \tag{2.10}$$

which will be investigated further to characterize some of the properties of ARMA processes. These polynomials are often referred to as the **characteristic equations** of the ARMA process. A first result provides us with an equivalence criterion for the existence of a stationary process.

Lemma 2.10. *If and only if $\tilde{\Phi}(z)$ and $\tilde{\theta}(z)$ have no common roots and $\tilde{\Phi}(z)$ has no roots on the unit circle, i.e. $\tilde{\Phi}(z) \neq 0$ for all $z \in \mathbb{C}, |z| = 1$, then there exists a unique stationary solution of the ARMA equation (2.7).*

Proof: A proof can be found on pp.74-75 in [30].

Some ARMA processes have additional properties allowing a representation of the process X by linear combinations of past noises ϵ (*causality*) or a representation of the latest noise by a linear combination of past observations (*invertibility*). These two properties can be defined formally as follows:

Definition 2.11 (Causal ARMA Process)**.** *We say that an ARMA(p,q) process $X = (X_t)_{t \in I}$ is* **causal***, if there exist a set $J \subset \mathbb{N} \cup \{\infty\}$ and constants $(\psi_j)_{j \in J}$ such that $\sum_{j \in J} |\psi_j| < \infty$ and*

$$X_t = \sum_{j \in J} \psi_j \epsilon_{t-j}, \quad \text{for all } t \in I. \tag{2.11}$$

Definition 2.12 (Invertible ARMA process)**.** *An ARMA(p,q) process $X = (X_t)_{t \in I}$ is called* **invertible***, if there exist a set $M \subset \mathbb{N} \cup \{\infty\}$ and constants $(\pi_m)_{m \in M}$ such that $\sum_{m \in M} |\pi_m| < \infty$ and*

$$\epsilon_t = \sum_{m \in M} \pi_m X_{t-m}, \quad \text{for all } t \in I. \tag{2.12}$$

For both properties, a characterization applies in each case, based on the roots of one of the characteristic equations $\tilde{\Phi}(z)$ or $\tilde{\theta}(z)$ respectively.

Lemma 2.13. *An ARMA(p,q) process $X = (X_t)_{t\in I}$ is causal, if and only if the polynomial $\tilde{\Phi}(z)$ has no roots in the unit circle, i.e.*

$$\tilde{\Phi}(z) := 1 - \Phi_1 z - \cdots - \Phi_p z^p \neq 0, \quad \text{for all } z \in \mathbb{C} \text{ with } |z| \leq 1. \tag{2.13}$$

Lemma 2.14. *An ARMA(p,q) process $X = (X_t)_{t\in I}$ is invertible, if and only if the polynomial $\tilde{\theta}(z)$ has no roots in the unit circle, i.e.*

$$\tilde{\theta}(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \neq 0, \quad \text{for all } z \in \mathbb{C} \text{ with } |z| \leq 1. \tag{2.14}$$

Once again, we refer to Section 3.1 in [30] for the derivation of and extended theory on the last two lemmas. Indeed, most models fitted in practice and to real data are both causal and invertible solutions of the defining Equation (2.7) (cf. [53], p. 132). To gain a a better understanding of ARMA processes, the following example illustrates some of the key aspects of this class of processes by plotting sample paths using the Python package `statsmodels` for which the code can be found in the notebook `ARMA_Examples`.

Example 2.15. *In this example, exemplary MA, AR and ARMA processes will be simulated.*

1. *An AR(1) process is given by*

$$X_t = \Phi_1 X_{t-1} + \epsilon_t, \tag{2.15}$$

for all $t \in I$. Figure 2.3 shows the impact of different parameter values for Φ_1, where a higher (absolute) value for Φ_1 leads to a higher (absolute) autocorrelation for the lags and for $\Phi_1 < 0$, the autocorrelation is alternating between positive and negative values.

By iterating the expression for X_t, we get

$$\begin{aligned} X_t &= \Phi_1 \left(\Phi_1 X_{t-2} + \epsilon_{t-1}\right) + \epsilon_{t-2} \\ &= \Phi_1^{k+1} X_{t-k-1} + \sum_{i=1}^{k+1} \Phi_1^i \epsilon_{t-i} \end{aligned} \tag{2.16}$$

for all $t \in I$ and k such that $t - k - 1 \geq 0$. Thus, an AR(1) process can be represented as an MA(∞) process and is causal, if $|\Phi_1| < 1$ and k large.

2. *An MA(q) process, $q \in \mathbb{N}$, is given by*

$$X_t = \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t, \tag{2.17}$$

for all $t \in I$. This is obviously a causal process and a sample path for $q = 4$ plotted together with its autocorrelation function in Figure 2.4 shows that the latter one cuts off at lag q as desired, i.e. $\rho_h = 0$ for all $|h| > q$.

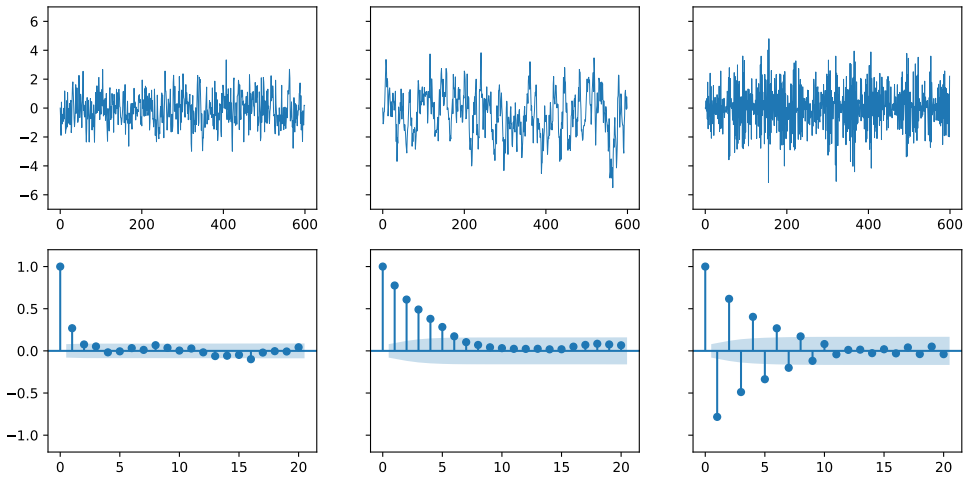


Figure 2.3: (Top) AR(1) process with $n = 600$ and (bottom) corresponding autocorrelation functions for values $\Phi_1 \in \{0.3, 0.8, -0.8\}$ and 20 lags

3. *Combining both models, an ARMA(1,1) process is given by*

$$X_t = \Phi_1 X_{t-1} + \theta_1 \epsilon_{t-1} + \epsilon_t, \tag{2.18}$$

for all $t \in I$. The characteristic polynomials are given by

$$tilde\Phi(z) = 1 - \Phi_1 z \quad and\tilde{\theta}(z) = 1 + \theta_1 z \tag{2.19}$$

respectively. Assuming that $\theta_1 + \Phi_1 \neq 0$, they do not have any common roots and the root for $\tilde{\Phi}(z)$ is given by $z^* = \frac{1}{\Phi_1}$. For $|\Phi_1| < 1$ the root is outside the unit circle and thus the process is causal and can be represented as an MA(∞) process. With the additional condition $|\theta_1| < 1$, it is possible to express X as an AR(∞) process ([53], p.131).

A sample path for such an ARMA(1,1) process together with its autocorrelation function is presented in Figure 2.5.

There exist further extensions to the concept of ARMA-processes: **Autoregressive integrated moving-average (ARIMA(p,d,q))** models are stationarized through differentiation, where d gives the order of differentiation necessary to achieve stationarity. This idea can be generalized further by taking seasonalities into account. An application of a **seasonal ARIMA (SARIMA)** model in derivative markets can be found in [18], where the authors use this approach to capture the seasonal patterns of electricity prices which are dependent on the weather and the seasons.

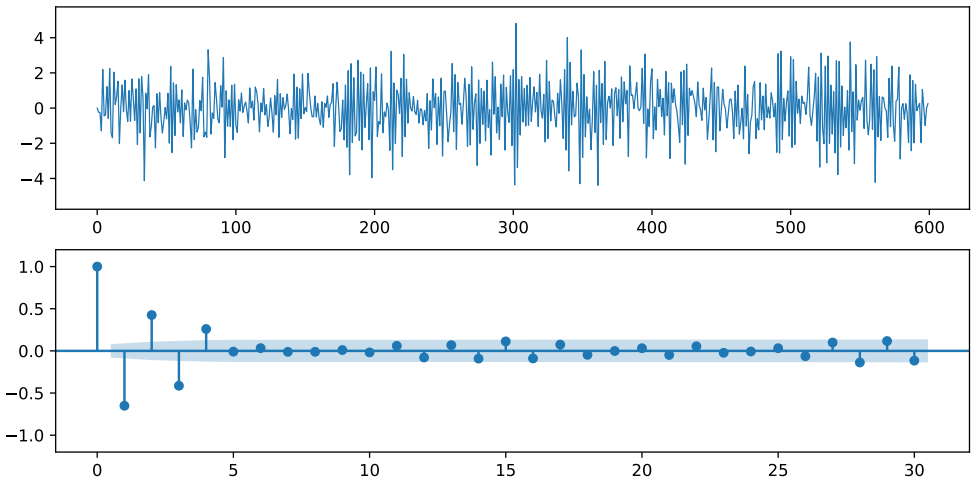


Figure 2.4: (Top) Sample path for an MA(4) process with $n = 600$ and (bottom) corresponding autocorrelation function for values $\theta_1 = -0.8, \theta_2 = 0.4, \theta_3 = -0.5, \theta_4 = 0.7$ and 30 lags

Nevertheless, Tsay ([180], p.56) mentions that ARMA models are rarely used successfully in practice. However, the models presented in the next subsection are used more often in the industry to capture the variance of a time series.

2.1.3 GARCH Processes

The ARMA models introduced in the previous subsection allow for a modeling of the conditional mean

$$\mathbb{E}\left[X_t|\mathcal{F}_{t-1}\right], \quad t \in I \setminus \{0\}, \tag{2.20}$$

where $\mathcal{F} = (\mathcal{F}_t)_{t\in I}$ is the natural filtration. In this subsection we will present a different autoregressive model to structure the conditional variance

$$\mathrm{Var}\left[X_t|\mathcal{F}_{t-1}\right], \quad t \in I \setminus \{0\}, \tag{2.21}$$

that is **conditionally heteroscedastic**. Heteroscedastic means that the conditional variance varies over time. Consequently, these models are called *generalized autoregressive conditionally heteroscedastic (GARCH(p,q))* processes where an ARMA(p,q) approach is used to model the squared conditional variance.

In 1982 a first model to introduce nonconstant conditional but constant unconditional variances was presented by Engle ([54]). This *ARCH(p) model* was another step towards modeling the "stylized facts" of econometrics in an appropriate manner. Four years later,

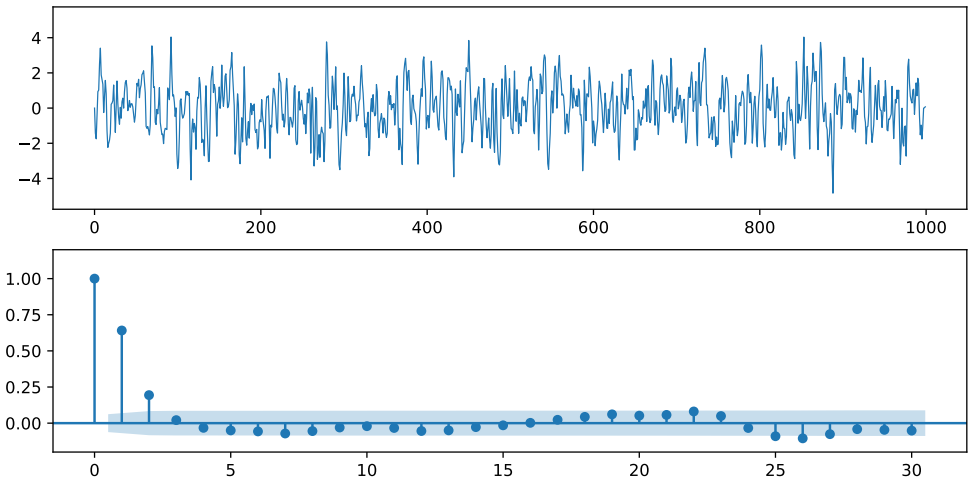


Figure 2.5: (Top) Sample path for an ARMA(1,1) process with $n = 1000$ and (bottom) the corresponding autocorrelation function for values $\theta_1 = 0.4, \Phi_1 = 0.6$ and 30 lags

Bollerslev ([21]) generalized the model by, in addition to the dependence on previous squared values of the process, including lagged conditional variances to impact the current conditional variance as well.

Definition 2.16 (GARCH Process)**.** *A process $X = (X_t)_{t\in I}$ is called a* **generalized autoregressive conditionally heteroscedastic (GARCH(p,q))** *process, if X is strictly stationary and*

$$\begin{aligned} X_t &= \sigma_t \epsilon_t, \\ \sigma_t^2 &= \alpha_0 + \sum_{i=1}^{p} \alpha_i X_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2, \end{aligned} \tag{2.22}$$

where $\sigma = (\sigma_t)_{t\in I}$ is a strictly positive-valued process, $\alpha_0 > 0$, $\alpha_i \geq 0 (i = 1, \ldots, p)$, and $\beta_j \geq 0 (j = 1, \ldots, q)$. Moreover, $\epsilon = (\epsilon_t)_{t\in I}$ is a strict white-noise process, i.e. a series of iid random variables with finite values, mean 0 and variance 1.
The non-negative integers p and q are called **orders** *of the process.*

Note that the innovations or noises $(\epsilon_t)_{t\in I}$ can follow any distribution with mean 0 and variance 1. Unless otherwise stated, we assume Gaussian innovations, i.e. $\epsilon_t \sim \mathcal{N}(0, 1)$.

We desire to have a covariance stationary process. Bougerol and Picard ([25]) provide and prove a necessary and sufficient condition to characterize this form of stationarity. The core idea of the condition is to guarantee $\mathbb{E}[|X_t|] < \infty$ for all $t \in I$.

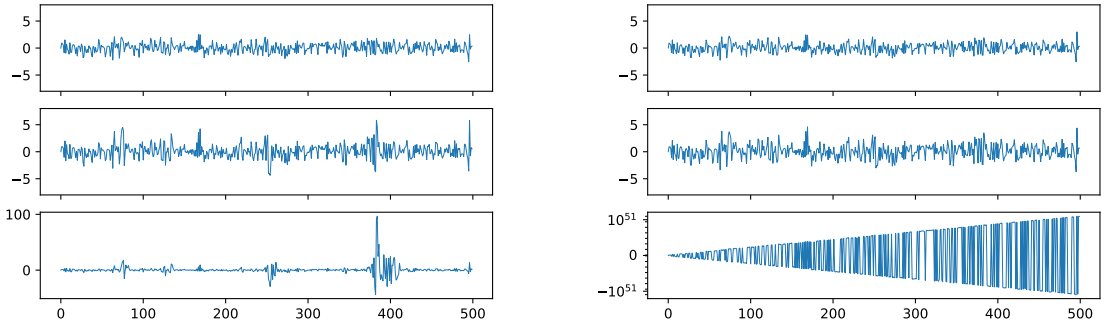


Figure 2.6: Comparison of sample paths ($n = 500$) for a GARCH(1,1) process. (Left) Different values for $\alpha_1 \in \{0.1, 0.5, 1.0\}$ with fixed $\alpha_0 = 0.5$, $\beta_1 = 0.2$ and (right) different values for $\beta_1 \in \{0.1, 0.5, 1.5\}$ with fixed $\alpha_0 = 0.5$, $\alpha_1 = 0.2$

Lemma 2.17. *A GARCH(p,q) process is covariance stationary, if and only if*

$$\sum_{i=1}^{p} \alpha_i + \sum_{j=1}^{q} \beta_j < 1. \tag{2.23}$$

To develop an intuition for GARCH(p,q) processes, it is advisable to first investigate GARCH(1,1) models. In addition, it should be mentioned that it is common to use low order GARCH models in practice.

Example 2.18. *A GARCH(1,1) process is given by*

$$\begin{aligned} X_t &= \sigma_t \epsilon_t, \\ \sigma_t^2 &= \alpha_0 + \alpha_1 X_{t-1}^2 + \beta_1 \sigma_{t-1}^2. \end{aligned} \tag{2.24}$$

We use the Python programm `GARCH_Example` *to create sample paths for GARCH(1,1) processes with different values for α_1 and β_1. Figure 2.6 shows the results. Note that the y-axis of the bottom right plot is logarithmic and covers a range of values from 10^{-51} to 10^{50}. We can observe in this plot that the process "explodes" for bad selections of the parameters, i.e. for a choice of α_1 and β_1 violating the condition for covariance stationarity given in (2.23). The two upper paths on the left and right show stationarity patterns and also the desired volatility clustering with periods of lower and higher volatility.*

In the last decades, several versions and extensions to the GARCH(p,q) model have been introduced. **Integrated GARCH (IGARCH)** processes are a restricted version of GARCH processes for which there holds

$$\sum_{i=1}^{p} \alpha_i + \sum_{j=1}^{q} \beta_j = 1. \tag{2.25}$$

In other words, the characteristic equation of the process has a unit root. Another modification are so called **GARCH in the mean (GARCH-M)** models where one extends the generalized model by introducing a constant mean μ and a constant risk premium c to model a return series

$$R_t = \mu + c\sigma_t^2 + X_t. \tag{2.26}$$

There are different ways to introduce the volatility term, e.g. by using the logarithm or the non-squared value of σ_t instead.

Nelson ([121]) proposes another modified model to handle leverage effects and tackle some of the limitations of GARCH models in practice, such as contraints through the neccesity of positive parameters $\{\alpha_i\}_{i=1,\dots,p}$ and $\{\beta_j\}_{j=1,\dots,q}$ or the fact that the basic model assumes that only the magnitude but not the sign of the excessive return has an impact on the process. This model is well known as the **exponential GARCH (EGARCH)** model and the name is derived from the fact that the conditional volatility is given by $\ln(\sigma_t^2)$ in this model which is described through a linear combination of weighted innovations with real-valued parameters ([28]). This idea can be extended by introducing a threshold term in form of an indicator into the model. A model with this modification is referred to as a **threshold GARCH (TGARCH)** model, which has been presented first by Glosten, Jagannathan and Runkle in [65]. In Chapter 3 of [180], Tsay gives a detailed overview over different variants of autoregressive models including the ones mentioned above.

In applications, it is common to combine both the ARMA and the GARCH model in a way that GARCH processes are used to model the noise ϵ_t in the ARMA process. Framed mathematically, this can be written as

$$\epsilon_t = \sigma_t Z_t, \tag{2.27}$$

where $\{Z_t\}_{t\in I}$ follows a GARCH model for conditional volatility.

So far the focus has been on modeling every single time step of a time series. Of, for example, the observation points I are close together in terms of time, it can be useful to observe the averaged or smoothed time series instead of individual observations. This concept will be dealt with in the next section which follows the introductions by Embrechts et al. ([53], p.138) and Dixon et al. ([48], Section 6.2.10).

2.1.4 Smoothing Techniques

Smoothing can be helpful if the data at hand has rough patterns which is often the case for financial market data. However, this approach also provides us with a model which can generate predictions. **Exponential smoothing** is used to generate smoothed predictions $\tilde{X}_{t+1}$ by exponentially decreasing the weight of past observations by a parameter α which is often set between 0 and 1. This smoothing method takes the forecast at the previous point of time $\tilde{X}_t$ and adjusts the new prediction $\tilde{X}_{t+1}$ by the prediticion error $X_t - \tilde{X}_t$,

where X_t is the actual observed value at time t. Thus, our new forecast is given by

$$\begin{aligned}\tilde{X}_{t+1} &= \tilde{X}_t + \alpha\left(X_t - \tilde{X}_t\right) \\ &= \alpha X_t + (1-\alpha)\,\tilde{X}_t.\end{aligned} \tag{2.28}$$

Iterating this equation back over time yields

$$\tilde{X}_{t+1} = \sum_{i=0}^{t} \alpha\,(1-\alpha)^i\, X_{t-i}. \tag{2.29}$$

Thus, a long-term model has been introduced and the predicition depends on the entire data observed up to time $t+1$ and not only on subsequence like in AR(p) models. Furthermore, we can conclude from Equation 2.29 that the larger α is chosen, the more weight is put on recent observations.

This smoothing approach is suitable for more general models and does not require that the data we use as an input to the model to be stationary. As for most econometric models for time series, deterministic seasonal components should be excluded from the data before it is handed to the model.

However, exponential smoothing is limited when it comes to reproducing the trend of data in the forecasts. In 1957, Charles Holt introduced an improved smoothing method which is known as **Holt's linear trend method** ([82]). This method includes two dimensions of smoothing, one for the level and one for the trend or slope of the time series at time t. In most cases this approach generalizes better to forecasts than simple exponential smoothing. Over the years, the method has been improved by adding exponential trends, damping and seasonalities for which exemplary applications can be found in [95].

As mentioned above, the input required to generate a forecast for a time series using an exponential smoothing model can be large. In addition, we have concentrated on univariate time series so far. However, it is obvious that observations on other time series of the market can be helpful to estimate the future performance of the time series of interest, e.g. if there is a strong correlation to other data. Gathering more data and feeding it to the model increases the complexity of the task and therefore it is desirable to compress the data, i.e. reduce the dimensionality. In the next section, a method to accomplish this will be introduced.

2.1.5 Principle Component Analysis

Let us assume that we have an input X_i consisting of d-dimensional data. **Principal Component Analysis (PCA)**, originally introduced by Pearson in 1901 ([132]), offers a concept to reduce the dimensionality of the data to $m < d$ while conserving the most relevant and valuable information. Mathematically speaking, PCA finds a matrix $W \in \mathbb{R}^{m\times d}$ which transforms the input vector X_i to

$$X_i^* := WX_i \in \mathbb{R}^m. \tag{2.30}$$

The columns of the matrix W are determined by first finding the unit vector w_1 for which the whole input data set $X = (X_i)_{i=1,\dots,n}$ has the maximum variance. The maximum variance indicates that this direction contains the most relevant information and is therefore selected as the first axis. Next, from the set of unit vectors orthogonal to w_1, we pick the one with the maximum variance and repeat this procedure iteratively until the d orthogonal bases have been selected. This can be understood as an orthogonal transformation of the coordinate system of the data onto a vector space spanned by $(w_1, \dots, w_d)$ ([156]).

In practice, PCA is efficient in detecting linear data relationships but it incapable to handle non-linear or even piecewise linear dependencies in complex data sets ([182]). Several non-linear versions of PCA such as Kernel PCA ([156]) have been proposed and proven their ability to capture more complex relations. An overview over these techniques can be found in [32]. On the other hand, the rise of deep learning also provided us with a neural network structure called **Autoencoder (AE)** which is able to perform highly effective dimensionality reduction and shall be the focus of our interest in Section 5.4.

2.2 Fitting of Autoregressive Models

This section aims at presenting a brief overview over the procedures for fitting an ARMA or a GARCH model to a given time series. The content summarizes the famous *Box-Jenkins approach* presented in Section 6.3 in [48] and other steps presented in Sections 4.2.4 and 4.3.4 in [53].

Intuitively, when fitting an ARMA-GARCH model one would consider first fitting the ARMA process before treating the residuals of the ARMA model by a GARCH model. However, this is not the preferred modus operandi for advanced and complex tasks. It is rather recommended to fit the models simultaneously which is easily possible with modern statistical software. Time series models are known for more than 75 years and the procedure of choice to fit and access an AR(p) model is called the **Box-Jenkins approach** ([26]). The three basic steps of this modeling approach, which favors models with fewer parameters, are

1. **I**dentification - determine the order of the model,
2. **E**stimation - estimate model parameters, and
3. **D**iagnostics - evaluate the fit of the model.

It is important to always keep in mind, that we want to optimize the tradeoff between biases and variances. In other words, a model that is not adequately fitted by choosing a model with too few parameters or too low dimensionality, can cause the model to miss the relevant relations and thus results in large bias terms. The oppposite, i.e. fitting the model too close to the data, leads to modeling the random noise itself und thus results

in high variances. Moreover, it can be useful if we are able to restrict ourselves to a class of functions using methods of pre-analysis. Using the Python packages `arch` and `pmdarima` provides the user with a simple way of fitting autoregressive models. However, the following subsection includes an overview over the step-by-step fitting procedure that forms the basis for autoregressive models. Here, we focus on general ARIMA processes and the adaption of the procedure to GARCH processes is straightforward.

2.2.1 Fitting of ARIMA Models

The fitting procedure consists of several major steps that are illustrated in Figure 2.7 and will be explained in what follows.

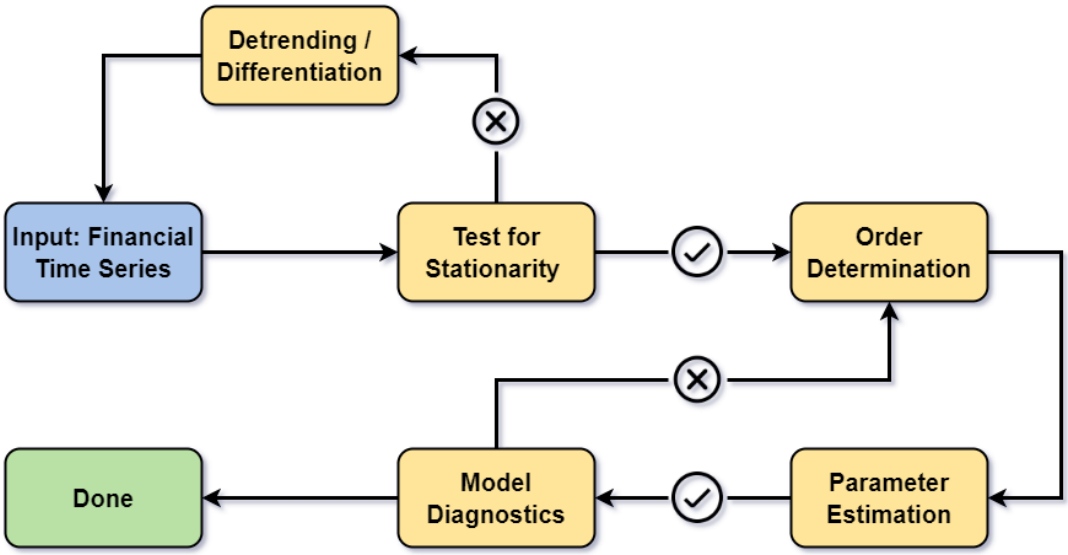


Figure 2.7: The scheme of a fitting procedure for autoregressive time series models. Ticks indicate that the model has passed a test or evaluation while crosses indicate failure. .

First, we need to prepare the financial time series and hand it over to the procedure. For complicated tasks, it is useful to exclude extreme values and fill possible gaps or NaNs in the data. Next, a test for stationarity of the input sequence X is performed and this can be done in two different ways by either manually checking the mean and variance of X over time or by statistical significance tests. Both methods can easily be implemented with Python. The **Augmented Dickey-Fuller (ADF) test** which was introduced in 1979, is a popular choice of an unit root test and verifies if a process is stationary or not ([45]). The Null hypothesis of the ADF test is that the characteristic polynomial exhibits at least the unit root for an autoregressive process, i.e. one of the

roots is equal to 1 and hence the data is non-stationary. If the Null hypothesis can be rejected at a certain confidence level, e.g. $\alpha = 0.05$, the alternative hypothesis that the data is stationary will be accepted. In Python, the package `statsmodels` offers an implementation of the ADF test.

If the result of the test for stationarity indicates that X is not stationary, some modifications have to be made. Detrending the data and finding the seasonal patterns is one option which is rarely possible but in some cases it takes care of the issue, e.g. for weather data. In this case, we switch to a SARIMA process. Alternatively, we can turn to differentiation and verify the necessary order of differentiation in an $ARIMA(p, d, q)$ model to achieve stationarity. The modified time series is then again tested for stationarity and the differentiation might has to be performed several times until stationarity is accomplished.

To then determine the remaining two orders p and q, we turn to the empirically estimated partial autocorrelation function (PACF) and the empirically estimated autocorrelation function (ACF) respectively. Both orders are determined by the largest significant lag, usually at a 95% confidence interval given by

$$\pm 1.96 \frac{1}{\sqrt{T}}. \tag{2.31}$$

Choosing the exact orders of the process X can be a subjective task which is why the **Akaike Information Criterion (AIC)** has been introduced to measure the quality of the fit and is well-known in statistics ([3], [153]). Let RSS denote the residual sum of squares of the fitted model, $k = p + q + 1$ the number of paramters and T the number of total observations. Then, the AIC is given by

$$AIC := T \ln\left(\frac{RSS}{T}\right) + 2k. \tag{2.32}$$

Sometimes, the AIC is defined differently in the literature as

$$AIC := \ln\left(\frac{RSS}{T}\right) + \frac{2k}{T}. \tag{2.33}$$

The overall approach is similar to regularization methods which are well known in numerical optimization such as *LASSO* penalization building on the ℓ_1-norm and *ridge* penalization which builds on the ℓ_2-norm ([117]). In contrast to these methods, the AIC is estimated post-hoc while LASSO and ridge regularization are often directly involved in the optimization process as we shall see in Section 3.4.2. The AIC is to be minimized over the order of the model and expresses the bias-variance trade-off while favoring models with fewer parameters. AIC allows to compare models with a different error distribution and avoids several well-known testing issues. However, it cannot be universally applied to models based on a different data set but the AIC is sufficient for our purposes. There exist numerous variants and extension of the AIC, for example **Takeuchi's Information**

Criteria (TIC) ([177]) or **Small Sample AICs** ([16]), each arguing to cure shortcomes of the original criterion.

The subsequent estimation of the parameters can be done by following a classical least-squares procedure, a standard maximum likelihood approach or a modified **Yule-Walker method** in combination with the estimation of spectral parameters suitable for autoregressive models ([58]). We do not delve deeper into the theory behind this approach, but refer to the literature mentioned above. Obviously, determining the order of the model and fitting the model parameters are not seperated in time but sometimes run in parallel. It is common to parallelize the fitting process and fit several model architectures at the same time using different CPU or GPU cores to speed up the training and compare the results later using the AIC or other criterions.

After the model has been selected and with the fitted parameters at hand, we evaluate the quality of the fit. The residuals of the fitted model should be pure white-noise, if not the model is underfitted, i.e. the orders have been chosen to low. To test for white-noise, we test for autocorrelation in the residuals. A class of tests to verify this property of the errors are **Portmanteau tests**. Here, the statistics of choice are the **Box-Pierce** and **Ljung-Box** statistics but there exist many other diagnostic tests. For the sample autocorrelations $\hat{\rho}_i$, the Null hypothesis is

$$H_0 : \hat{\rho}_1 = \cdots = \hat{\rho}_m = 0 \tag{2.34}$$

which is tested against the alternative hypothesis

$$H_a : \hat{\rho}_i = 0, \quad \text{for some } i \in \{1, \ldots, m\}. \tag{2.35}$$

The Box-Pierce test is based on the Portmanteau statistic

$$Q^*(m) := T \sum_{l=1}^{m} \hat{\rho}_l^2 \tag{2.36}$$

and the Ljung-Box test uses the statistic

$$Q(m) := T\,(T+2) \sum_{l=1}^{m} \frac{\hat{\rho}_l^2}{T-l}. \tag{2.37}$$

Both statistics follow a chi-squared distribution with m degrees of freedom. Moreover, the decision rules to reject H_0 at the significance level α are

$$Q^*(m) > \eta_\alpha^2 \quad \text{and} \quad Q(m) > \eta_\alpha^2, \tag{2.38}$$

where η_α^2 denotes the $100 \cdot (1-\alpha)$th percentile of the distribution.

If the test is passed, the fitting has been completed successfully and we can use the model to make predictions (Section 2.3). On the other hand, if the test is failed, the

Box-Jenkins procedure has to be repeated until the model no longer underfits the data and the diagnostic tests show no more autocorrelation in the residuals. Extensive theory on this type of statistical testing can be found in [111] and [26].

Again, it is common in practice to simlutaneously evaluate various candidate model on small subsets of data and to continue with the one indicating the best statistics.

For further details on fitting time series to autoregressive models, we refer to a standard book for financial econometrics written by Tsay ([180]).

2.3 Predictions

After fitting the model parameters with the Box-Jenkins approach, which allows us to verify the quality of fit for the given data, we would like to use the model for forecasting purposes. In financial markets, this can be helpful in risk assessment tasks and investment decision making. However, a sufficiently well-fitted model does not guarantee a strong predictive performance, also known as **generalization power**. Forecasting based on autoregressive models means taking the conditional expectation of the data. That is, if we want to make the prediction $\hat{Y}_{t+h}$ at h time steps ahead of the current time t with information Ω_t, for an $AR(p)$ model the forecast is given by

$$\hat{Y}_{t+h} = \mathbb{E}[Y_{t+h}|\Omega_t] = \sum_{i=1}^{p} \Phi_i \hat{Y}_{t+h-i}. \tag{2.39}$$

Obviously, $\hat{Y}_{t+h} = Y_{t+h}$ for $h \leq 0$ and $\mathbb{E}[\epsilon_{t+h}|\Omega_t] = 0$ for $h > 0$. Moreover, note that the unconditional and the conditional expectations of the residuals are not necessarily equal: $\mathbb{E}[\epsilon_{t+h}|\Omega_t] = \epsilon_t + h$ for $h \leq 0$, whereas $\mathbb{E}[\epsilon_{t+h}] = 0$ since the residuals are white-noise if the diagnostic test is passed.

To measure the quality of the forecast, the **Mean-Squared Error (MSE)** or the **Mean Absolute Error (MAE)** are commonly used. These measures are defined by

$$\text{MSE}(Y, Y^{\text{obs}}) := \frac{1}{H} \sum_{k=1}^{H} \left(\hat{Y}_{t+k} - Y_{t+k}^{\text{obs}}\right)^2 \tag{2.40}$$

and

$$\text{MAE}(Y, Y^{\text{obs}}) := \frac{1}{H} \sum_{k=1}^{H} |\hat{Y}_{t+k} - Y_{t+k}^{\text{obs}}|, \tag{2.41}$$

where H is the length of the forecasting period and $Y^{\text{obs}} = \left(Y_1^{\text{obs}}, \ldots, Y_H^{\text{obs}}\right)$ are the actually observed data, e.g. close prices.

The idea of forecasting can be extended to ARIMA and GARCH models in a similar manner. Moreover, forecasting in Python using the `pmdarima` and `arch` packages can be done by calling the `predict` and the `forecast` functions respectively.

A straightforward example of how ARMA models can be used to predict binary events instead of the exact prices can be found in Section 6.4.1 in [48].

2.4 Example

In this section, an example of using an ARIMA-GARCH model to forecast financial market data will be presented and compared to other basic methods for forecasting such as exponential smoothing.

Our goal is to generate five-days-ahead forecasts from this model. In the notebook `ARIMA_GARCH_Fit` which can be found in the attachment, we followed the procedure to fit autoregressive models described in Section 2.2 on the close prices of the S&P500 index. The file contains a detailed description of the procedure which fits the models with regard to the AIC.

Figure 2.8 shows the log returns of the index as well as a comparison of the forecasts generated by the fitted ARIMA(3,0,0)-GARCH(2,2) model with the actual validation data. Table 2.1 shows the fitted parameters. Note that the p-value for β_1 is very high, but we can accept this since the p-value for β_2 indicates significance of the second order. Making predictions with our model, the RMSE over the validation period is 0.017937 for a return series with average 0.000581. We compare the RMSE with the error of a primitive forecasting method which generates a forecast by simply taking the return of the previous day and get an RMSE of 0.0222. Thus, the ARIMA-GARCH model clearly outperforms this naive approach.

Table 2.1: The details of the fitted ARIMA-GARCH model.

ARIMA(3,0,0)		
Parameter	*Value*	*p-Value*
Φ_1	-0.1048	0.000
Φ_2	-0.0686	0.000
Φ_3	0.0271	0.003

GARCH(2,2)		
Parameter	*Value*	*p-Value*
α_0	0.0404	0.000
α_1	0.0802	0.015
α_2	0.1322	0.000
β_1	0.1342	0.657
β_2	0.6161	0.024

The primitive approach is equivalent to exponential smoothing with $\alpha = 1$. In the notebook we additionally implemented automated rolling forecasts based on exponential smoothing with both a fixed α over time and an optimized α for every forecast. Normally, forecasts generated by exponential smoothing methods are constant over time. Here, we updated the model in every step and only considered a limited amount of data from previous time steps. For step-by-step optimized α and a reference period consisting of the previous 30 days, the results are displayed in the bottom graph in Figure 2.8.

Moreover, we find that the RMSEs, which can be found in Figure 2.2, are lower than for the ARIMA-GARCH model while the fitting requires significantly less computational resources. As we will later see, smoothing in general is a nice tool to help us improve the generalization power of algorithm based forecasting for financial time series.

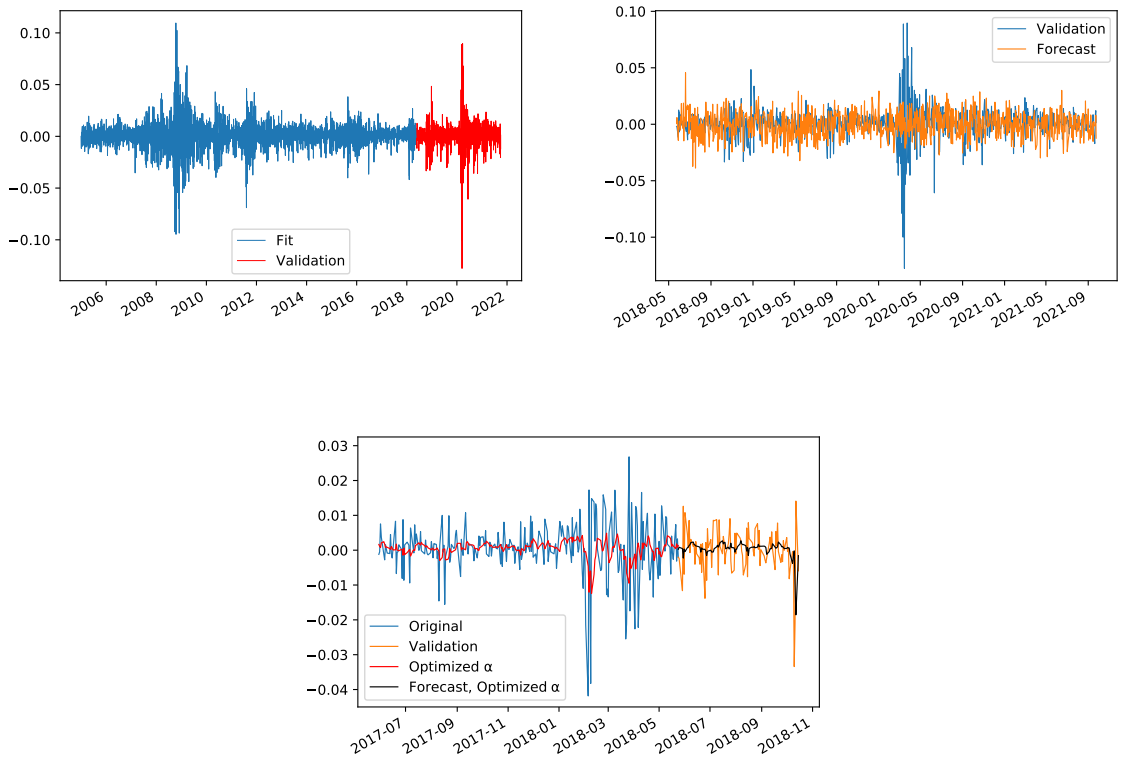


Figure 2.8: (Top left) Log returns for the S&P500 index and division into a fitting and a validation set. (Top right) Collected five-days-ahead forecasts of the fitted and weekly updated ARIMA(3,0,0)-GARCH(2,2) model versus validation data. (Bottom) Dynamic exponential smoothing forecasts with optimized α.

The results, in particular the RMSE in comparison to the average log return of the validation set, show that an ARIMA-GARCH model is not able to handle the complexity of market movements in a sufficient manner. Simple linear models, like the ones presented here, fail to capture the process and in general it is not recommended to use ARIMA-GARCH models for forecasting in particular for small-step forecasts, e.g. hourly prices. Furthermore, we found that we end up with a model with very different orders if we reduce or increase the input data. However, this model type allows us to detect a general trend and is easy to understand and analyze.

Table 2.2: Comparison of RMSE for different models.

	ARIMA-GARCH	**Exponential Smoothing**			
	Order $(3,0,0)$-$(2,2)$	$\alpha = 0$	$\alpha = 0.2$	$\alpha = 0.6$	dynamic α
RMSE	0.017937	0.0222	0.014913	0.017614	0.014527

2.5 Summary

In this chapter, we defined time series which are a sequence of observations of a stochastic process at discrete points of time. We introduced autocorrelation and partial autocorrelation as well as two forms of stationarity. The Wold representation theorem guaranteed us that a covariance stationary time series can be written as a weighted sum of a deterministic and a weighted stochastic time series. Afterwards, we looked at processes consisting of a weighted sum of past observations and weighted averages of past noises called autoregressive moving-average process (ARMA) and investigated its properties such as causality and invertibility. A different type of autoregressive models called Generalized Autoregressive Conditionally Heteroscedasticy (GARCH) processes capture the conditional variance over time. We then turned to a smoothing method (exponential smoothing) and an approach to reduce the potentially high dimensionality of the process (Principal Component Analysis (PCA)).

The second section dealt with how autoregressive models, for example an ARIMA-GARCH model, can be fitted to data. The basic procedure is called Box-Jenkins approach. After guaranteeing that the process at hand is stationary, the order is determined and the parameters are estimated. The quality of the model is then verified using diagnostic tests, e.g. Ljung-Box statistics.

Generating forecasts with a fitted autoregressive model is straightforward: the prediction is created by taking the conditional expectation of the data. The quality of the predictions is evaluated by measures, for example the mean-squared error (MSE). It is worth mentioning that a model which fits well on the inital data set does not necessarily guarantee for a strong predictive performance.

An example of forecasting financial market data showed that the ARIMA-GARCH model is not performing very well and is only able to discover long-term trends. Therefore, we seek for a more powerful alternative and exponential smoothing hints that non-linear approaches providing us with a memory of the process can be useful and more accurate.

This chapter introduced the basic econometric theory that will be used in the following. In the next chapter, the basic concepts of machine learning will be introduced and we will furthermore introduce methods in Chapter 4 which can be regarded as generalizations of autoregressive models.

3 Introduction to Machine Learning

In 1955, a young American Assistant Professor of Mathematics and his fellow research colleagues submitted a proposal for a research project on **artificial intelligence (AI)** at Darmouth College, New Hampshire, which was held in the summer of 1956. Their ambitious goal is summarized in the following quotation ([115]):

> *The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves.*

Enormous progress has been made since and machines are able to outperform the human brain in single tasks in very specific frames. Examples that have been in the centre of public attention are the defeats of reigning chess world champion Garry Kasparov by the IBM computer Deep Blue in 1996 ([68]) and multiple Go world champion Lee Sedol by Google's AlphaGo computer in 2016 ([69]).

The study of computer algorithms that use data and improve through experience without being explicitly programmed is called **machine learning (ML)**. It is a field of AI and uses training data (sample) that can be used to provide predicitions or forecasts based on a particular input. The field of applications is vast and often connected to different forms of time series such as voice recognition or financial time series, but also to tasks like recognition of handwriting or spam filtering. Generally speaking, ML algorithms are particularly suitable for problems that usually require a long list of rules, complex tasks without a satisfactory traditional solution, problems that have to be updated constantly and the analysis of large amounts of data. While the focus of this thesis will be on theory which is particulary useful in the context of financial mathematics, the concept of machine learning is well-known in most sciences and it has stimulated interdisciplinary exchange ever since its introduction. As an example in linguistics, we refer to Schneider ([155]) who introduced an algorithm to allow a machine to build lexical knowledge from noisy texts.

In this chapter, which combines the theory presented in Chapter 1 in [48], Chapter 1 in [60] and Chapter 8 in [41], a brief introduction to the ideas and concepts of Machine Learning will be given followed by an overview over the different approaches. After presenting the challenges of ML in practice, the chapter will be concluded by a section focusing on the mathematical background of the training process in both, a non-probabilistic

and a probabilistic environment.

3.1 Motivation and Fundamental Idea

In 2007 and 2008, the global financial crisis revealed fundamental weaknesses in the control mechanisms of the financial industries. Since then, regulation turned to a more data-based approach that also takes into account alternative and previously unused data out of the usual scope of the industry such as data retrieved from social media and news articles. Several of the most well-known data providers offer processed data that can be used as inputs for trading algorithms. This new type of data is often high-dimensional and not strictly categorical. Thus, classical econometric models based on linear algebra tend to fail on such datasets since the dimension of the variables exceeds the number of observations. In contrast, machine learning models are capable to detect hidden patterns and structures in this high-dimensional setting, as we shall see later in this thesis. In this work, we will restrict ourselves to simpler input data: the observations of the prices at previous points of time. The data sets could be extended by news sentiment indices, trading volume, commodities, exchange rates or cryptocurrencies if all observation points are in line which is often not the case.

Critics of the method itself focus on the black-box allegation, i.e. that the exact processes of ML are not observable. On the other hand, researchers as Lopez de Prado argue that advanced methods allow for a more detailed understanding and thus the black-box view has to be dismissed ([112]).

Nowadays, a whole *fintech industry* deals with technology-based developments in the financial sector and is growing rapidly. Examples are cryptocurrencies, deep learning to detect fraud or investment advisory by bots.

Mathematically speaking, machine learning can be regarded as the construction of a mapping

$$F : X \to Y, F(x) = y, \tag{3.1}$$

for $x \in X \subseteq \mathbb{R}^{n_x}$ and $y \in Y \subseteq \mathbb{R}^{n_y}$, which converts input data x into an output y where the structure of the map is explored by processing the observations through the algorithm and evaluations of the data (**training)**. Both x and y can potentially be multidimensional, for example x a vector whose elements x_i are increments of a time series, e.g. the observations of the past n_x close prices, and y the close price of the current day, which implies $n_y = 1$. In other words, machine learning aims to extrapolate an unknown function F in order to map the output y, which can be both continuous or discrete. It is very useful to have the data ready in an appropriate vector or matrix representation before the exploration of the map starts.

The fitted map F is often referred to as the **predictor** which produces an output given a particular set of input information. There are two major approaches of modeling:

models as pure function and **models as probability distributions**. The former one is non-probabilistic and does thus not take directly into account that observations might be noisy. An example for this approach are the classes of linear or polynomial functions including functions of the form

$$f(x) = \alpha x + \alpha_0 \quad \text{or} \quad f(x) = \sum_{i=1}^{n_p} \beta_i x^i + \beta_0 \tag{3.2}$$

for $x \in \mathbb{R}^{m_x}$, $\alpha_0 \in \mathbb{R}$, $\alpha \in \mathbb{R}^{m_x - 1}$, $\beta_0 \in \mathbb{R}$, and $\beta \in \mathbb{R}_p^n$, where the concept of **empirical risk estimation** is used to train the model. The exact procedure will be introduced in Section 3.4.1.

On the other hand, introducing models based on probability distributions allows for the consideration and sometimes the filtration of noise and uncertainty. It is often assumed that the observations do not strictly follow a function, but that small independent and identically distributed (iid) errors ϵ with the same dimension as the output y are added. Gaußian errors can be found in many models, i.e. ϵ is white-noise following a standard normal distribution $\mathcal{N}(0, 1)$. Therefore it is assumed that the true observations $\hat{y}$ can be modeled by

$$\hat{y} = F(x) + \epsilon. \tag{3.3}$$

Instead of considering the predictor as a specific function like above, one considers the predictor to be a probabilistic model describing the distribution of possible functions. Instead of returning a single value for a given input, the predictor would return probability distribution for the output value. The search for the most suitable distribution is carried out by means of **maximum likelihood estimation (MLE)** and **probabilistic modeling**. These concepts as well as empirical risk estimation will be introduced in Section 3.4.

It is common to continuously update several models or maps F to prevent relying on a model that runs into a data silo effect only effective for a single market view.

3.2 Different Approaches to Machine Learning

Machine learning algorithms can be classified according to different criteria. In this section, classifications based on the aspect of supervision, the ability to learn incrementally from a stream of data and the way of generalizing will be presented. The former approach is the most common one to distinguish between several types of algorithms and shall therefore be introduced first.

3.2.1 Supervised, Unsupervised and Reinforcement Learning

Depending on the type and extent of supervision an algorithm receives during its training phase, a classification into three different categories can be made:

- *Supervised learning*,
- *Unsupervised learning*, and
- *Reinforcement learning*.

So called **supervised learning (SL)** is based on labeled data

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n), \tag{3.4}$$

i.e. the training data includes the desired solutions. The goal is to find a mapping that captures the relationship between the **features** or **attributes** $x_1, x_2, \ldots, x_n \in X$ and the **labels** or **responses** $y_1, y_2, \ldots, y_n \in Y$. An example for a response is a financial time series and in this case features could be other financial time series such as previous close prices, news sentiment data or data from other markets. Other typical tasks for SL are classifications such as detecting spam mails in your e-mail inbox or the prediction of house prices based on factors such as neighbourhood, age, interior and so on.

To minimize the error of the mapping F, the algorithm needs to be trained. The supervision aspect is that the algorithm or mapping can be tested and the output shows if the mapping is good enough or not by comparing the generated output with the one observed in reality. In other words, the algorithm receives feedback by a **teacher** who checks the results. The teacher can be a performance measure like the MSE presented above.

Supervised learning can be divided into two model types: **discriminative** and **generative models**. The former class learns the probability of an output given an input i.e. learns the conditional distribution of the outputs, while the latter one learns the joint distribution of the input and the output.

Important examples of SL algorithms are:

- Linear regression,
- k-Nearest Neighbors ([8]),
- Support Vector Machines ([51]),
- Decision trees ([119]), and
- Neural networks (Section 4).

The focus in this thesis will be on supervised learning and in particular on neural networks.

Unsupervised learning (UL) is characterized through the fact that the data is unlabled and is not given in pairs or other groupings:

$$Y = \{y_1, y_2, \ldots, y_n\}. \tag{3.5}$$

The goal of unsupervised learning is to analyze the data for hidden patterns and possible grouping constellations of similar observations. This approach is often referred to as *data*

mining using cluster analysis algorithms and in some cases hidden Markov models or hierarchies can be detected. Examples for UL algorithms are clustering for the detection of connections or groups, anomaly or novelty detection, dimensionality reduction to simplify the data or association rule learning to identify relations between certain attributes.

The third category of machine learning tasks is called **reinforcement learning (RL)**. This sort of algorithm aims to find the optimal sequence of actions or strategy in a dynamic environment. The learning system, called an **agent**, gets a **reward** for its actions. Mathematically speaking the goal is to achieve Bellman optimality of a Markov decision process and thus focusing on cumulative rewards. This requires an implementation of the algorithms in a dynamical programming environment.

Applications for reinforcement learning are, among others, optimal trade executions and portfolio allocation within a given time horizon. More detailed theory and examples can be found in Chapters 9 and 10 in [48]. Furthermore, the program on the computer AlphaGo mentioned above is based on SL and applied the startegy learned by analyzing millions of Go matches to beat the former world champion.

3.2.2 Batch and Online Learning

Another classification of Machine Learning algorithms with regard to the ability to learning from increments of continuously streaming input data is possible.

If the algorithm is not able to do so, it follows a **batch learning** approach. It has to be trained using all the data available and not just parts or chunks of it. This might require a lot of computational resources and time if the system has to be trained from scratch very often. In addition, the algorithm only applies what it has learned and does not update its strategy during the process. We refer to this as **offline learning**. If the user of the algorithm wants to include new data, a new versions of the system has to be trained before replacing the old algorithm with the new one. For rapidly changing data, this approach might be too slow to deliver satisfactory results in applications.

The solution to this problem is **online learning**, where algorithms are able to deal with sequentially fed data. Updating the system on a regular basis with smaller sets of new data is fast and requires little computation capacity. The **learning rate** parameter indicates how fast the algorithm should adapt to new data. Thus, a high learning rate means that the systems tends to quickly forget about old data and the trade-off between long- and short-term memory has to be taken into consideration. The issue will be the subject of further reflections when discussing *Long Short-Term Memory Networks (LSTM)* in Section 5.2.

Online learning is used if the computational capacities are too limited to train a large general mode or the model needs to adapt to changes quickly.

3.2.3 Instance-Based and Model-Based Learning

Machine learning algorithms can also be categorized by their way of generalizing. Most of the time, users select ML algorithms to make good predictions based on new cases and thus generalize observations made before. Of course, a high accuracy on the training data is needed, but the real challenge is to perform well for new instances.

Instance-based learning consists of learning a task by heart and then generalizing the system to detect not just identical but also similar data patterns. For example, if a ML approach is used to detect spam emails, the algorithm learns the exact content of collected spam mails that serve as the training data by heart and then flags incoming new mails as a spam if they have a certain amount of words in common with the training spam emails. The *measure of similarity* specifies the degree of similarity that is needed to flag the new incoming email.

A different approach, called **model-based learning**, is to introduce a model equation and fit its parameters using the training data set. For example, the assumption that the observations follow a linear model,

$$y_i = \alpha_0 + \alpha_1 x_i, \tag{3.6}$$

can be made and the parameters α_0, α_1 can be determined by applying numerical methods such as linear regression which minimize a loss or cost function indicating the deviation of the model from the actual data. If the explanatory power of the model equation is not satisfactory, a different model equation has to be chosen or more attributes have to be included.

3.3 Challenges of Machine Learning

The task for the implementer of a machine learning algorithm consists of selecting the data and the learning algorithm. A bad selection of one of them will most likely lead to a result that is not representative and thus to incorrect conclusions and forecasts.

The data used to train the algorithm should be of sufficient quantity, of high quality - which might require pre-processing of the data set - and data that is not representative for the generalization should be excluded from the training data set to avoid noises. Some researchers, such as Norvig et al. ([71]), state that for complex problems the quantity and qualtity of data matters more than the selection of the algorithm. To date, however, most data sets are small- or medium-sized and thus, effective algorithms continue to be of greatest interest. In addition, only features that are relevant should be handed to the algorithm. The process of selecting the features is called **feature engineering**.

If the data set is prepared and well suited to the task, one has to make sure that the models are suitable for the task. Allowing our model equation to have too many degrees of

freedom, for example if the equation is a high dimensional polynomial, the algorithm would be prone to **overfitting**. Overfitting means that the performance on the training data is excellent but the generalization to new instances shows unlikely patterns or results. Often, too many degrees of freedom or a too few data cause the model to detect patterns in the noise of the training data that are not existing or patterns that are nonrepresentative or irrelevant. To cure these unwanted effects, a simplification of the model by allowing for fewer parameters or restricting them (**regularization)** is as helpful as the reduction of noise in the training data set. The regularization is mathematically formalized by the hyperparameters of the learning algorithm. A high valued hyperparameter prevents overfitting but makes a good solution with a small error unlikely.

On the other hand, the opposite effect, called **underfitting**, occurs if the model equation is too simple to detect and capture the relevant structure. A solution of the issues can be achieved by selecting a model with more parameters, widening the parameter constraints or reviewing the feature engineering process.

In practice, the data used is usually split into a training and a test data set. After fitting the model parameters using the training data set, the model is evaluated with the test data. If the loss measured by e.g. MSE or the error rate of the test is significantly higher than the low error of the training data, this indicates overfitting of the training data and the model should be simplified. Moreover, simultaneously training models with different hyperparameters allows for the selection of the best fit. It is common to evaluate candidate models on various small sets of the data. By averaging the evaluation, the choice of the best model is even more accurate. This method is called **cross-validation**.

Obviously, the process of selecting a suitable model can be challenging and time-consuming. The generalization error is usually expressed as the sum of three types of errors: **bias**, **variance** and **irreducible error**. The bias is rooted in wrong assumptions such as selecting a linear model when a quadratic model would suit the task better while the simpler model is prone to underfitting. On the other hand, if the model is too sensitive to small variations in the training data, for example when offering many degrees of freedom, a high variance can be observed indicating overfitting. The third type of error, the irreducible error, is based on the data's noisiness itself and can only be handeled by improving the quality of data or cleaning up the data. Reducing a model's complexity and degrees of freedom increases the bias but reduces the variance, while increased complexity reveals the opposite effect. This challenge is referred to as the **bias-variance tradeoff** which has already been touched upon above.

It is important to note, that in general and without making any assumptions on the data, it is not possible to a priori select a model that is working better for the task than others. This result was presented first by David Wolpert in a much-noticed paper in 1996 as the **No Free Lunch Theorem** ([188]). As a consequence, there is no other way to find the best model than to evaluate them all.

3.4 Training of Machine Learning Models

In this section, we will turn to the mathematics of training a machine learning model. The theory builds in some parts on basic numerical analysis theory and we refer to a book written by Stoer and Bulirsch for a detailed introduction to this topic ([171]).

Building and applying a machine learning algorithm usually consists of three phases:

1. Model selection and selection of hyperparameters,
2. Training the model, and
3. Gnereations of predictions using the trained model.

The selection of the model type has a huge impact on the second step, which is why the training will be considered separately for models that are pure deterministic functions (Subsection 3.4.1) and probabilistic models (Subsection 3.4.3). By hyperparameters model-defining characteristics such as the degree of the polynomial or the range of the parameters are referred to. The third step will be subject of the concluding section of this chapter.

The term **training of the model** has already been used several times. Training means nothing other than the determination of the model parameters or distributions either using a direct method or - which is more often the case - an iterative method. In mathematical terms, one looks for best set of parameters $\alpha = (\alpha_0, \ldots, \alpha_m)$ out of the parameter space $\mathcal{A} \subseteq \mathbb{R}^m$, where we use the shorthand notation $\bar{\alpha} := (\alpha_1, \ldots, \alpha_m)$. This section will provide an overview over the basic mathematical concepts to train a machine learning model focusing on supervised learning. The main challenge will be to find a balance between fitting the data well and having a simple explanantion. Following Chapter 4 in [60] and Chapter 8 in [41], the focus will be on different relatively simple regression models and iterative optimization in form of gradient descents. More advanced techniques that are frequently used in practice when training neural networks are presented in Section 4.2.

To understand the theory of training a ML algorithm, it is recommended to have a good understanding of linear algebra and also some background in optimization theory as well as in probability theory. A sufficient overview over the necessary theoretic concepts can be found in Chapters 1 to 7 in [41].

Before we continue, let us fix some notation. In practice, one splits the available data $D = (X, Y)$ into a training set

$$D_{\text{train}} := (X_{\text{train}}, Y_{\text{train}}) := (X_{i,:}, Y_{i,:})_{i=1,\ldots,N} \tag{3.7}$$

and a test set

$$D_{\text{test}} := (X_{\text{test}}, Y_{\text{test}}) := (X_{i,:}, Y_{i,:}.)_{i=N+1,\ldots,n_x}\,, \tag{3.8}$$

where $N < n_x$ denotes the cutoff for how much of the labeled data is included in the training set. It is common to use about 75%, 80% or 85% of the data for the training,

i.e. $N \approx 0.75n_x$, $N \approx 0.8n_x$ or $N \approx 0.85n_x$ but this can vary depending on the amount of data available. The test set is used to verify if the model performs well on unseen data and not only on known data, thus evaluating the generalization performance. Moreover, the predictor F we are looking for is formally defined as a map that takes a vector as its input and produces a vector or scalar as the output. However, it is also possible that the input consists of multiple vectors, a matrix or a tensor, e.g. if multivariate time series are handed over to the model. In the case of a data set consisting of feature vectors $x_i = (X)_i$ and labels $y_i = (Y)_i$, the mapping F should be understood elementwise, i.e.

$$F(X) := (F(x_i))_i \,. \tag{3.9}$$

This can easily be extended to tensors and other multi-dimensional data and the pair (x_i, y_i) should be interpreted as the i-th instance of the data set D.

We are now in a position to proceed to the theory of training non-probabilstic models in the next section which is based on Chapter 4 in [60] and Section 8.2 in [41].

3.4.1 Non-Probabilistic Models

The training of models of pure determinstic functions is based on the principle of empirical risk minimization. This concept became popular in the context of support vector machines (SVM), which are solving binary classification tasks in supervised learning. The basic theory on SVMs can be found in Chapter 12 of [41]. However, empirical risk minimization is useful in a wide range of applications and is fairly general.

The goal continues to be finding a parameter set α^* which defines a function or predictor $F(\cdot, \alpha^*)$ which is an element of a parametrized class of functions $\mathcal{F}$ which approximates best the relation between the elements of the pairs $(x_1, y_1), \ldots, (x_n, y_n)$ of a supervised learning task via

$$F(x_i, \alpha) \approx y_i, \quad \text{for all } i = 1, \ldots, n. \tag{3.10}$$

An example for the class of functions $\mathcal{F}$ is the class of linear functions given by

$$\mathcal{F} := \{F : \mathbb{R}^{m_x} \times \mathbb{R}^{m_x+1} \longrightarrow \mathbb{R} | F(x_i, \alpha) := \bar{\alpha}^T x_i + \alpha_0\} \tag{3.11}$$

and parametrized by the set of coefficients $\alpha \in \mathbb{R}^{m_x+1}$.

To quantify and measure the qualtity of the fit of a predictor resulting from the deterministic approach, *loss functions* are introduced.

Definition 3.1 (Loss Function)**.** *A* **loss** *or* **error function** $\ell(Y, \hat{Y})$ *is a function that takes labels Y that are part of the actual data and predictions $\hat{Y}$ generated by a predictor F as its input and returns a non-negative scalar that represents the error made in this particular prediction. Thus, $\hat{Y} := F(X)$ and $\ell(Y, \hat{Y}) = \ell(Y, F(X))$ for features X.*
The average over the losses of a sample, defined by

$$\frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, \hat{Y}_i) = \frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, F(X_i)), \tag{3.12}$$

is called **averaged loss function**.

The goal is to minimize the average loss on the sample set by finding good parameters α^*. In mathematical terms, one has to solve the optimization problem

$$\min_{\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n_\alpha}} \frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, \hat{Y}_i) = \frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, F(X_i)), \tag{3.13}$$

where we averaged the loss over all labels and corresponding predictions. This strategy is called **empirical risk minimization**.

Remark 3.2. *Note that loss functions are used for both, the training process and thus on the training data set as well as for the measurement of the performance on new data sets when evaluating the predictor on a test sample. In some applications the loss function used for the training can be different from the one used for testing and evaluating. Often, the performance measure for the model evaluation is the more sensible one.*

Frequently used averaged loss functions are the *root-mean-squared error* and *mean absolute error* functions.

Definition 3.3 (Common Averaged Loss Functions)**.** *Let X, Y be real-valued matrices of dimension $N \times m_x$ and $N \times m_y$ respectively and F a predictor as defined above.*

The **mean-squared error (MSE)** *is defined by*

$$MSE(X, Y, F) := \frac{1}{N} \sum_{i=1}^{N} \left(F(x_i) - y_i\right)^2. \tag{3.14}$$

The **root-mean-squared error (RMSE)** *is defined by*

$$RMSE(X, Y, F) := \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(F(x_i) - y_i\right)^2} \tag{3.15}$$

and corresponds to the Euclidian or ℓ_2-norm.

The **mean absolute error (MAE)**

$$MAE(X, Y, F) := \frac{1}{N} \sum_{i=1}^{N} |F(X_i) - Y_i| \tag{3.16}$$

corresponds to the ℓ_1-norm.

By definition, the RMSE is more sensitive to outliers or extreme values than the MAE, while MSE is the most sensitive of them. The advantage of using RMSE over MSE is

that the result is in a similar scale and has the same unit as the labels, e.g. \$, whereas MSE would result in squared units, e.g. $\$^2$. Furthermore, it is important to note that both the MSE and RMSE are convex in the input set of parameters determining the model F. In the context of a convex function, nonlinear optimization theory provides us with nice necessary and sufficient conditions for optimality. The interested reader is referred to [27] for more extensive mathematical theory on this topic.

Using loss functions like RMSE and MAE allows us to minimize the empirical risk, but as mentioned above the actual aim is to achieve good performance for unknown test data D_{test}, i.e. a good generalization of the predictor. According to Mitchell ([118]), empirical risk minimization can lead to overfitting, and thus does not generalize well to unknown data in general. To prevent this, *regularization* by penalizing the model parameters is introduced, making it harder for the optimizer to return a flexible and complex predictor and easier to find a balance between the accuracy and complexity of the solution (bias-variance tradeoff).

Definition 3.4 (Regularization)**.** *Let ℓ be an averaged loss function and a penalty term $P : [0, \infty) \times \mathbb{R}^{m_x+1} \longrightarrow [0, \infty)$ a function of a parameter $\lambda > 0$ and the model parameters α. Then*

$$\min_{\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n_\alpha}} \frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, \hat{Y}_i) + P(\lambda, \alpha) = \frac{1}{N} \sum_{i=1}^{N} \ell(Y_i, F(X_i)) + P(\lambda, \alpha) \tag{3.17}$$

is called the **(averaged) regularized risk minimization problem**. *The positive parameter λ is called* **regularization parameter** *and the term $P(\lambda, \alpha)$ the* **regularization term**.

Popular examples of regularization are given in the following example.

Example 3.5. *If the MSE is used and the regularization term P is defined by*

$$P(\lambda, \alpha) := \lambda \|\alpha\|_2^2 = \lambda \sum_{i=0}^{n_\alpha} \alpha_i^2, \tag{3.18}$$

we talk about **Tikhonov regularization** *based on the square of the ℓ_2-norm.*
Sticking to the MSE, but using the ℓ_1-norm instead of the squared ℓ_2-norm, the **ℓ_1-regularization** *is given by the regularization term*

$$P(\lambda, \alpha) := \lambda \|\alpha\|_1 = \lambda \sum_{i=0}^{n_\alpha} |\alpha_i|. \tag{3.19}$$

A combination of both approaches is called **elastic net regularization** *where a mix ratio r defines the mixture of Tikhonov and ℓ_1 regularization. For this, the definition P is as follows:*

$$P(\lambda, \alpha) := r\lambda \|\alpha\|_2^2 + \frac{1-r}{2} \lambda \|\alpha\|_1 = r\lambda \sum_{i=0}^{n_\alpha} \alpha_i^2 + \frac{1-r}{2} \sum_{i=0}^{n_\alpha} |\alpha_i|. \tag{3.20}$$

If $r = 0$, the elastic net regularization is equivalent to the Tikhonov and for $r = 1$ it is equivalent to the ℓ_1 regularization.

As mentioned above, in practice it is recommended to use a partition of the data available to validate the predictor resulting from processing the training data D_{train}. The validation or test data D_{test} can potentially be small due to the limited amount of data which are available for many tasks. Using a small set of test data can lead to an estimate with variance. To tackle this, the concept of **cross-validation** has been introduced. In cross-validation, the whole data set available is separated in K pairwise disjoint chunks. $K-1$ of the chunks serve as training data while the remaining data is regarded as the test data set. By training the model with all possible combinations of training sets and test set, the process results in K different models $f^{(k)}$ for $k = 1, \ldots, K$. For each model, the generalization is evaluated using the test set and a performance measure like RMSE. Taking the average over all of the performance measure, cross-validation allows for a better approximation of the expected generalization error than when only evaluating one model. Of course, fitting K models requires more computational resources, but the task can easily be parallelized and solved using CPU cores, university or company servers or cloud computing. This method proves to be particularly useful because most optimizers used in practice are stochastic and thus the results for the same differ in each run to a certain extend.

Figure 3.1 shows an example for the data separation of a cross-validation with $K = 5$.

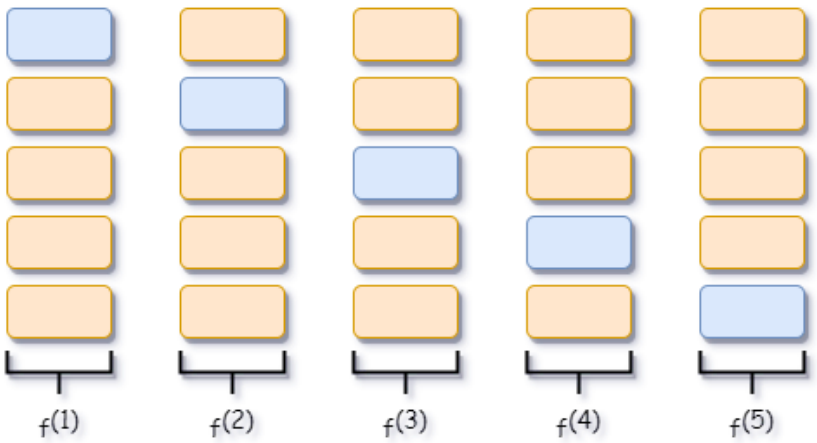


Figure 3.1: Illustration of cross-validation for $K = 5$. Orange boxes indicate training data sets and blue boxes test data sets. Figure inspired by Figure 8.4 in [41].

3.4.2 Optimization Techniques for Non-Probabilistic Models

So far, we introduced examples of classes of functions for the predictor F and examples for performance measures heavily used in practice. We have extended the concept by allowing for a regularization term, but we have not presented mathematical techniques to actually perform the optimization of the minimization problems. Let us first consider a simple linear model without regularization. The task is to find the parameters $\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n_x+1}$ such that the predictor

$$F_\alpha(x) = \alpha_0 + \alpha^T x_i = \alpha_0 + \alpha_1 x_{i,1} + \cdots + \alpha_{n_x} x_{i,n_x}, \tag{3.21}$$

where n_x describes the number of features and x_i is the feature of the i^{th} instance of a data set. This model is known as the **linear regression model**. Usually, one would use the RMSE as the performance measure, but finding a minimum for the RMSE is equivalent to finding a minimum for the MSE which is easier to handle. Recall that the MSE is defined by Equation 3.14 and the empirical risk minimization problem is given by

$$\min_{\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n\alpha}} \frac{1}{N} \sum_{i=1}^{N} \left(\alpha^T x_i - y_i\right)^2, \tag{3.22}$$

where N is the length of the training data set, (x_i, y_i) is the i^{th} instance of the training data set $D_{\text{train}} = (X_{\text{train}}, Y_{\text{train}})$ and $n_\alpha := n_x + 1$.

If we assume that Y_{train}, i.e. the labels of the training data are vector shaped, we can derive the following results concerning the solvability of the optimization problem which also includes a closed-form solution.

Lemma 3.6 (Analytical Solution for Linear Regression)**.** *If the empirical risk minimization problem for the class of linear function is given by Equation 3.22 and the matrix $X_{train}^T X_{train}$ is invertible or equivalently if X_{train} is of full rank, a closed-form solution is given by*

$$\alpha^* = \left(X_{train}^T X_{train}\right)^{-1} X_{train}^T Y_{train}. \tag{3.23}$$

Moreover, the optimality is guaranteed if $X^T X$ is positive definite.

Proof. The optimal α^* for (3.22) is identical with the optimal value for the problem

$$\begin{aligned} &\min_{\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n\alpha}} \frac{1}{2} \sum_{i=1}^{N} \left(\alpha^T X_{\text{train}_i} - Y_{\text{train}_i}\right)^2 \\ \Leftrightarrow &\min_{\alpha \in \mathcal{A} \subseteq \mathbb{R}^{n\alpha}} \frac{1}{2} \left(\alpha^T X_{\text{train}} - Y_{\text{train}}\right)^T \left(\alpha^T X_{\text{train}} - Y_{\text{train}}\right), \end{aligned} \tag{3.24}$$

where the sum is represented in a more compact form using matrix products. For notational reason we write $\phi(\alpha)$ for the term that has to be optimized and shorten our notation by dropping the tag train for X and Y. Moreover, we will write y instead of Y

for the label vector. From basic optimization theory, e.g. [1], it is well known that to find the optimum we need the derivative with regard to α:

$$\begin{aligned}\frac{\partial \phi}{\partial \alpha} &= \frac{1}{2}\frac{d}{d\alpha}\left(\alpha^T X^T X\alpha - 2y^T X\alpha + y^T y\right) \\ &= \left(\alpha^T X^T X - y^T X\right).\end{aligned} \tag{3.25}$$

Setting this equal to zero leads to

$$\begin{aligned}\frac{\partial \phi}{\partial \alpha} = 0 &\Leftrightarrow \left(\alpha^T X^T X - y^T X\right) = 0 \\ &\Leftrightarrow \alpha^T X^T X = y^T X \\ &\Leftrightarrow \alpha^T = y^T X\left(X^T X\right)^{-1} \\ &\Leftrightarrow \alpha = \left(X^T X\right)^{-1} X^T y.\end{aligned} \tag{3.26}$$

Thus, the optimal solution is given by the last equation, which is called **normal equation** since the Hessian is given by

$$\frac{\partial \phi}{\partial \alpha^2} = X^T X \tag{3.27}$$

and thus positive definite by assumption. □

In applications, the data sets can be huge and therefore the computation costly, in particular because of the need to compute the inverse $\left(X^T X\right)^{-1}$. One approach to reduce the computational effort is to use standard matrix factorization techniques, for example **singular value decomposition** ([170]). This approach is more efficient, but not suitable for some data sets and limited memory, which is why we turn to iterative methods using algorithmic optimization techniques.

Imagine being lost on a hike in the mountains after dense fog appeared. Having lost your orientation, you could go downhill following the deepest slope from your position for several steps before re-evaluating the direction you are heading and finally return to the valley once there is no slope to get lower anymore. In the context of mathematics, the direction with the sleepest slope is given by the negative *gradient* of the loss function.

Definition 3.7 (Partial Derivative and Gradient)**.** *For a function $f : \mathbb{R}^n \to \mathbb{R}$ and the input $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, the* **partial derivatives** $\frac{\partial f}{\partial x_i}$ $(i = 1, \ldots, n)$*, are defined as*

$$\begin{aligned}\frac{\partial f}{\partial x_1} &:= \lim_{h\to 0}\frac{f(x_1 + h, x_2, \ldots, x_n) - f(x)}{h} \\ &\vdots \\ \frac{\partial f}{\partial x_n} &:= \lim_{h\to 0}\frac{f(x_1, \ldots, x_{n-1}, x_n + h) - f(x)}{h}.\end{aligned} \tag{3.28}$$

The collection of the partial derivatives in a row vector

$$\nabla_x f = grad\ f = \frac{df}{dx} = \left[\frac{\partial f(x)}{\partial x_1} \cdots \frac{\partial f(x)}{\partial x_n}\right] \tag{3.29}$$

is called the **gradient** *or the* **Jacobian** *of f.*

If f is a convex function, optimization theory provides us with the following result.

Theorem 3.8. *Let $f : \mathbb{R}^n \supseteq \mathcal{D} \to \mathbb{R}$ be a convex function, i.e.*

$$f((1-\lambda)x + \lambda y) \leq (1-\lambda)f(x) + \lambda f(y), \quad \forall x, y \in \mathcal{D} \text{ and } \lambda \in [0,1], \tag{3.30}$$

and $\mathcal{F}$ a convex set, i.e.

$$(1-\lambda)x + \lambda y \in \mathcal{F}, \quad \forall x, y \in \mathcal{D} \text{ and } \lambda \in [0,1]. \tag{3.31}$$

Then there holds:

1. *If $x^* \in \mathcal{D}$ is a local minimum of f, i.e.*

$$f(x) \geq f(x^*) \quad \forall x \in \mathcal{D} \cap \{x \in \mathbb{R}^n : \|x - x^*\|_2 < \epsilon\} \tag{3.32}$$

   for some $\epsilon > 0$, then x^* is a global minimum of f on $\mathcal{D}$, i.e. Equation 3.32 holds for all $x \in \mathcal{D}$ instead of only for all $x \in \mathcal{D} \cap \{x \in \mathbb{R}^n : \|x - x^*\|_2 < \epsilon\}$.

2. *If $\mathcal{D}$ is an open set, $f \in C(\mathcal{D}, \mathbb{R})$, i.e. f is continuously differentiable on $\mathcal{D}$ and $x^* \in \mathcal{D}$ a stationary point, i.e.*

$$\nabla f(x^*) = 0, \tag{3.33}$$

   then x^* is a global minimum.

Proof. More detailed theory and the proof for a similar result can be found in [44]. □

We apply this to a loss or cost function ℓ parametrized by a set of parameters α to find its minimum over $\alpha \in \mathcal{A}$ using the straightforward Algorithm 1, the **gradient descent for unconstrained optimization (GD)**. Here, we write ℓ_α to point out that this function is depending on the choice of α when using the model as pure deterministic functions approach.

The algorithm attempts to decrease the loss ℓ step by step and to converge to a minimum with $\nabla \ell_{\alpha^{(i)}} = 0$. A small deviation is allowed with tolerance ϵ to account for numerical inaccuracies.

In Line 4 a **step size algorithm** is mentioned. The step size $\gamma^{(i)}$, in the context of machine learning often referred to as the *learning rate*, is essential for the performance of the algorithm. If the rate is too small, it is likely that many iterations, and thus a lot of time, are needed before convergence can be observed (left of Figure 3.2). On the other hand, choosing a rate that is too high might lead to jumping over the desired minimum - or valley in our metaphor - and thus divergence is possible (right of Figure 3.2). One finds the optimal step size by either solving another optimization problem with regard to the step size $\gamma^{(i)}$ or by applying other line search methods of which **Armijo's rule** introduced in [12] and the **delta rule** investigated in [172] are the most common.

Algorithm 1: The gradient descent algorithm for unconstrained minimization problems

Input : A loss function ℓ_α, randomized initial iterative $\alpha^{(0)}$, an error tolerance ϵ and a maximum number of iterations n_{max}.

Output: (Local) minimum α^* of ℓ_α.

```
for i = 0, 1, ..., n_max do
    if |∇ℓ_{α^(i)}| > ε then
        Set direction of descent s^(i) = −∇ℓ_{α^(i)}(α^(i)).
        Determine step size γ^(i) using some step size algorithm.
        Compute new iterate α^(i+1) := α^(i) + γ^(i) s^(i).
    else
        Stop and return α* = α^(i).
    end
end
```

There exists mathematical theory providing conditions to guarantee the convergence of GD for unconstrained problems and convex loss functions. According to Shi and Iyengar,

$$\gamma^{(i)} < \frac{2}{L} \quad \text{for all } i, \tag{3.34}$$

is such a necessary and sufficient condition for the convergence to a global minimum, where L is the Lipschitz constant of $\nabla\ell$ (Theorem 7.1 in [160]).

This nice result only holds for this rather simple setting. For general nonconvex optimization, it can only be shown that GD converges to stationary points which is not sufficient for most ML tasks ([122]). Nevertheless, if for a nonconvex problem the objective or loss function only allows for strict stationary points and is twice differentiable, a global minimum can be found if the initial value $\alpha^{(0)}$ is generated randomly and the step sizes are fixed and smaller than $\frac{1}{L}$ ([128], Corollary 7.1, Lemma 7.1 and Proposition 7.1 in [160]). But in general, GD is a pretty slow optimizer even though the quality of the results is very high.

If we apply GD on the whole batch of training data $\mathcal{D}_{\text{train}}$ in every step, we call the algorithm **batch gradient descent**.

Example 3.9. *Let $\mathcal{D}_{train} = (X_{train}, Y_{train})$ be the training data set and let the averaged loss function ℓ be given by the MSE. Then the gradient of the loss function is*

$$\nabla_{\alpha^{(i)}} MSE\left(X_{train}, Y_{train}, \alpha^{(i)}\right) = \frac{2}{m_x} X^T\left(X\alpha^{(i)} - Y_{train}\right). \tag{3.35}$$

Furthermore, the new iterate $\alpha^{(i+1)}$ can be computed as follows:

$$\alpha^{(i+1)} = \alpha^{(i)} - \gamma^{(i)}\left(\nabla_{\alpha^{(i)}} MSE\left(X_{train}, Y_{train}, \alpha^{(i)}\right)\right) \tag{3.36}$$

for a step size $\gamma^{(i)}$.

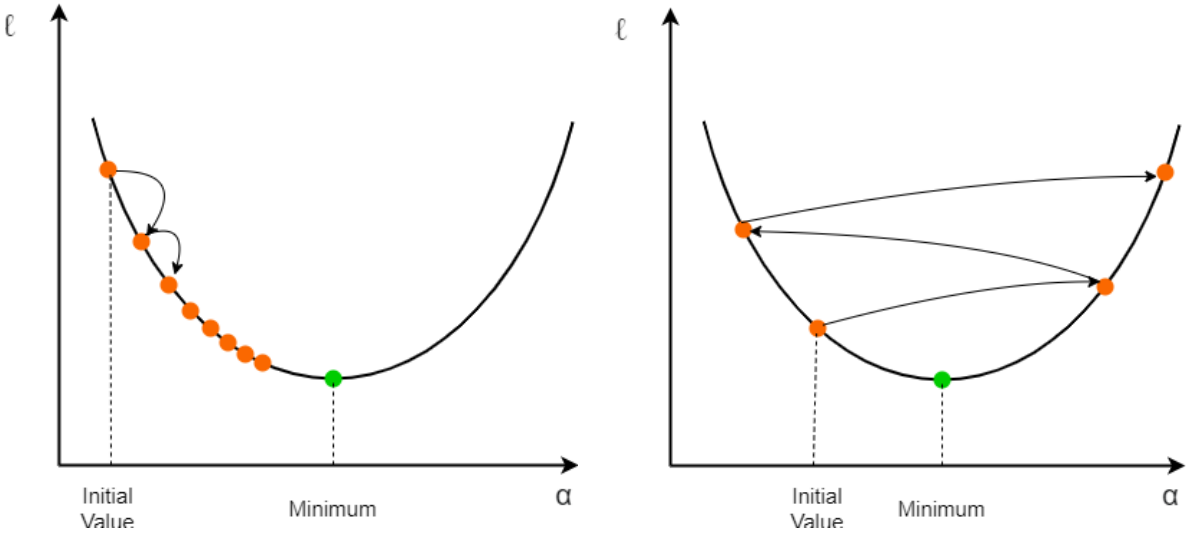


Figure 3.2: Illustration for choice of step size for gradient descent: (left) step size chosen too small and (right) chosen too big. The figure follows Figures 4-4 and 4-5 in [60].

Computing the gradient using the whole data set potentially requires a lot of computational resources and time. To tackle this issue, a method called **stochastic gradient descent (SGD)**has been introduced. The idea of this approach is to compute the gradient only of one small and randomly chosen instance of X_{train}. As a consequence, the algorithm proceeds a lot faster but does not necessarily settle down in the optimum unless the learning rate is reduced step by step. Conditions for convergence of SGD can be found in [147] and a convergence result for a non-convex setting is presented in [56]. **mini-batch gradient Descent** combines both ideas by computing the gradients of small random sets of data and can thus be parallelized easily.

Since in some applications, and in particular for machine learning tasks, very large data sets and high dimensional inputs are common, it is desirable to further accelerate the process of identifying the minimum. In 1964, Polyak ([138]) introduced **momentum optimization**, where a momentum vector m remembers gradients from previous steps and is taken into account when computing the next iterate $\alpha^{(i+1)}$. To prevent the momentum from having too much of an impact, a new hyperparameter $\delta \in [0, 1]$, called the **momentum**, is introduced. $\delta = 0$ means that there is a high friction on the impact of previous gradients while $\delta = 1$ implements no friction at all. A typical value for δ would be 0.9. Formally, the update of the iterate in Algorithm 1 has to be adapted and we end up with Algorithm 2.

Momentum optimization has proven to work well in practice and is usually significantly faster than regular GD ([60], p.353). Mathematically speaking, the momentum is an approach that takes a vague approximation of the second-order derivative into account

Algorithm 2: The momentum optimization algorithm

Input : A loss function ℓ_α, randomized initial iterative $\alpha^{(0)}$, an error tolerance ϵ, a maximum number of iterations n_{max} and a momentum δ.

Output: (Local) minimum α^* of ℓ_α.

```
1  for i = 0, 1, ..., n_max do
2      if |∇ℓ_{α^(i)}| > ε then
3          Set direction of descent s^(i) ← −∇ℓ_{α^(i)}(α^(i)).
4          Determine step size γ^(i) using some step size algorithm.
5          if i = 0 then
6              m^(0) ← 0
7          else
8              Compute new momentum m^(i) ← δm^(i−1) − γ^(i)s^(i)
9          end
10         Compute new iterate α^(i+1) := α^(i) + m^(i).
11     else
12         Stop and return α* = α^(i).
13     end
14 end
```

without explicitely computing the Hessian whose computation is extremely costly if it is not well-conditioned.

In 1983, Nesterov presented a modification of this approach, known as **Nesterov accelerated gradient (NAG)** ([123]). In NAG the update of the direction of descent in Line 3 of Algorithm 2 changes to

$$\text{Set direction of descent } s^{(i)} \leftarrow -\nabla \ell_{\alpha^{(i)}} \left(\alpha^{(i)} + \beta m^{(i-1)}\right). \tag{3.37}$$

In words, the gradient is evaluated slightly ahead of the current iterate $\alpha^{(i)}$ in direction of the momentum vector $m^{(i-1)}$. In practice, NAG proves to be even faster than momentum optimization.

It is straightforward to apply the methods introduced above to regularized models as well. This is called **regularized regression** and in this case, the regressions are called **ridge regression**, if the regularization term in Equation 3.17 is given by the Tikhonov regularization

$$P(\lambda, \alpha) = \lambda \|\alpha\|_2^2 \tag{3.38}$$

and **lasso regression** if P is given by the ℓ_1-regularization

$$P(\lambda, \alpha) = r\lambda \|\alpha\|_2^2 + \frac{1-r}{2}\lambda \|\alpha\|_1. \tag{3.39}$$

Using a combination of both regularization terms as shown in Equation 3.20 is called **elastic net regression**.

For most regularized regression methods it is recommended to scale the data, e.g. to the intervals $[0, 1]$ or $[-1, 1]$, before handing them over to the algorithms since they are sensitive to high scaled data. More details about and examples for regularized regression can be found in Chapter 4 of [60].

So far, the theory of regression was only introduced for linear models of functions. "Linear" only refers to the linearity in the parameters and thus allows us for nonlinear transformations of the inputs to generalize the regression model by linearly combining the components of this transformation, to ultimately lead to what is known as **nonlinear regression**. The transformation is given by a nonlinear function $\psi : \mathbb{R}^{m_x} \to \mathbb{R}^k$ and ψ_k denotes the kth component of the transformed feature vector ψ. For a set of parameters $\alpha \in \mathbb{R}^K$, the regression model is then given by

$$y_i = \psi^T(x_i)\alpha, \quad i = 1, \ldots, n_x, \tag{3.40}$$

which is still linear in its parameters α.

Example 3.10. *Having a polynomial model is possible by adding the power of each feature as new features by introducing the transformation*

$$\psi(x) = \begin{bmatrix}\psi_0(x) & \psi_1(x) & \ldots & \psi_{n_p}(x)\end{bmatrix} = \begin{bmatrix}1 & x & \ldots & x^{n_p-1}\end{bmatrix}, \tag{3.41}$$

where $n_p := K - 1$. In other words, instead of having the features

$$x = \begin{bmatrix}x_1 & x_2 & \ldots & x_{n_x}\end{bmatrix}^T, \tag{3.42}$$

we now have the transformed features

$$\psi(x) = \begin{bmatrix}1 & x_1 & x_1^2 & \cdots & x_1^{n_p} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n_p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n_x} & x_{n_x}^2 & \cdots & x_{n_x}^{n_p}\end{bmatrix}. \tag{3.43}$$

Replacing X_{train} in Lemma 3.6 by $\psi(X_{\text{train}})$, an analytical optimal solution α^* in the sense of empirical risk minimization exists, if $\psi^T(X_{\text{train}})\,\psi(X_{\text{train}})$ is invertible:

$$\alpha^* = \left(\psi^T(X_{\text{train}})\,\psi(X_{\text{train}})\right)^{-1}\psi^T(X_{\text{train}})\,Y_{\text{train}}. \tag{3.44}$$

However, numerical methods such as the different versions of GD and regularization are used in the vast majority of tasks in practice.

Some of the regression algorithms in combination with a nonlinear transformation are particulary useful for classification tasks. In this particular context, the transformation function is called **activation function**. The probabilities of an instance belonging to a certain class are often estimated by **logistic regression**. Here, the **logistic function**

$$\sigma(t) := \frac{1}{1 + \exp(-t)} \in [0, 1] \tag{3.45}$$

is an example of a transformation that is chosen often. If inserting an instance x yields a value greater than 0.5, the model predicts that the instance x belongs to a particular class. This idea can be generalized to K binary classifiers using the **softmax regression** which is built on the **softmax function**

$$\hat{p}_k := \frac{\exp\left(s_k(x)\right)}{\sum_{j=1}^{K} \exp\left(s_j(x)\right)}, \tag{3.46}$$

where $s(x)$ contains the scores of each class for one instance x. More details on logistic regression can be found again in Chapter 4 of [60] and a variety of activation functions will be investigated further in Section 4.1.

3.4.3 Probabilistic Models

In the previous section, the focus was on the training of models as pure deterministic functions. As mentioned before, there also exists a probabilistic modeling approach. In this and the following section, which follow Sections 8.3 and 8.4 as well as Chapter 9 in [41], the modeling of uncertainty using probability distributions and optimization techniques for this approach will be introduced as well as the corresponding optimization methods. It should be noted that we are still looking at labeled supervised learning tasks and that the methods presented in the context of probabilistic modeling will not be relevant further on, apart from the related optimization methods in Section 3.4.4, and especially in the context of neural networks. Nevertheless, the probabilistic approach will be dealt with here, as it is of great relevance for some fields of machine learning.

Before we delve deeper into the theory, let us recall the definition of *conditional probabilities*.

Definition 3.11 (Conditional Probability)**.** *Let* $(\Omega, \mathcal{A}, \mathbb{P})$ *be a probability space,* $A, B \in \mathcal{A}$ *events and* $\mathbb{P}[B] > 0$*. Then the* **conditional probability** *of* A *given* B *is defined as*

$$\mathbb{P}[A|B] := \frac{\mathbb{P}[A, B]}{\mathbb{P}[B]}. \tag{3.47}$$

Note that in some literature, for the joint probability of A and B the notation $\mathbb{P}[A \cap B]$ is used instead of $\mathbb{P}[A, B]$.

In the setting of non-probabilistic modeling, loss functions have been introduced to fit a model of functions through minimization and to quantify the fit of the model. These type of functions does not allow us to evaluate the fit of probabilistic models which provide models describing the distribution of possible functions based on characterizing parameters α. A possibility to find a fitting model and evaluate it for this type of models is the well-known **maximum likelihood estimation (MLE)** which is characterised by the fact that it attempts to find the best parameter set in a restricted space which makes the generation of observations most probable. Let us now define what the *negative log-likelihood* is.

Definition 3.12 (Negative Log-Likelihood)**.** *For a set of parameters α, a random variable D and a family of probability densities $\mathbb{P}[D|\alpha]$ the* **negative log-likelihood** *is given by*

$$\mathcal{L}_D(\alpha) := -\log \mathbb{P}[D|\alpha]. \tag{3.48}$$

Choosing the notation $\mathcal{L}_D(\alpha)$ indicates the dependence on varying parameters α while the random variable D (which can be regarded as the data) is fixed. MLE aims at maximizing the negative log-likelihood with regard to the parameters and thus finding the most likely parameter α^* for a noisy set of data X. In the setting of supervised learning the data is given by the pairs $(x_1, y_1), \ldots, (x_N, y_N)$ with features $x_i \in \mathbb{R}^{n_x}$ and labels $y_i \in \mathbb{R}$. The goal is to specify the conditional probability distribution of the labels parametrized by a particular parameter set α. A typical example is that the noise in the observations follows a Gaussian distribution $\mathcal{N}(0, \sigma^2)$ for some variance σ^2.

A slightly different approach to estimating the parameters is called **maximum a posteriori (MAP) estimation**. The goal is to maximize the a posteriori distribution $\mathbb{P}[\alpha|X]$, building on prior knowledge about the distribution $\mathbb{P}[\alpha]$ of the parameters α which could for example be distributed like a multivariate Gaussian with mean 0 and a covariance matrix Σ. With the information at hand, it is possible to compute $\mathbb{P}[\alpha|X]$ using *Bayes' theorem* which allows us to draw conclusions about X given observations on Y.

Theorem 3.13 (Bayes' Theorem)**.** *Let $(\Omega, \mathcal{A}, \mathbb{P})$ be a probability space, $X, Y \in \mathcal{A}$ events and $\mathbb{P}[Y] > 0$. Then there holds*

$$\mathbb{P}[X|Y] = \frac{\mathbb{P}[Y|X]\mathbb{P}[X]}{\mathbb{P}[Y]}. \tag{3.49}$$

Proof. Applying the definition of the conditional property in Equation 3.47 for $\mathbb{P}[X|Y]$ and $\mathbb{P}[Y|X]$, we get

$$\mathbb{P}[X|Y] = \frac{\mathbb{P}[X,Y]}{\mathbb{P}[Y]} \quad \text{and} \quad \mathbb{P}[Y|X] = \frac{\mathbb{P}[X,Y]}{\mathbb{P}[X]}. \tag{3.50}$$

Transforming both equations to $\mathbb{P}[X,Y]$ and setting them equal yields

$$\mathbb{P}[X|Y]\mathbb{P}[Y] = \mathbb{P}[Y|X]\mathbb{P}[X] \quad \Leftrightarrow \quad \mathbb{P}[X|Y] = \frac{\mathbb{P}[Y|X]\mathbb{P}[X]}{\mathbb{P}[Y]}. \tag{3.51}$$

Considering that $\mathbb{P}[Y] > 0$, this completes the proof. □

The **posterior** $\mathbb{P}[X|Y]$ tells us about what we know about X after having observed Y. The **likelihood** $\mathbb{P}[Y|X]$ describes the relationship between the input X and Y. Furthermore, we refer to $\mathbb{P}[Y]$ as the **evidence** and to $\mathbb{P}[X]$ as the **prior**.

Applying Theorem 3.13 to $\mathbb{P}[\alpha|x]$, where $x = X_i \in \mathbb{R}^{m_x}$ for some data $X \in \mathbb{R}^{n_x \times m_x}$, yields

$$\mathbb{P}[\alpha|x] = \frac{\mathbb{P}[x|\alpha]\mathbb{P}[\alpha]}{\mathbb{P}[x]}. \tag{3.52}$$

It is desired to find the set of parameters α that maximizes the posterior. Since $\mathbb{P}[X]$ does not depend on α, this reduces to maximizing the numerator. That is what is called MAP estimation. One can interprete MAP estimation as a cross-over between non-probabilitsic and probabilistic modeling since the knowledge of a prior distribution is necessary and only a point estimate is generated.

When fitting the models, overfitting and underfitting should be avoided in any case. A general procedure in probabilistic modeling is to allow for a rich class of models with many parameters and use regularization techniques similar to these presented in the previous section. Restricting the prior can also prevent overfitting.

Taking the idea of probabilistic modeling with random variables one step further, we would like to determine not only the best estimate resulting from parameter estimations, like MLE or MAP estimation, but to learn about the full parameter distribution and thus gather more detailed information to make more robust decisions. Finding this posterior distribution $\mathbb{P}[\alpha|X]$ for some data set X is known as **Bayesian inference**.

First of all, Bayes' theorem is applied on the whole set of data X taking into account that

$$\mathbb{P}[X] = \int \mathbb{P}[X|\alpha]\mathbb{P}[\alpha]d\alpha, \tag{3.53}$$

yielding

$$\mathbb{P}[\alpha|X] = \frac{\mathbb{P}[X|\alpha]\mathbb{P}[\alpha]}{\mathbb{P}[X]}. \tag{3.54}$$

A prediction for a new feature $\hat{x} \in \mathbb{R}^{m_x}$ is made by averaging over all possible parameter values $\alpha \in \mathcal{A}$, i.e.

$$\mathbb{P}[x] = \mathbb{E}_\alpha\left[\mathbb{P}[x|\alpha]\right] = \int \mathbb{P}[x|\alpha]\mathbb{P}[\alpha]d\alpha. \tag{3.55}$$

In contrast to parameter estimation, the key computational problem is not an optimization but an integration task. Unfortunately, the estimate can only be computed analytically if the distribution of the prior and posterior are of the same type. Otherwise, approximations either deterministic or stochastic like **Markov Chain Monte Carlo** ([63]), have to be applied. We will not go into more detail here, but it is important to note that Bayesian inference allows for taking account of more information and is very useful in practice to find model parameters. Optimization of this type is often called **Bayesian optimization** and detailed theory can be found in [166]. It is possible to introduce **latent variables** allowing for simpler and richer models. Latent-variable models will be of our interest when turning to neural networks in Chapter 4 but we shall not deal with this concept any further here. For a short applied introduction to this topic, the reader is referred to the work of Ge ([59]).

3.4.4 Optimization Techniques for Probabilistic Models

In the context of non-probabilistic models and regression, we focused on maps F parametrized by parameters α that directly map the input x to the corresponding label $F(x) = y$. Now, probabilistic modeling generalizes this approach by allowing for noise ϵ in the observations and aiming to find out about the likelihood $\mathbb{P}[Y|X]$. Note that the instances will be denoted by (X_i, Y_i) since the setting is probabilistic and thus both X_i and Y_i are random variables that are conventionally noted in capital letters. Thus the basic model is

$$Y_i = F(X_i) + \epsilon. \tag{3.56}$$

To provide a better overview in this section, we will restrict ourselves to zero-mean Gaussian noise where the variance σ^2 is assumed to be known in advance, i.e. $\epsilon \sim \mathcal{N}(0, \sigma^2)$ and

$$\mathbb{P}[Y_i|X_i] = \mathcal{N}\left(Y_i|F(X_i), \sigma^2\right). \tag{3.57}$$

As we have seen above, non-linear regressions can also be traced back to linear ones, so we will restrict ourselves to linear models of the form $f(x) = x^T\alpha$, where $x, \alpha \in \mathbb{R}^{n_x}$. In this case the likelihood is given by

$$\mathbb{P}[Y_i|X_i] = \mathcal{N}\left(Y_i|X_i^T\alpha, \sigma^2\right). \tag{3.58}$$

Let's assume we have a data set $D = (X, Y)$ with conditionally independent features X_i and X_j and the corresponding outputs Y_i and Y_j for all $i, j \in \{1, \ldots, N\}$ with $i \neq j$. The following example illustrates how the desired distribution or likelihood can be determined. The basic idea is to transform the expression for the negative log-likelihood and maximize it similarily to what can be done for standard linear regression.

Example 3.14. *For a linear model $X_i^T\alpha$ the Gaussian noise for each instance (X_i, Y_i), $i = 1, \ldots, N$, the likelihood is given by*

$$\mathbb{P}[Y_i|X_i, \alpha] = \mathcal{N}\left(Y_i|X_i^T\alpha, \sigma^2\right). \tag{3.59}$$

If the set of labeled pairs is iid, which is often assumed in machine learning tasks, it is possible to wrap the negative log-likelihood of the whole dataset $D = (X, Y)$ into one term:

$$\mathcal{L}_X(\alpha) = -\log \mathbb{P}[Y|X, \alpha] \overset{\text{identically distributed}}{=} -\log \prod_{i=1}^{N} \mathbb{P}[Y_i|X_i, \alpha] \overset{\text{ind.}}{=} -\sum_{i=1}^{N} \log \mathbb{P}[Y_i|X_i, \alpha]. \tag{3.60}$$

For Gaussian noise and a linear model, we can continue by inserting the definition of the

Gaussian density and obtain

$$\begin{aligned}\mathcal{L}_X(\alpha) &= -\sum_{i=1}^{N} \log \mathbb{P}[Y_i|X_i,\alpha] = -\sum_{i=1}^{N} \log \mathcal{N}\left(Y_i|X_i^T\alpha,\sigma^2\right) \\ &= -\sum_{i=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(Y_i - X_i^T\alpha)^2}{2\sigma^2}\right)\right) \\ &= \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(Y_i - X_i^T\alpha\right)^2 - \sum_{i=1}^{N}\log\frac{1}{\sqrt{2\pi\sigma^2}}.\end{aligned} \tag{3.61}$$

Since the second term is constant, minimizing the likelihood function $\mathcal{L}_X(\alpha)$ reduces to minimizing the first term which corresponds to a standard least-squares problem:

$$\begin{aligned}\alpha_{ML}^* &= \arg\max_{\alpha\in\mathcal{A}} \mathcal{L}_X(\alpha) \\ \Leftrightarrow \alpha_{ML}^* &= \arg\max_{\alpha\in\mathcal{A}} \frac{1}{2}\sum_{i=1}^{N}\left(Y_i - X_i^T\alpha\right)^2.\end{aligned} \tag{3.62}$$

Here, we dropped the factor $\frac{1}{\sigma^2}$ because of our assumption that σ is known a priori and thus not a free parameter. In the setting of this example and according to Lemma 3.6, the analytical solution of the optimization problem exists if X^TX is invertible and is given by

$$\alpha_{ML}^* = \left(X^TX\right)^{-1}X^TY. \tag{3.63}$$

The likelihood, we have been looking for, is then given by

$$\mathbb{P}[Y_i|X_i,\alpha] = \mathcal{N}\left(Y_i|X_i^T\alpha_{ML}^*,\sigma^2\right) = \mathcal{N}\left(Y_i|X_i^T\left(X^TX\right)^{-1}X^TY,\sigma^2\right). \tag{3.64}$$

In words, this means that given X_i the labels Y_i are normally distributed with mean $X_i^T\alpha_{ML}^*$ and variance σ^2.

The least-squares problem in Equation 3.62 in the previous example can actually be solved analytically which is not the case in general. For other likelihood functions not based on Gaussian noise, we would have to turn to numerical optimization methods like gradient descent and its variants to end up with an approximate numerical solution.

Above, we mentioned that regression can be generalized to nonlinear regression by introducing a nonlinear transformation ψ. For the sake of short notations, we will use the abbreviated notation ψ for $\psi(X)$. In this case and if $\psi^T\psi$ is invertible, the maximum likelihood is given by

$$\begin{aligned}\mathbb{P}[Y_i|X_i,\alpha_{ML}^*] &= \mathcal{N}\left(Y_i|\psi^T\alpha_{ML}^*,\sigma^2\right) \\ &= \mathcal{N}\left(Y_i|\psi^T\left(\psi^T\psi\right)^{-1}\psi^TY,\sigma^2\right).\end{aligned} \tag{3.65}$$

So far, we have assumed the variance σ^2 to be known. However, maximum likelihood can also be applied to determine the maximum likelihood estimator σ^2_{ML} for the variance of the noise. The procedure is analogous to the MLE for the set of parameters α_{MLE} only differing in that the derivations are with respect to σ^2 instead of α. We start with looking at the log-likelihood

$$\log \mathbb{P}[Y|X, \alpha, \sigma^2] = \sum_{i=1}^{N} \log \mathcal{N}\left(Y_i|\psi^T\alpha, \sigma^2\right). \tag{3.66}$$

Plugging in the density of the normal distribution and setting the partial derivative with regard to σ^2 leads to the optimal analytical solution

$$\sigma^2_{ML} = \frac{1}{N}\sum_{i=1}^{N}\left(Y_i - \psi^T\alpha\right)^2. \tag{3.67}$$

This estimate of the noise variance can be interpreted as the squared distance between the noise-free regression function values and the noisy observations. More detailed explanantions and theory can be found in Section 9.2.1 in [41].

MLE is known to be prone for overfitting. To tackle this we could once again introduce regularization or we, alternatively, could turn to MAP estimation. Recall that the idea of this approach is to make use of Bayes' theorem, Theorem 3.13, given a prior distribution $\mathbb{P}[\alpha]$ of the parameters. The goal is to maximize the posterior distribution $\mathbb{P}[\alpha|X, Y]$ instead of the likelihood.

The following example presents an exemplary application of MAP estimation. Recall that we restricted ourselves to Gaussian noise ϵ.

Example 3.15. *Let's assume that the prior $\mathbb{P}[\alpha]$ is given by $\mathcal{N}(0, s^2 I)$ with I being the identity. Theorem 3.13 yields*

$$\mathbb{P}[\alpha|X, Y] = \frac{\mathbb{P}[Y|X, \alpha]\mathbb{P}[X]}{\mathbb{P}[Y|X]}. \tag{3.68}$$

Maximizing this term is equivalent to maximizing only the numerator since $\mathbb{P}[Y|X]$ does not depend on α. Ignoring the term in the denominator, the negative log-posterior is given by

$$-\log \mathbb{P}[\alpha|X, Y] = -\log \mathbb{P}[Y|X, \alpha] - \log \mathbb{P}[X] \tag{3.69}$$

with the gradient being

$$-\frac{d \log \mathbb{P}[\alpha|X, Y]}{d\alpha} = -\frac{d \log \mathbb{P}[Y|X, \alpha]}{d\alpha} - \frac{d \log \mathbb{P}[X]}{d\alpha}. \tag{3.70}$$

The first term already appeared in Example 3.14 and thus the MAP estimate formulates a compromise between the prior and the likelihood of the data. Maximizing the log-posterior is equivalent to minimizing the negative log-posterior and hence,

$$\alpha^*_{MAP} \in \arg\min_{\alpha \in \mathcal{A}} \left(-\log \mathbb{P}[Y|X, \alpha] - \log \mathbb{P}[X]\right). \tag{3.71}$$

Analogous to the procedures in Lemma 3.6 and Example 3.14 and by generalizing it taking a nonlinear transform ψ into account, we get that we have to minimize

$$-\log \mathbb{P}[\alpha|X,Y] = \frac{1}{2\sigma^2}(Y-\psi\alpha)^T(Y-\psi\alpha) + \frac{1}{2s^2}\alpha^T\alpha. \tag{3.72}$$

We do so by determining the gradient, setting Equation 3.70 equal to 0 *and then isolating* α*:*

$$\begin{aligned} -\frac{d\log \mathbb{P}[\alpha|X,Y]}{d\alpha} &= \frac{1}{\sigma^2}\left(\alpha^T\psi^T\psi - Y^T\psi\right) + \frac{1}{2s^2}\alpha^T \overset{!}{=} 0 \\ \Leftrightarrow \alpha^*_{MAP} &= \left(\psi^T\psi + \frac{\sigma^2}{s^2}I\right)^{-1}\psi^T Y. \end{aligned} \tag{3.73}$$

This solution is similar to the one found by MLE in Equation 3.63. The only difference is the additional term $\frac{\sigma^2}{s^2}I$*, which ensures that the inverse exists and* α^*_{MAP} *is the unique solution. Furthermore, the posterior distribution is then given by*

$$\mathbb{P}[\alpha^*_{MAP}|X,Y] = \frac{\mathbb{P}[Y|X,\alpha^*_{MAP}]\mathbb{P}[X]}{\mathbb{P}[Y|X]}. \tag{3.74}$$

Remark 3.16. *Note that the higher the degree of the model function* $\psi(X)$*, the higher impact of the prior becomes. This is very useful to reduce the risk of running into overfitting. If we compare this to regularized regression, it is obvious that the prior acts like a regularizer. In the sense of regularized regression, the Tikhonov regularization (Equation 3.18) is given by the regularization parameter* $\lambda = \frac{1}{2s^2}$ *here.*

Taking the idea of probabilistic modeling one step further by deciding not to fit parameters but to determine the full posterior distribution and then compute a mean over "plausible" parameters $\alpha \in \mathcal{A}$. This approach is known as **Bayesian linear regression**. The model considered includes a prior and a likelihood. We restrict ourselves once again to a Gaussian prior and Gaussian noise. Thus, the model for Bayesian linear regression is given by

$$\begin{aligned} \mathbb{P}[\alpha] &= \mathcal{N}(m_0, S_0) \\ \mathbb{P}[Y_i|X_i,\alpha] &= \mathcal{N}\left(Y_i|\psi^T\alpha, \sigma^2\right), \end{aligned} \tag{3.75}$$

where m_0 and S_0 denote the mean and the covariance matrix respectively. Since we would like to make predictions with the model, we aim at finding the joint distribution $\mathbb{P}[Y_i,\alpha|X_i] = \mathbb{P}[Y_i|X_i,\alpha]\mathbb{P}[\alpha]$. In this Bayesian setting, the focus is not on the parameters themselves. Taking the prior distribution $\mathbb{P}[\alpha]$ into account, the average prediction $\hat{y}$ over all plausible α given a test feature $\hat{x}$ can be obtain via

$$\mathbb{P}[\hat{y}|\hat{x}] = \int_{\mathcal{A}} \mathbb{P}[\hat{y}|\hat{x},\alpha]\mathbb{P}[\alpha]d\alpha = \mathbb{E}_\alpha\left[\mathbb{P}[\hat{y}|\hat{x},\alpha]\right]. \tag{3.76}$$

No training data is needed to make this prediction, only the feature $\hat{x}$.

Example 3.17. *If the prior distribution is the one given in Equation 3.75 the predictive distribution is a normal:*

$$\mathbb{P}[\hat{y}|\hat{x}] = \mathcal{N}\left(\psi^T(\hat{x})m_0, \psi^T(\hat{x})S_0\psi(\hat{x}) + \sigma^2\right). \tag{3.77}$$

The computation is based on theoretical results on normal distributions and linearity which can be revisited in Sections 6.5 and 6.6 in [41]). Here, the term $\psi^T(\hat{x})S_0\psi(\hat{x})$ accounts for the uncertainty associated with the parameters while the term σ^2 deals with the uncertainty evoked by the noise.

If predicting noise-free values is of interest, the resulting distribution is given by

$$\mathcal{N}\left(\psi^T(\hat{x})m_0, \psi^T(\hat{x})S_0\psi(\hat{x})\right). \tag{3.78}$$

On the other hand, for a training data set $D_{train} = (X_{train}, Y_{train})$, the **marginal likelihood** *$\mathbb{P}[X_{train}|Y_{train}]$, a likelihood averaged over all plausible parameters, can be determined in a similar fashion:*

$$\mathbb{P}[X_{train}|Y_{train}] = \int_{\mathcal{A}} \mathbb{P}[X_{train}|Y_{train}, \alpha]\mathbb{P}[\alpha]d\alpha = \mathbb{E}_\alpha\left[\mathbb{P}[X_{train}|Y_{train}, \alpha]\right]. \tag{3.79}$$

The parameter posterior $\mathbb{P}[\alpha|X_{train}, Y_{train}]$ can then be determined by applying Bayes' formula, which in this case is

$$\mathbb{P}[\alpha|X_{train}, Y_{train}] = \frac{\mathbb{P}[Y_{train}|X_{train}, \alpha]\mathbb{P}[\alpha}{\mathbb{P}[Y_{train}|X_{train}}. \tag{3.80}$$

In our setting with a normal distributed prior and noise and under consideration of Theorem 9.1 in [41], the parameter prior can be computed explicitly as

$$\mathbb{P}[\alpha|X_{train}, Y_{train}] = \mathcal{N}\left(\alpha|\tilde{m}, \tilde{S}\right), \tag{3.81}$$

where

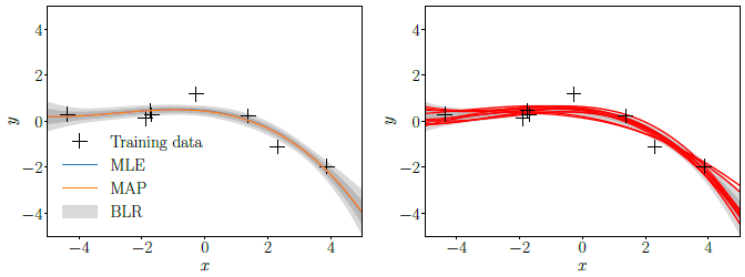
$$\begin{aligned} \tilde{S} &= \left(S_0^{-1} + \sigma^{-2}\psi^T\psi\right)^{-1} \\ \tilde{m} &= \tilde{S}\left(S_0^{-1}m_0 + \sigma^{-2}\psi^T Y_{train}\right) \end{aligned} \tag{3.82}$$

In the above equatoions, a short form notation $\psi := \psi\left(X_{train}\right)$ for a nonlinear transformation ψ has been introduced. Analogous to Equation 3.77 and given the training set D_{train}, we end up with the predictive distribution

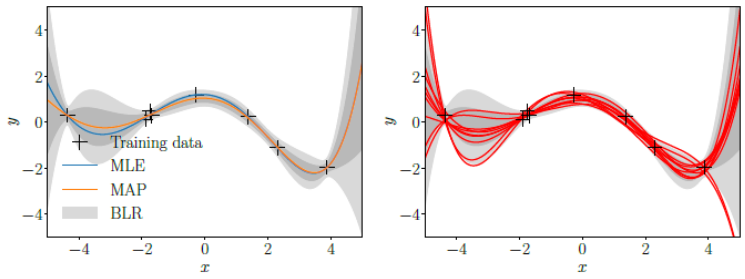
$$\mathbb{P}[\hat{y}|X_{train}, Y_{train}, \hat{x}] = \mathcal{N}\left(\psi^T(\hat{x})\tilde{m}, \psi^T(\hat{x})\tilde{S}\psi(\hat{x}) + \sigma^2\right), \tag{3.83}$$

where again the first term of the variance reflects the uncertainty of the parameters and σ^2 the noise uncertainty. Moreover, following the results presented by Deisenroth et al. ([41]) in Section 9.3.5 the marginal likelihood under the model (3.75) can be computed to be

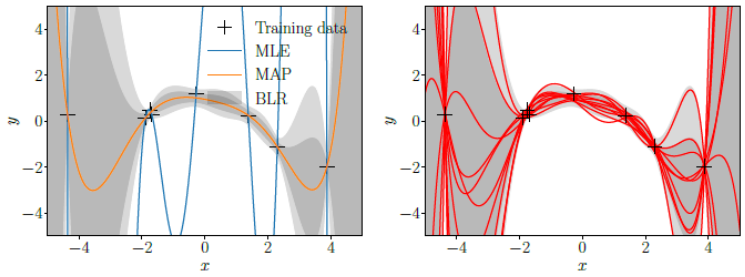
$$\mathbb{P}[Y_{train}|X_{train}] = \mathcal{N}\left(X_{train}m_0, X_{train}S_0X_{train}^T + \sigma^2 I\right). \tag{3.84}$$



(a) Posterior distribution for polynomials of degree $M = 3$ (left) and samples from the posterior over functions (right).



(b) Posterior distribution for polynomials of degree $M = 5$ (left) and samples from the posterior over functions (right).



(c) Posterior distribution for polynomials of degree $M = 7$ (left) and samples from the posterior over functions (right).

Figure 3.3: Posterior distributions for Bayesian linear regression. Dark gray shades indicate the 67% and light gray bounds the 95% confidence bounds. MLE and MAP generate a single estimate while Bayesian linear regression returns the whole distribution. (Taken from [41], Figure 9.11)

Figure 3.3 shows examples of posterior distributions for Gaussian models. While MLE and MAP only provide us with a point estimate, Bayesian linear regression offers us the whole range of possibilities given by the distribution. Some samples drawn from the posterior distribution are shown on the right. MLE is known to be prone for overfitting, which can be observed in the bottom of the figure, where the MLE estimate is oscillating. MAP gives an estimate that seems to be more realistic and with a smaller error. The posterior distribution offers us both at once and also tells us that the posterior uncertainty is huge for a high-degree polynomial model.

Before we introduce new machine learning approaches in the next chapter, let us first be more specific about how we actually select models which allow for a high quality generalization and thus for reasonable predictions.

3.5 Model Selection

This section is following Section 8.6 in [41].

For non-probabilistic models and small data sets, we already introduced the concept of cross-validation to achieve a better generalization. In cross-validation, the data is separated into K chunks and then for all possible combinations $K-1$ of them serve as training data while the remaining one is used for testing and evaluating. Taking the average over all performance measures allows for an approximation of the expected generalization error.

In practice, it is common to go one step further. For non-probabilistic approaches, the idea of cross-validation can be carried into every shuffled model $f^{(k)}$ for $k = 1, \ldots, K$ (**nested cross-validation**). This means that within the training set, we repeat cross-validation and for the sake of distinguish them from the test set separated from the training data, the inner test sets $\mathcal{V} = \left(\mathcal{V}^{(1)}, \ldots, \mathcal{V}^{(V)}\right)$ are called **validation sets**. Figure 3.4 illustrates this idea for one model at both levels. The performance of different models can be evaluated on the inner level, while the outer level of classical cross-validation provides us with an estimate of the generalization performance measured, e.g. by the RMSE or other loss functions:

$$\mathbb{E}[\ell\left(\mathcal{V}, f^{(k)}\right)] \approx \frac{1}{K}\sum_{i=1}^{V} \ell\left(\mathcal{V}^{(i)}, f^{(k)}\right). \tag{3.85}$$

In addition, any form of cross-validation allows for an analysis of high-order statistics starting with the standard error and thus helps to precisely frame the uncertainty. Finally, the non-probabilistic model with the best performance on the test set should be chosen.

If a probabilistic approach is taken, the quantification of the trade-off between complexity and fit of a model is more challenging. Finding a model as simple as possible which explains the data reasonably well is known as the statistical version of **Occam's razor**. A simpler model is most often only capable to make proper predictions for a small variety of input data, while more complex models cover a wide range. The costs of this higher degree of universality are that the precision on data sets that are covered by the simpler model version is significantly lower, i.e. the model runs into overfitting. This trade-off is already embodied when using the marginal likelihood for estimation considering that the model parameters are integrated out. In **Bayesian model selection**, this property is exploited. If a set of candidate models $f = \{f^{(1)}, \ldots, f^{(K)}\}$ which are each characterized by a set of parameters α_k out of a parameter space $\mathcal{A}$ for $k = 1, \ldots, K$ is based on the

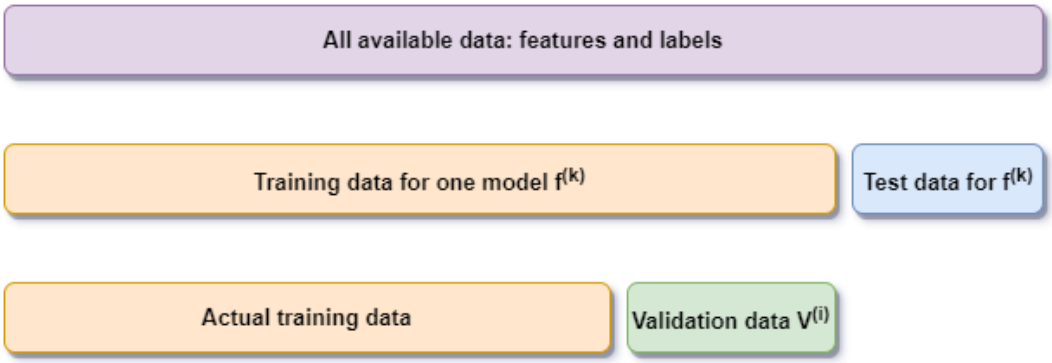


Figure 3.4: Illustration of nested cross-validation. Orange indicates training data, blue data for testing and green validation sets. (Inspired by [41], Figure 8.13)

basic Bayesian model

$$\begin{aligned} F^{(k)} &\sim \mathbb{P}\left[F\right] \\ \alpha_k &\sim \mathbb{P}\left[\mathcal{A}|F^{(k)}\right] \\ \mathcal{D} &\sim \mathbb{P}\left[\mathcal{D}|\alpha_k\right]. \end{aligned} \tag{3.86}$$

Here, $\mathcal{D}$ is a given set of training data and the set of models is based on a prior $\mathbb{P}\left[F\right]$ which gives every model a priori a certain probability. The type of the probability distribution of $\mathbb{P}$ has to be identical for all three modeling steps, e.g. normal distributions. Once again, Bayes' theorem, Theorem 3.13, allows us to compute the desired posterior distribution for each model $F^{(k)}$. Arguing with proportionality and integration over plausible parameters, leads to the possibility to determine the MAP estimate of the best set of parameters α^* and thus a suggestion for the choice of a model:

$$\begin{aligned} & \alpha^* = \arg\max_{\alpha\in\mathcal{A}} \mathbb{P}\left[F^{(k)}|\mathcal{D}\right] \\ \Leftrightarrow \quad & \alpha^* = \arg\max_{\alpha\in\mathcal{A}} \mathbb{P}\left[F^{(k)}\right]\mathbb{P}\left[\mathcal{D}|F^{(k)}\right] \\ \Leftrightarrow \quad & \alpha^* = \arg\max_{\alpha\in\mathcal{A}} \mathbb{P}\left[F^{(k)}\right]\int \mathbb{P}\left[\mathcal{D}|\alpha_k\right]\mathbb{P}\left[\alpha_k F^{(k)}\right] d\alpha_k. \end{aligned} \tag{3.87}$$

In a more general setting, where two probabilistic models might not be based on the same Bayesian basic model, we can turn to **posterior odds** to compare two models $F^{(1)}$ and $F^{(2)}$ given data D. The ratio is defined as

$$\frac{\mathbb{P}[F^{(1)}|\mathcal{D}]}{\mathbb{P}[F^{(2)}|\mathcal{D}]} = \frac{\frac{\mathbb{P}[\mathcal{D}|F^{(1)}]\mathbb{P}[F^{(1)}]}{\mathbb{P}[\mathcal{D}]}}{\frac{\mathbb{P}[\mathcal{D}|F^{(2)}]\mathbb{P}[F^{(2)}]}{\mathbb{P}[\mathcal{D}]}} = \underbrace{\frac{\mathbb{P}\left[F^{(1)}\right]}{\mathbb{P}\left[F^{(2)}\right]}}_{\text{prior odds}} \underbrace{\frac{\mathbb{P}\left[\mathcal{D}|F^{(1)}\right]}{\mathbb{P}\left[\mathcal{D}|F^{(2)}\right]}}_{\text{Bayes factor}}. \tag{3.88}$$

The **prior odds** quantify the a priori favouring of model $F^{(1)}$ over $F^{(2)}$, whereas the **Bayes factor** measures the accuracy of the prediction of $\mathcal{D}$ by the two models. If the posterior odd is greater than 1, model $F^{(1)}$ is preferable over model $F^{(2)}$, otherwise the latter one will be chosen.

Remark 3.18. *For all methods for evaluation and estimation that make use of the marginal likelihood, integration has to be performed, unless all components of the Bayesian basic model follow the same type of distribution which would allow for a closed form solution. Numerical integration is costly and approximations have to be used. However, standard numerical approximations can lead to success but stochastic approximations based on a Bayesian Monte Carlo approach as suggested by Rasmussen and Ghahramani ([140]) turn out to be more promissing.*

3.6 Summary

In this rather lengthy and detailed chapter, we learned about the basic idea of machine learning methods which is to extrapolate a map to describe relations observed in data. The process of finding and fitting the map is called training. Different approaches and classifications have been introduced. Tasks can be classified into supervised, unsupervised and reinforcement learning while some algortihms are able to learn online in contrast to batch learning algorithms. A distinction is also made between approach-based on instances or models. We continued by turning to the challenges of ML of which the major is known as the bias-variance trade-off. Before models can actually be trained, the user has to decide if they wants to pursue a probabilistic model type or limit themself to models based on polynomials or non-linear functions to describe the relationship between the input and output of a process. Loss functions such as the root mean squared error have been introduced which have to be minimized using linear regression and different forms of gradient descent methods. To avoid overfitting, regularization can be added which penalizes complex structures with many degrees of freedom for a model. Generalizing the theory for linear models to non-linear models appeared to be straightforward by introducing a non-linear transform. For probabilistic methods, maximum likelihood and maximum a posteriori estimation play a key role to determine a best estimate of the posterior distribution of interest. To determine the whole distribution of the posterior distribution, Bayesian linear regression has proven to be very useful. If we end up with several models we want to compare to each other, the cross-validation in combination with MAP estimation and Bayesian model selection provide us with quantification tools to mathematically justify the choice of a model in the end.

Most of the theory is following Chapters 8 and 9 in [41]. This book is a good starting point for the interested reader to delve deeper into the mathematics behind machine learning in both a probabilistic or deterministic setting.

As already mentioned, we will not deal further with probabilistic models, but will focus

on deterministic approaches in the further course, which always generate exactly one estimate.

4 Artificial Neural Networks

So far, the structure of the potentially non-linear mapping F for inputs x and outputs y describing the relationship within or between the data has not been specified for more complex tasks. Both, probabilistic and non-probabilistic models have been introduced. The data we refer to in this chapter is assumed to be labled and thus the machine learning approach of supervised learning will be of our interest. In this chapter one approach to model F that is nowadys regarded as the core method of machine learning will be introduced, the so called *(Artificial) Neural Networks (NN)*. To develop an intuition for NNs, the easiest version - *feedforward neural networks (FFNNs)* - will be discussed in detail. We then focus on several methods to explicitly train NNs culminating in the *Adam* algorithm that will be used later when comparing the performance of different NN architectures for financial time series. To conclude the chapter, an example which illustrates the application of a simple FFNN to financial markets to the prediction of prices will be presented.

The first section begins with the motivation for using neural networks and is inspired in parts by Chapter 1 in [31].

4.1 Basics

(Artificial) neural networks (NN) are a sub-field of machine learning and deal with the modeling of the non-linear mapping F. They are inspired by the structure of biological neural networks in animal and human brains. These networks are relying on the foundational units which are called **neurons** whose main tasks are to receive and process information before they transmit the results of this process to other cells. The information enters into the **cell body** through **dendrites** that are connected to upstream cells. The more often a dendrite has been used, the stronger it is and the higher the information entering through this very dendrite is weigthed when processing the information in the cell body. The processed information is then send out to other cells as a new signal that travels along the **axon** and exits the neuron through the **axon terminals**. The left part of Figure 4.1 sketches this simplified anatomy of a biological neuron. Neurons are very small and are entangled with about 6000 other neurons ([145]).

In 1943, McCulloch and Pitts suggested a first model for computers inspired by the structure of biological neural networks ([116]). The immitations of neurons that can be

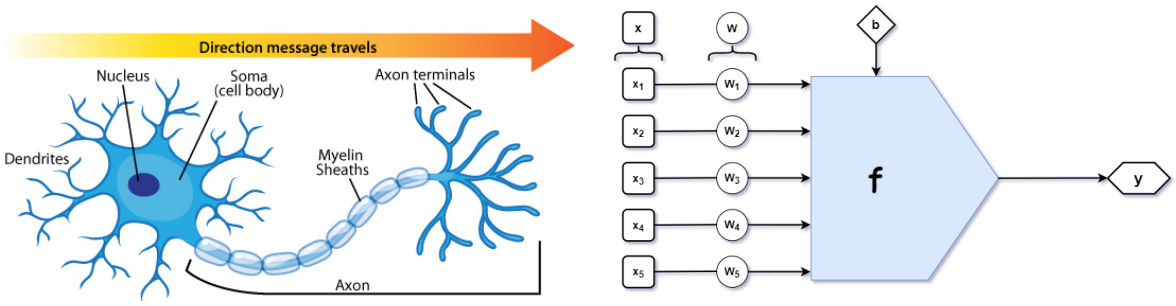


Figure 4.1: (Left) The anatomy of a biological neuron and (right) the structure of an artificial neuron. The figure on the left is taken from [29].

understood by computer are called **artificial neurons**. The input of the neuron is given by a potentially high-dimensional input matrix X corresponding to the information that enters biological cells through the dendrites. Rows of X are the features in the sense of supervised learning. The strength of the dendrites is modeled by a weight matrix W. To process the weighted input information, a semi-affine function f is used creating an output of desired shape that can be passed to the next neuron. An example for f is the indentity. In most settings, it is assumed that the data is noisy and we thus introduce a bias b that enters the processing function f. The output y of the neuron is then given by

$$y = f(XW + b). \tag{4.1}$$

Simple tasks can often be handled well with f being a function of linear type:

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}, \quad f(z) := cz + d, \tag{4.2}$$

for $c \in \mathbb{R}^n$ and $d \in \mathbb{R}$. But in general it is desired, and actually necessary for complex learning tasks, to introduce some form of non-linearity. In practice, three types of functions, known as **activation functions** of the neuron, are heavily used.

Definition 4.1 (Non-linear activation functions)**.** *Let $z \in \mathbb{R}$ for $n \in \mathbb{N}$. The* **logistic function** *L is a function of the sigmoid type, i.e. following an S-shape, and defined by*

$$L : \mathbb{R} \longrightarrow [0, 1], \quad L(z) := \frac{1}{1 + \exp(-z)}. \tag{4.3}$$

If the output should be ranged from -1 to 1 instead of 0 to 1, another type of S-shaped functions called the **hyperbolic tangent function** $\tanh$ *defined as*

$$\tanh : \mathbb{R} \longrightarrow [-1, 1], \quad \tanh(z) := \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} = \frac{\exp(2z) - 1}{\exp(2z) + 1}, \tag{4.4}$$

should be applied. This one is often preferred over the former one because it is centered in 0. The third activation function is hockey-stick-shaped and called the **restricted linear unit** *ReLU and is defined by*

$$ReLU : \mathbb{R} \longrightarrow [-1, 1], \quad ReLU(z) := \max\{0, z\}. \tag{4.5}$$

The idea of the former two activiation functions is to project the function values into a compact interval to express the degree of activation of the neuron. Nevertheless, the restricted linear unit function is chosen for most practical tasks. Figure 4.2 shows plots of all three activation functions. The plots have been created in a iPython notebook and the code can be found in the attached file `Plot_Activation_Functions.ipynb`.

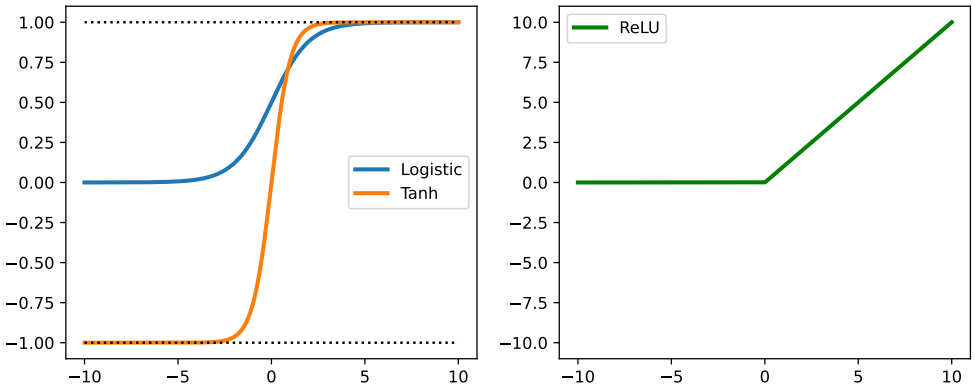


Figure 4.2: (Left) Plot of the logistic function L and the hyperbolic tangent function tanh and (right) Graph of restricted linear unit ReLU, both for $z \in [-10, 10]$.

Note that the definition of each activation function $\sigma \in \{L, \tanh, \mathrm{ReLU}\}$ can be generalized to $\tilde{z} \in \mathbb{R}^n$ by applying the function to every entry of the vector, i.e.

$$\sigma(\tilde{z}) := \begin{bmatrix} \sigma(\tilde{z}_1) \\ \vdots \\ \sigma(\tilde{z}_n) \end{bmatrix}. \tag{4.6}$$

Example 4.2. *For the simple case that the input is a vector $x \in \mathbb{R}^n$, the weights are given by a vector $w \in \mathbb{R}^n$, the bias b is a scalar and for f being the indentity, this results in*

$$y = \sum_{i=1}^{n} w_i x_i + b. \tag{4.7}$$

The right part of Figure 4.1 illustrates this simple form of a single artificial neuron and $n = 5$.

Obviously, a single neuron is not able to solve difficult tasks like detecting fraud in financial markets. To tackle this, many neurons are connected and combined in one network, a so called **(artifical) neural network (NN)**. We follow the biological brain and organize these neurons in **layers** which are groupings of neurons. Different layers do not have to

contain the same amount of neurons in general. There are two layers that appear in every NN: the **input** and the **output layer**. The latter one can be read by the user and delivers the data for a further interpretation. If the information or data is not just passed through the neuron without processing it, at least one layer of neurons between these two is mandatory. The processes in the layers that are actually processing data are not necessarily observable, which is why they are called **hidden layers**. In analogy with the human brain, the input and the output can be observed, but what happens exactly in the inner parts of the neural network is hard if not impossible to track, even though this is where the magic happens. It is also important to mention, that a neuron in a layer does not necessarily have to be connected to every neuron in the next layer.

If a network consists of two or more hidden layers, it is called a **deep neural network (DNN)**. In this case, the l-th column of the weight matrix X correspond to the set of weights for the l-th layer (for $l = 1, \dots, L$) and the l-th column of the bias b corresponds to the bias entering in the l-th layer. On the other hand, if there is only one hidden layer in the NN and the activation function is the identity, the relationship between the input and the output is just of a linear type.

The most intuitive structure of a NN is that the neurons of each layer are forward-connected to the neurons in the next layer by non-linear functions which are composed one after the other. This is known as a *feedforward neural network* and belongs to the class of supervised machine learning tasks since the inputs mark the features and the output the labels of the data. Recall that a neural networks aims at the following relation:

$$F(x) = y + \epsilon, \tag{4.8}$$

where ϵ is some error term. In the following, the map is often applied to the whole set of features at once, i.e. $F(X) = Y + \epsilon$ with $X = (x_1, \dots, x_{n_x})^T \in \mathbb{R}^{n_x \times m_x}$ and $Y = (y_1, \dots, y_{n_y})^T \in \mathbb{R}^{n_y \times m_y}$. Often, $m_y = 1$ for the labels. This has to be understood element-wise: $F(X) = (F(x_1), \dots, F(x_{m_x}))$.

The following definition introduces a mathematical formulation of this basic network type.

Definition 4.3 (Feedforward Neural Network)**.** *A neural network with L layers modelled by the map $F : \mathbb{R}^{n_x \times m_x} \to \mathbb{R}^{n_y \times m_y}$ for inputs $X \in \mathbb{R}^{n_x \times m_x}$ and outputs $Y \in \mathbb{R}^{n_y \times m_y}$ is called a* **feedforward neural network (FFNN)***, if the mapping F is given by*

$$F_{W,b}(X) := \left(f^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}} \right)(X) \approx Y, \tag{4.9}$$

where

$$f^{(l)}_{W^{(l)},b^{(l)}}(\tilde{X}) := y^{(l)} := \sigma^{(l)} \left(W^{(l)} \tilde{X} + b^{(l)} \right) \tag{4.10}$$

are semi-affine functions for all $l = 1, \dots, L$ with $\sigma^{(l)}$ being a univariate and continuous non-linear activation function for the l-th layer, the weight matrices $W = \left(W^{(1)}, \dots, W^{(L)}\right)$, where $W^{(l)}$ denotes the weight matrix for the l-th layer and the biases $b = \left(b^{(1)}, \dots, b^{(L)}\right)$.

Remark 4.4. *In the above definition, naming the exact dimensions of every weight matrix and bias has been avoided. This is due to the fact that, as mentioned above, the amount of neurons in every layer and the amount of connections to the next layer usually vary and it is useful to avoid a notation that is loaded. Even though it is not satisfactory, we now have to stick to the bold statement: Assume that the dimensions are appropriate so that the matrix multiplications are well-defined.*

Figure 4.3 (left) shows a simple example of a FFNN with a single hidden layer with two neurons (orange). In this simple example, the netowrk is **fully connected**, i.e. every neuron is connected with all neurons in the next layer, which is not mandatory. To demonstrate how neural networks different to FFNNs can be designed, the right part of the figure shows a so called *recurrent neural network (RNN)* with two hidden layers which allows to also process information from the same layer from previous passes of data. Informally speaking, the network has a memory of information it has processed before through its neurons. RNNs will be the subject of later studies in Section 5.1.

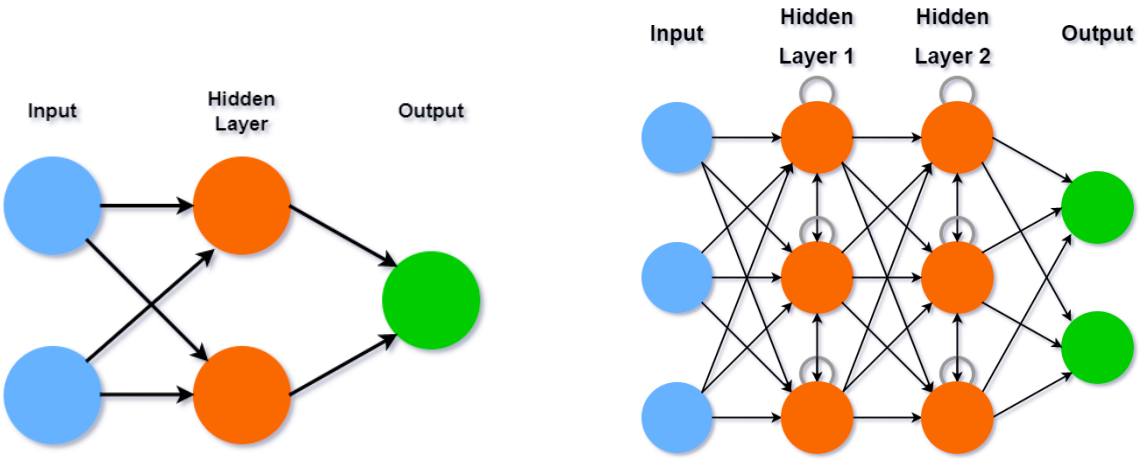


Figure 4.3: Different neural networks: (Left) A feedforward neural network with one hidden layer of latent variables consisting of two neurons and (right) a recurrent neural network with two hidden layers. The figures are inspired by [183].

The question how many layers and neurons to use is not easy to answer, but in practical applications FFNNs usually have much more than one hidden layer. The aim of this thesis is not to provide very in-depth knowledge about the functionality of machine learning and network design but rather to compare and point out the unique selling points.

The most basic mathematical foundation for neural networks is the **universal representation theorem** presented by Hornik et al. ([84]) which states that a feedforward neural network with a single hidden layer can approximate any continuous function. They also

show that this result holds regardless of the activation function, the input layer and its dimension. In other words, having a FFNN with numerous neurons in the single hidden layers, it is theoretically guaranteed that any non-linear function can be satisfactorily approximated if only the number of neurons is high enough. Unfortunately, the theorem is not constructive at all and does not tell us about possible performance boosts when adding more layers which can be observed in applications. However, recent research like the work of Poggio ([137]) shows that deep NNs can achieve a significantly better performance than linear regression models while being able to deal with a lower dimensionality.

4.2 Training of Deep Neural Networks

In the previous section, we introduced neural networks and a first example. Analogously to the more general framework of machine learning in Section 3.4, we are interested in the mathematics behind the training process of a neural network. Recall that training a machine learning algorithm is all about reducing the error and learning the parameters of either the model function or the posterior distribution in the probabilistic Bayesian setting. In the context of NNs, the parameters we can influence are the weight matrices. Thus learning means nothing else than adjusting the weights until the process generates the desired labels given a set of input data.

Building on the knowledge of training in general machine learning settings, we will first introduce training methods for feedforward neural networks, which are relatively easy and straightforward. Afterwards, more complex techniques suitable for NNs other than FFNNs will be introduced before we turn to the Adam algorithm combining momentum optimization with finding a good learning rate and will be used later when we actually train networks with Python.

Before we introduce the first training methods, it is worth mentioning that an intelligent selection and increasing the size of the training data may help to speed up the training process. The selection process shall not be in the focus of this work, but methods that are used often in practice are scaling of the data and cleaning-up the data by excluding single extreme values. For now, we assume that we can access a training data set D that is large enough and of good quality. Unless stated otherwise, we focus on non-probabilistic models.

In the process of training, samples are sent through the network and the generated output is then evaluated. The number of samples that are sent through before the model weights are updated is called **batch size**. Most optimization algorithms work well with batch sizes of 16, 32 or 64. GPUs can handle batch sizes of 128 or 256 which may improve the performance of the training algorithm. Increasing the batch size includes fewer updates in the course of sending every sample through the network once, also called an **epoch**. It is highly recommended to train the model over several epochs, but

performing too many complete pass-throughs will ultimately result in overfitting. If the optimal amount of training epochs is hard to guess a priori, the addition of regularization reduces the chances of overfitting. Also, early stopping and model checkpointing are easy to implement with machine learning libraries in Python.

4.2.1 Training of Feedforward Neural Networks

This subsection deals exclusively with the training of FFNNs, follows Chapter 2 in [31] and parts of Chapter 10 in [60] while building on the methods introduced in Section 3.4.

The optimization taking place in the process of training the network focuses on minimizing a loss or error function $\ell_{W,b}$ that compares the actual observed labels with the output of the neural network parametrized by the weight matrix W and the bias b. Both can be regarded as the parameters $\alpha = \left(\alpha^{(1)}, \dots, \alpha^{(L)}\right) = \left((W^{(1)}, b^{(1)}), \dots, (W^{(L)}, b^{(L)})\right)$ of the model.

Recall that a FFNN builds on compositions of the functions $f^{(l)}_{\alpha^{(l)}} := \sigma^{(l)}\left(W^{(l)}X + b^{(l)}\right)$ for every layer $l = 1, \dots, L$. To be more precise, the functions can be written as

$$\begin{aligned} f^{(1)} &= \sigma^{(1)}\left(W^{(1)}X + b^{(1)}\right) \\ f^{(2)} &= \sigma^{(2)}\left(W^{(2)}\left(f^{(1)}(X)\right) + b^{(2)}\right) \\ &\vdots = \vdots \\ f^{(L)} &= \sigma^{(L)}\left(W^{(L)}\left(f^{(L-1)}(f^{(L-2)}(\cdots f^{(2)}(f^{(1)}(X))\cdots))\right) + b^{(L)}\right). \end{aligned} \tag{4.11}$$

We simplify the terms by defining

$$z^{(l)} := W^{(l)}\left(f^{(l-1)}(f^{(l-2)}(\cdots f^{(2)}(f^{(1)}(X))\cdots))\right) + b^{(l)}. \tag{4.12}$$

Using this, we can express $f^{(l)}$ a lot shorter as

$$f^{(l)} = \sigma^{(l)}\left(z^{(l)}\right). \tag{4.13}$$

The first idea, once again, is to perform gradient descent to minimize the loss function and find a good set of parameters. To do so, the gradient of the loss function with regard to the parameters α has to be computed. Since we are in the setting of a deep network, the chain rule of partial differentiation will be of great interest and importance to deal with the composed function F_α.

Lemma 4.5 (Chain Rule for Differentiation)**.** *For two functions f and g the derivative of $(g \circ f)(x)$ is given by*

$$\frac{\partial}{\partial x}(g \circ f)(x) = \frac{\partial g}{\partial f}\frac{\partial f}{\partial x}. \tag{4.14}$$

Proof. The proof can be found as well as additional theory in [11]. □

Numerically thinking, the biggest challenge is the computation of the gradient $\nabla_\alpha F$ to be able to perform gradient descent. Basically, there are four ways how to approach this: manual, numerical, symbolic and automatic differentiation techniques. The primer one includes the actual determination of the derivative by hand using the basic derivative rules of calculus which can be time consuming and is error prone. Numerical differentiation usually builds on **finite differences** ([171]) and allows for an algorithmic computation of the derivative. Even though it is easy to implement, problems occur with numerical stability and accuracy because of rounding and truncation errors ([94]). Furthermore, numerical differentiation methods scale poorly for gradients with a high number of parameters as observed often in machine learning tasks ([15]). Symbolic differentiation tries to tackle the weeknesses of the former methods by offering an automated version of manual differentiation applying the known derivation rules. This approach serves well for the accurate determination of derivative values but the function enters in closed form and thus cannot be handed to follow-up conditions and loops. In addition, expressions of the derivative can be a lot more complex than the original function (**expression swell**, [38]). The numerical technique of choice is **automatic differentiation** , which is also known as **autodiff**. In a nutshell, this method can be thought of as a set of numerical techniques to evalute the gradient up to machine precision. For this purpose, the functions are converted into a composition of elementary operations and the gradient is easily computable using the chain rule. There are two options for autodiff: forward-mode and reverse-mode autodiff. The former one is equivalent to symbolic differentiation in a sense that the same algorithmic operations are performed ([105]). For high-dimensional networks, reverse-mode autodiff turns out to be computationally cheaper than the forward-mode approach ([41], p.140f).

The exact mathematics of autodiff shall not be discussed here. The interested reader is referred to Baydin et al. ([15]) for in-depth theory for autodiff in machine learning and to Géron (Appendix D in [60]) who explains how the commonly used machine learning software library TensorFlow implements reverse-mode autodiff.

We are now able to introduce the concept of the **backpropagation** training algorithm that was introduced in the groundbreaking paper by Rumelhart, Hinton and Williams, published in 1985 ([149]). The core idea is to compute the changes in the error when changing the parameters and can hereby determine the sensitivity of the loss function with regard to single connections and weights. Backpropagation can be regarded as a special case of autodiff. Before its introduction, researchers struggled for many years to train deep neural networks. This algorithm basically offers an efficient method for the computation of the gradients by only passing the data through the networks twice - one forward and one backward pass. With the results a gradient descent step can be performed and the procedure is repeated until the loss or error is sufficiently small.

Backpropagation consists of multiple but rather simple and numerically cheap steps. Before the algorithm can do its work, a deep FFNN with given structure, activation functions and hyperparameters has to be initialized with randomly generated weights

and biases. It is important that the neurons in one layer have different initial weights. Otherwise the whole layer would be updated simultaneously and would act like a single neuron. Initialization is also very important to avoid vanishing or exploding gradients that often occur in deep neural networks. This challenge led to almost abandoning DNNs at the beginning of the current century before Glorot and Bengio pointed out the reason for the unfortunate behavior of the gradients and proposed a way to avoid it ([64]). Depending on the type of activation function used in the network, several researchers propose a different type of initialization. **Glorot initialization** is recommended for hyperbolic tangent, logistic or softmax activation functions, while **He initialization** introduced by He et al. in [74] performs best for rectified linear units and its variants. In 1998, Orr and Müller ([126]) already recommended **LeCun initialization** which can be regarded as a generalization of Glorot initialization but has been flying under the radar since. The different approaches differ in the variance used for the normal distributions that initializes the weights and biases. We will not cover these methods in detail and the interested reader is referred to the referenced literature.

After the network parameters have been initialized, we can now focus on the algorithm itself. A step of the backpropagation algorithm can be described verbally as follows:

1. An instance of input data is handed to the network and passed through the layers. This is similar to making a prediction and called **forward pass**. In addition, the intermediate results are preserved for later usage.
2. The output error between the generated output and the actual labels is computed using an (averaged) loss function ℓ.
3. The impact of each weighted connection in the last layer to the total error is computed using the chain rule.
4. The algorithm then moves backward through the network and determines the error contribution of each connection in the previous layer until it reaches the input layer (**reverse pass**). This allows to measure the loss gradient over all connections.
5. A gradient descent step is performed to tune the weights using the error gradient.
6. Repeat the procedure starting from 2. until the error is sufficiently small.

A pseudo code for the backpropagation algorithm is given in Algorithm 3.

Before training the network, it is worth noticing that apart from the parameters α that are a direct part of the model, hyperparameters such as the learning rate (or step rate) of the gradient descent step or restrictions on the parameters are key for the training performance and generalization power. Moreover, the methods introduced before to improve and speed up the training process are recommended to be used for this sort of descent as well. First of all, preparing the data by scaling and exclusion of extreme data can be very useful. Having high-quality data at hand, stochastic gradient descent and mini-batch gradient descent use only a partition of the data available at once and reduce the computation time drastically for the cost of a minor inaccuracy that can be

Algorithm 3: The backpropagation algorithm

Input : A loss function ℓ, randomized initial parameter iterative α_0, an error tolerance ϵ, the activation functions $\sigma^{(1)}, \ldots, \sigma^{(L)}$, a training data set $D = (X_{\text{train}}, Y_{\text{train}})$ and a maximum number of iterations $n_{\max}$

Output: Parameters α^* parametrizing the trained FFNN F.

Initialize the network by drawing random initial parameters W_0, b_0
Prepare training data, e.g. scaling and clean-up, and get instances X and labels Y
for $k = 0, 1, \ldots, n_{max}$ **do**
    $M_0 = X$
    for $l = 1, 2, \ldots, L$ **do**
        Compute intermediate network results $M_l \leftarrow f^{(l)} = \sigma^{(l)}\left(W_k^{(l)} M_{l-1} + b_k^{(l)}\right)$.
    end
    Prediction given by last intermediate result $Y_{\text{pred}} \leftarrow M_L$.
    Compute error $E_k = \ell(Y, Y_{\text{pred}})$.
    if $E_k < \epsilon$ **then**
        Stop and return $\alpha^* \leftarrow \alpha_k$.
    else
        Compute gradient of loss function w.r.t. parameters of the last layer $\alpha_k^{(L)}$
        $\frac{\partial \ell}{\partial \alpha_k^{(L)}} \leftarrow \frac{\partial \ell}{\partial f^{(L)}} \frac{\partial f^{(L)}}{\partial \alpha^{(L)}}$.
        for $l = L-1, L-2, \ldots, 1$ **do**
            Compute gradient of loss of every layer going backwards through the network and repeatedly using the chain rule (Lemma 4.5)
            $\frac{\partial \ell}{\partial \alpha_k^{(l)}} \leftarrow \frac{\partial \ell}{\partial f^{(L)}} \frac{\partial f^{(L)}}{\partial f^{(L-1)}} \cdots \frac{\partial f^{(l)}}{\partial f^{(l)}} \frac{\partial f^{(l)}}{\partial \alpha^{(l)}}$
        end
        Set together $s^{(k)} \leftarrow -\nabla_\alpha \ell$ from the layer gradients.
        Determine learning rate γ_k using some step size algorithm.
        Update parameters $\alpha_{k+1} \leftarrow \alpha^{(k)} + \gamma_k s^{(k)}$.
    end
end

handled by reducing the descent's learning rate step by step. Momentum optimization as presented in Algorithm 2 or Nesterov gradient descent allow for a further improvement by taking into account the gradients of previous steps. For reasons that are well-known at this point, the available data should be splitted into training and test data. Using cross-validation or even nested cross-validation is very helpful to find the model that generalizes the best in particular since the optimization process is stochastic and to make use of the parallelization modern CPUs and GPUs offer. Recall that complex models such as high-dimensional polynomial basic models are prone to overfitting. This can be cured by using the regularization methods introduced in Section 3.4.2. A different approach to prevent

overfitting that has not been introduced so far is **dropout**. The idea is to only keep a neuron active with a probability of p, which then becomes another hyperparameter, and otherwise the neuron is inactive and cannot transmit any information. This forces the network to be accurate even if certain information is not forwarded and prevents too much dependence on any neuron. Fundamental theory on dropout methods and the effect on the performance can be found in the foundational work [169].

4.2.2 Training of More Complex Neural Networks

In the previous subsection we introduced backpropagation which is the standard method for training a feedforward neural network. Of course, there exist modifications of this algorithm but Algorithm 3 in combination with the idea of descending the error function serves as the starting point for most methods used in the context of machine learning. In this section which follows Chapter 4 in [31] and Chapter 11 in [60], modified techniques to train neural networks that do not necessarily have to follow a feedforward architecture will be presented.

Let us start with some good news: all neural networks independent of their complexity can be trained using numerical descent methods. For networks with a limited amount of layers and neurons, the techniques presented before serve the purpose of fitting the parameters to form a model that generalizes well quite satisfactionary. On the other hand, deep neural networks require adjustments in the training algorithms. In this work, we focus on quadratic loss functions which are convex and thus ideal for optimization. However, using a network structure with many layers destroys the convexity of the error and potentially makes the optimization prone to running into a local minima or a saddle point which are different from the global minimum. This issue is well-known from optimization and numerical optimization topics and therefore worth mentioning. Fortunately, Dauphin et al. show that gradient descent algorithms and its relatives - in particular stochastic gradient descent - seem to be unlikely to run into and get stuck in a local minimum or a saddle point ([40]) and we therefore do not have to adapt the algorithms. One of the issues that actually plays a major role in applications are vanishing or exploding gradients that we already mentioned in Section 4.2.1 next to how initialization can be used to reduce the risk of running into this problem. Furthermore and once again, regularization is the key tool to prevent overfitting of the network while dropout is useful to achieve a higher degree of generalization and reduce the dependence on single neurons and links in the network.

We will now turn to a short introduction of some other techniques to reduce the generalization error and avoid numerical issues for deep neural networks.

The paper [64] by Glorot and Bengio mentioned before argues that the choice of the activation function can have a severe impact on the optimization process and that the

famous sigmoid function,

$$s(x) := \frac{1}{1 + \exp(-x)} \tag{4.15}$$

for $x \in \mathbb{R}$, is not the best option despite being similar to what is observed in biological neural networks. They propose the usage of the ReLU function that has already been introduced. Further research showed that this function can lead to having many neurons outputting only 0, i.e. no information that could be processed at all. To tackle this issue of **dead** neurons, modifications of the ReLU function have been introduced. One example is the **leaky ReLU** activation function.

Definition 4.6 (Leaky ReLU Function)**.** *For $z \in \mathbb{R}$ and $\alpha > 0$ the function*

$$LeakyReLU_\alpha(z) := \max\{z, \alpha z\} \tag{4.16}$$

is called **leaky ReLU function** *with the* **leaking parameter** α*.*

The factor α, which is often chosen to be 0.01, ensures that the neurons do not die. The performance can be improved further by selecting the leaking parameter α randomly for partitions of data (**randomized leaky ReLU**) or by allowing the algorithm to learn the free hyperparamter α during the process of training (**parametric leaky ReLU**). The different variants of ReLU functions have been evaluated by Xu et al. and indicate that leaky ReLU almost always outperforms standard ReLU ([189]).

Clevert et al. ([37]) propose a new type of activation function called **exponential linear unit (ELU)** that even outperforms all variants of ReLU in the tasks investigated by the authors.

Definition 4.7 (Exponential Linear Unit)**.** *For $z \in \mathbb{R}$ and $\alpha > 0$ the function*

$$ELU_\alpha(z) := \begin{cases} \alpha\left(\exp z - 1\right), & \text{if } z < 0, \\ z, & \text{if } z \geq 0 \end{cases} \tag{4.17}$$

is called **exponential linear unit (ELU)**.

ELU helps to avoid vanishing gradients and dead neurons because of the non-zero gradient while converging faster. For the common choice $\alpha = 1$ the function is smooth for every $z \in \mathbb{R}$. The costs for the improvements is that it requires more computational effort and thus the time needed for testing the network will be longer than for other activation functions. Klambauer et al. introduce an advancement called **scaled ELU (SELU)** which in addition scales dense layers to mean 0 and variance 1 ([100]). This is only guaranteed to outperform the basic ELU when the network is of the feedforward type. Following Géron ([60], p.338), the scheme below describes which activation function should be used.

SELU > ELU > leaky ReLU > ReLU > tanh > logistic

Nevertheless, ReLU is most common in practice because computationally it is the fastest and the activation function of choice if the computational resources at hand are limited.

Figure 4.4 shows plots of the different activation functions. The code to create the plots can be found in the iPython notebook `Plot_Advanced_AF`.

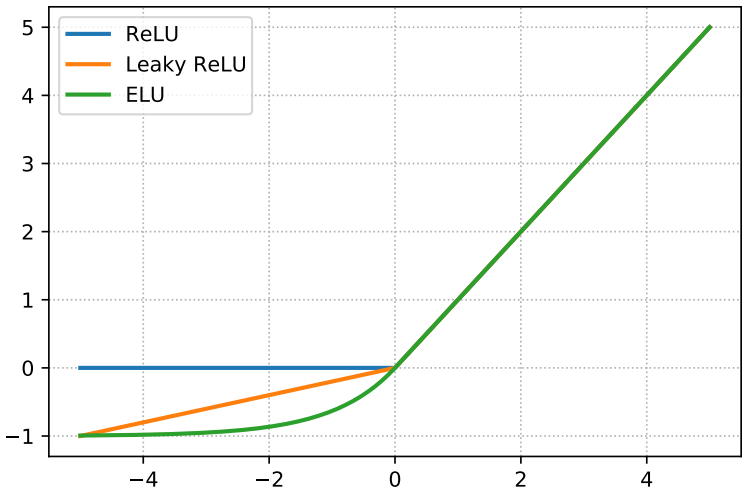


Figure 4.4: Comparison of three advanced nonlinear activation functions: ReLU, Leaky ReLU with $\alpha = 0.2$ and ELU with $\alpha = 1$.

Above, scaling the input data to achieve a better network performance has already been recommended. Together with initialization this reduces the danger of vanishing and exploding gradients at the first layer of neural networks. Ioffe and Szegedy propose **batch normalization (BN)** to address the issue that the former two approaches do not guarantee for proper gradients in deeper layers ([90]). Before or after every layer's activation function, BN centers the data to zero and normalizes it by evaluating the empirical mean and standard deviation and thus provides scaling in every layer. This idea allows the authors to speed up recognition training tasks significantly while actually improving the test accuracy. This comes at the cost of higher computational effort for passing an epoch through the network and a higher amount of parameters that have to be trained but is counterbalanced by faster convergence resulting in shorter training times in total. Adding BN layers has become more or less a convention in deep networks and is often implicitly assumed to be part of the network. However, up to date researchers like Zhang et al. (*fixed-update weight initialization*, [193]) are developing methods offering a similar performance without using BN.

In some types of networks - like recurrent neural networks that will be introduced later - the implementation of BN can be challenging. **Gradient clipping**, introduced by Pascanu

et al. ([130]), allows us to clip the gradients in every layer during the back-pass of backpropagation limiting them to a certain threshold parameter. For example, all partial derivatives can be clipped of at -1 and 1, i.e. a gradient vector $g = \begin{bmatrix} -0.5 & 3 \end{bmatrix}^T$ would be clipped to $\begin{bmatrix} -0.5 & 1 \end{bmatrix}^T$. The threshold can be regarded as a hyperparameter and thus be tuned. It is worth noticing that clipping might change the orientation of the gradient. This issue can be overcome by normalizing the gradient to a vector of length λ which is a hyperparameter. For $\lambda = 1$ the vector g from above would become roughly $\hat{g} = \begin{bmatrix} -0.1644 & 0.9864 \end{bmatrix}^T$.

In general, it is not advised to learn a new deep neural network from scratch for every new task. If there are networks trained for similar tasks available, some of the first layers and their parameters should be used for the new task. However, deeper layers and the output layers should be trained and adapted. This procedure is recommended to reduce the computational time and in general also helps to improve the generalization accuracy. When, due to the publication ([77]) of Hinton et al. which is regarded as the breakthrough of deep learning, research turned again to deep networks in the second half of the noughties, it was common practice to pre-train the layers in advance. **Pre-training** is performed as unsupervised learning for each layer to learn a non-linear transformation of the layer's input to capture the structure of what is transmitted by the previous layer. To do so, one starts with a single-layered unsupervised model, trains it and fixes the parameters before adding the next layer and so on (**greedy layer-wise pre-training**). The pre-training paves the way for the final step, supervised fine-tuning of the network and allows for a better generalization performance. We shall not delve deeper into the theory of pre-training and refer to a paper by Erhan et al. ([55]) which provides a nice overview over the benefits and hints some reasons for the usefulness. Up to date, pre-training can serve as a performance booster if only little training data is available or might be unlabled but advanced machine learning methods like *Autoencoders* allow for more efficient pre-training.

We have now covered the most widely used methods to prepare deep neural networks and how to avoid large-scaled gradients in backpropagation. Furthermore, overfitting can be prevented when adding a regularization term. It turns out that another speed boost can be achieved by further improvement of the optimizer which is the subject of the next subsection and was introduced in [98].

4.2.3 Adaptive Optimization Methods and the Adam Algorithm

Recall that in Section 3.4.2 we already introduced stochastic and mini-batch gradient descent to improve the convergence as well as momentum optimization methods such as the Nesterov accelerated gradient. The algorithm for standard momentum optimization was given in Algorithm 2 and sets the direction of the descent to

$$s^{(i)} := -\nabla \ell_{\alpha^{(i)}} \left(\alpha^{(i)} \right) \tag{4.18}$$

for an iterative set of parameters $\alpha^{(i)}$ and a loss function ℓ. The momentum with hyperparameter δ is then updated by

$$m^{(i)} := \delta m^{(i-1)} - \gamma^{(i)} s^{(i)} \tag{4.19}$$

where $\gamma^{(i)}$ is the step size or learning rate. The new parameter iterate is then compute to be

$$\alpha^{(i+1)} := \alpha^{(i)} + m^{(i)}. \tag{4.20}$$

In NAG the direction of the descent is adjusted to

$$s^{(i)} := -\nabla \ell_{\alpha^{(i)}} \left(\alpha^{(i)} + \delta m^{(i-1)} \right) \tag{4.21}$$

and leads to faster convergence.

So far, we have not focused on the learning rate $\gamma^{(i)}$ at all. We shall now turn to **adaptive learning rates** and **adaptive gradient methods** to deal with what has for a long time been the most troublesome issue of training deep networks. A learning rate that is chosen too high might have problems to converge and if it is too low the process will take very long. But before we introduce adaptive rates and methods, let us take a glimpse on **learning rate scheduling**.

Learning rate schedules vary the learning rate during the training process. From now on, the learning rate at step i of the optimization algorithm will be noted by $\gamma^{(i)}$. An optimal rate can be found by a step size algorithm or by certain rules such as the Armijo rule or the delta rule. Obviously, this requires additional computations and thus capacities and time. A common approach is to reduce $\gamma^{(i)}$ in the course of the training. This can be done **piecewise constant**, e.g.

$$\gamma^{(i)} = \begin{cases} 0.1, & i = 1, \dots, 5 \\ 0.01, & i = 6, \dots, 50 \\ 0.001, & i = 51, \dots, 500 \\ \vdots & \vdots \end{cases} \tag{4.22}$$

or **exponentially**, e.g.

$$\gamma^{(i)} = \gamma^{(0)} 0.1^{\frac{i}{s}} \tag{4.23}$$

which drops by a factor of 10 every s steps or even connected to the **performance** of the algorithm, i.e. when the error stops dropping from iteration to iteration. We refer to a paper by Senior ([157]) for a detailed comparison of the different types of scheduling, who suggests the usage of exponential scheduling due to its implementation simplicity and easy tuning of the hyperparameters like $\gamma^{(0)}$. Nowadays, however, a different type called **1cycle scheduling** introduced by Smith ([164]) is most widely used. Here, the learning rate is increased linearly over the first half of the training and then reduced in the second half. The paper shows that the improvements are magnificient regarding both computation time and accuracy for classification tasks.

Let's focus now on adaptive methods: In practice, we often notice that the direction of the steepest descent does not point to the global minimum. We would prefer if our algorithm would force the direction of descending to adjust and point more towards the minimum. In 2011, Duchi et al. introduced the **AdaGrad algorithm** which corrects the direction of descending by inversely scaling a learning rate for each parameter by the historical gradients of the parameters ([52]). In other words, the algorithm decays the learning rate and does so faster for steep dimensions.

Algorithm 4 gives the exact procedure. In Line 5, $\otimes$ indicates the elementwise multiplication and the fraction in Line 6 should be understood elementwise as well. In this context, the learning rates $\gamma^{(i)}$ and the smoothing rate ε are vectors $\boldsymbol{\gamma}^{(i)}$ and $\boldsymbol{\varepsilon}$ of the length of $\alpha^{(0)}$ and every entry is $\gamma^{(i)}$ or ε respectively. Here, the smoothing rate is a tiny number $\varepsilon \approx 10^{-7}$ or $\varepsilon \approx 10^{-10}$ to avoid division by zero. Often a global learning rate is assumed for AdaGrad, i.e. $\gamma^{(i)} = \gamma$ for all i.

Algorithm 4: The AdaGrad algorithm

Input : A loss function ℓ_α, randomized initial iterative $\alpha^{(0)}$, an error tolerance ϵ, a maximum number of iterations n_{max}, scheduled learning rates $\gamma^{(i)}$ and a smoothing rate ε.

Output: Parameters α^* parametrizing the trained deep NN F.

1 Initialize gradient accumulation vector $r^{(0)} \leftarrow \mathbb{0}$.
2 **for** $i = 1, \ldots, n_{max}$ **do**
3 **if** $|\nabla \ell_{\alpha^{(i)}}| > \epsilon$ **then**
4 Set negative direction of descent $s^{(i)} \leftarrow \nabla \ell_{\alpha^{(i)}}(\alpha^{(i)})$.
5 Accumulate gradients $r^{(i)} \leftarrow r^{(i-1)} + s^{(i)} \otimes s^{(i)}$.
6 Compute new iterate $\alpha^{(i+1)} \leftarrow \alpha^{(i)} - \frac{\boldsymbol{\gamma}^{(i)}}{\sqrt{\boldsymbol{\varepsilon} + r^{(i)}}} s^{(i)}$.
7 **else**
8 Stop and return $\alpha^* \leftarrow \alpha^{(i)}$.
9 **end**
10 **end**

Even though AdaGrad performs well for problems with quadratic error function, it often stops prematurely because of the strong decay of the effective learning rate. As a consequence, AdaGrad should not be used for deep structures but illustrates the basic principle for further, more suitable procedures that will be introduced in the following.

To avoid slowing down too fast like in AdaGrad, an algorithm proposed by Tieleman and Hinton uses exponential decay with a decay rate $\rho \in [0, 1)$, which typically is set to 0.9. ([179]) [1]. This variant is known as the **RMSProp algorithm** and can be interpreted as a moving average approach where we forget about gradients that we observed a long

[1] Even though the algorithm is very popular, the reseachers never published a paper about this algorithm which is why we have to informally refer to Hinton's Cousera class.

time ago. Additionally to the input, we hand the decay rate ρ over to the algorithm and change Line 5 of Algorithm 4 to

$$r^{(i)} \leftarrow \rho r^{(i-1)} + (1-\rho)\, s^{(i)} \otimes s^{(i)}. \tag{4.24}$$

The smaller the decay rate that is chosen, the shorter the memory of previous gradients. RMSProp in combination with momentum strategies almost exclusively outperforms AdaGrad and has been the state-of-the-art for deep NNs for most researchers until the **Adam** or **adaptive moment estimation algorithm** was introduced.

Adam combines the ideas of RMSProp and optimization with momentum by taking into account estimations of the mean and the variance, i.e. the first two moments of the gradient ([98]). Algorithm 5 shows the exact procedure.

Algorithm 5: The Adam algorithm

Input : A loss function ℓ_α, randomized initial iterative $\alpha^{(0)}$, an error tolerance ϵ, a maximum number of iterations n_{max}, scheduled learning rates $\gamma^{(i)}$, a momentum decay rate β_1, a scaling decay rate β_2 and a smoothing rate ε.

Output: Good parameters α^* parametrizing the trained deep NN F.

1 Initialize accumulation vectors $m^{(0)}, r^{(0)} \leftarrow \mathbb{0}$.
2 **for** $i = 1, \ldots, n_{max}$ **do**
3 **if** $|\nabla \ell_{\alpha^{(i)}}| > \epsilon$ **then**
4 Set negative direction of descent $s^{(i)} \leftarrow \nabla \ell_{\alpha^{(i)}}(\alpha^{(i)})$.
5 Update biased average estimate $m^{(i)} \leftarrow \beta_1 m^{(i-1)} + (1-\beta_1)\, s^{(i)}$.
6 Update biased raw variance estimate $r^{(i)} \leftarrow \beta_2 r^{(i-1)} + (1-\beta_2)\, s^{(i)} \otimes s^{(i)}$.
7 Bias correction of average estimate $\hat{m}^{(i)} \leftarrow \frac{m^{(i)}}{1-\beta_1^i}$.
8 Bias correction of raw variance estimate $\hat{r}^{(i)} \leftarrow \frac{r^{(i)}}{1-\beta_2^i}$.
9 Compute new iterate $\alpha^{(i+1)} \leftarrow \alpha^{(i)} - \frac{\boldsymbol{\gamma}^{(i)} \hat{m}^{(i)}}{\sqrt{\boldsymbol{\varepsilon} + \hat{r}^{(i)}}}$.
10 **else**
11 Stop and return $\alpha^* \leftarrow \alpha^{(i)}$.
12 **end**
13 **end**

The algorithm except of the addition of Lines 7 and 8 looks very similar to what we have seen above. The reason why these two lines have been added is that the biased average and the biased raw variance are initialized to 0 in Line 1 and thus biased towards 0. Introducing the two operations supports the algorithm in boosting the iterates $m^{(i)}$ and $r^{(i)}$ away from the bias ([60], p.357). This is one of the main issues of RMSProp and solving this made Adam famous. Typical choices of the additional hyperparameters are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The global learning rate does not require much tuning and can often be set to 0.001, which makes the usage even easier than for GD.

Since the Adam algorithm will be used heavily in later parts of this work, we will explicitly analyze the method and present a convergence result in a specific frame. Unfortunately, we have to limit ourselves to this case since to the best of my knowledge there exists no general convergence proof for Adam and other settings would require a proof beyond the scope of this work. But before we proceed to that, however, it is permissible to make a comment on modifications of Adam. One variant that is already introduced alongside Adam in the paper by Kingma and Ba ([98]) is called **AdaMax** and replaces the ℓ_2-norm by the ℓ_∞-norm for scaling down the parameters. AdaMax sometimes is more stable but in general does not perform better than Adam. The second variant we want to mention is called **Nadam** and projects the idea of Nesterov acceleration on Adam ([50]). It turns out that Nadam generally outperforms Adam. In addition, it should be mentioned that adaptive optimization methods can potentially generalize badly even though they often converge fast to a proper solution ([187]). In this case Nestrov GD is a good alternative.

Other extensions of Adam worth mentioning are among others **AMSGrad** ([142]) providing us with positive convergence results in the convex setting and **YOGI** ([141]) which shares the theoretical convergence results with Adam but allows for an even better performance for several tasks presented in the paper. Even though Adam boosted the training of deep NNs, fine-tuning of Adam-type methods is still a matter of active research. In particular, theoretically guaranteeing the convergence of Adam in a non-convex setting of a deep NN is only possible under heavy assumptions. On the other hand, convergence in a convex setting has already been proven in the paper that introduces Adam ([98]). Reddi et al. ([141]) prove the convergence of Adam for non-convex optimization only in a very restricted setting with $\beta_1 = 0$, a global learning rate and increasing size of mini-batches over time.

Here, we will present a convergence result for the convex setting which guarantees a convergence speed of $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$ for all $N \in \mathbb{N}$. Proving the result completely is tedious and out of the scope of this work which is why we will refer to a reference for the proof of an auxiliary lemma.

Before we come to a convergence statement, we introduce the notations $g^{(i)} := \nabla \ell_{\alpha^{(i)}}$ and collect the element-wise values of the gradient step by step, i.e. $g_j^{(1:i)} := \left(g_j^{(1)}, \dots, g_j^{(i)}\right)^T \in \mathbb{R}^i$ for a row indicator j. Moreover, we have to define a measure of the error which computes the difference between the optimal ℓ_{α^*} and current value of ℓ_α.

Definition 4.8 (Error Sum)**.** *Let α^* be the optimal set of parameters, i.e. $\alpha^* = \arg\min_{\alpha \in A} \sum_{i=1}^N \ell_{\alpha^{(i)}}$ with A being the set of parameters which will arise in the Adam algorithm. The* **error sum** *at the N-th step is then defined by*

$$R(N) := \sum_{i=1}^{N} \left(\ell_{\alpha^{(i)}} - \ell_{\alpha^{(i)}}\right). \tag{4.25}$$

Let us now turn to an auxilary lemma that provides us with an upper bound for $R(N)$. From now on, the loss function ℓ_α is assumed to be convex for all αs that *can* be generated using the Adam algorithm. Note that this is a strong assumption and convex loss functions are rare in most applications.

Lemma 4.9 (Upper Bound for Error Sum)**.** *Let $g^{(i)}$ be bounded with $\|g^{(i)}\|_2 \leq G$ and $\|g^{(i)}\|_\infty \leq G_\infty$ for all $i \in \{1, \dots, N\}$ and some real-valued bounds G and G_∞. Moreover, let the difference between two weights generated in the process be bounded, i.e. $\|\alpha_n - \alpha_m\|_2 \leq D$ and $\|\alpha_n - \alpha_m\|_\infty \leq D_\infty$ for all $n \neq m$, $n, m \in \{1, \dots, N\}$ and some real-valued bounds D and D_∞. Also assume that $\beta_1, \beta_2 \in (0,1)$, $\xi := \frac{\beta_1^2}{\sqrt{\beta}} < 1$ and that the learning rate is scheduled to be $\gamma^{(i)} := \frac{\gamma}{\sqrt{i}}$ for some basic learning rate $\gamma \in (0,1)$. Here, let the momentum decay rate be given by $\beta_1^{(i)} := \beta_1 \lambda^{i-1}$ for some scaler $\lambda \in (0,1)$.*

Then the error sum of the Adam optimization algorithm can be estimated by

$$\begin{aligned} R(N) \leq & \frac{D_\infty^2}{2\gamma\,(1-\beta_1)} \sum_{i=1}^{d} \sqrt{N \hat{r}_N^{(i)}} + \frac{d D_\infty^2 G_\infty}{2\gamma\,(1-\beta_1)\,(1-\lambda)^2} \\ & + \frac{\gamma\,(\beta_1+1)}{(1-\beta_1)\sqrt{1-\beta_2}(1-\xi)} \sum_{i=1}^{d} \left\| g_i^{(1:N)} \right\|_2, \end{aligned} \tag{4.26}$$

where d is the dimensionality of the gradient.

Proof: This result was originally stated and proven in Theorem 4.1 in [98]. However, the reader is referred to [20] for an improved version of the proof. □

We are now in the position to state and prove a convergence result which is taken from [98] and [20].

Corollary 4.10 (Convergence Rate of Adam)**.** *Let $g^{(i)}$ be bounded with $\|g^{(i)}\|_2 \leq G$ and $\|g^{(i)}\|_\infty \leq G_\infty$ for all $i \in \{1, \dots, N\}$, some real-valued bounds G and G_∞ and all possible sets of parameters $\alpha^{(i)} \in \mathbb{R}^d$. Moreover, suppose that the difference between two weights generated in the process be bounded, i.e. $\|\alpha_i - \alpha_j\|_2 \leq D$ and $\|\alpha_i - \alpha_j\|_\infty \leq D_\infty$ for all $n \neq m$, $n, m \in \{1, \dots, N\}$ and some real-valued bounds D and D_∞. Furthermore, assume that the assumptions on the algorithm parameters from 4.9 hold.*

Then for all $N \geq 1$ the following convergence estimate for the Adam optimization algorithm holds:

$$\frac{R(N)}{N} = \mathcal{O}\left(\frac{1}{\sqrt{N}}\right). \tag{4.27}$$

Proof: The requirements of Lemma 4.9 are fulfilled by the assumptions. Thus, we can

use Equation 4.26 and since $N > 0$ divide it by N.

$$\begin{aligned}\frac{R(N)}{N} \leq & \frac{D_\infty^2}{2\gamma\,(1-\beta_1)} \sum_{i=1}^{d} \frac{\sqrt{\hat{r}_N^{(i)}}}{\sqrt{N}} + \frac{d D_\infty^2 G_\infty}{N 2\gamma\,(1-\beta_1)\,(1-\lambda)^2} \\ & + \frac{\gamma\,(\beta_1+1)}{N\,(1-\beta_1)\,\sqrt{1-\beta_2}(1-\xi)} \sum_{i=1}^{d} \left\| g_i^{(1:N)} \right\|_2 .\end{aligned} \tag{4.28}$$

Let us now look for upper bounds for the sums

$$\sum_{i=1}^{d} \left\| g_i^{(1:N)} \right\|_2 \quad \text{and} \quad \sum_{i=1}^{d} \frac{\sqrt{\hat{r}_N^{(i)}}}{\sqrt{N}}. \tag{4.29}$$

For the primer sum, we apply the definition of the Euclidian norm, $g_i^{(1:N)}$ and make use of the fact that $g^{(i)}$ is bounded for all $i \in \{1, \ldots, N\}$:

$$\begin{aligned}\sum_{i=1}^{d} \left\| g_i^{(1:N)} \right\|_2 &= \sum_{i=1}^{d} \sqrt{(g_j^{(1)})^2 + \cdots + (g_j^{(N)})^2} \\ &\leq \sum_{i=1}^{d} \sqrt{G_\infty^2 + \cdots + G_\infty^2} \\ &= \sum_{i=1}^{d} \sqrt{N} G_\infty = d\sqrt{N} G_\infty.\end{aligned} \tag{4.30}$$

Before we turn to the second sum, we need to find an upper bound for the jth component of the corrected raw variance estimate $\hat{r}^{(i)}$ at the i-th step. To determine it, we make use of the updates of $r^{(i)}$ and $\hat{r}^{(i)}$ defined in Lines 6 and 8 of Algorithm 5 and the fact that $g^{(i)}$ is bounded.

$$
\begin{aligned}
\sqrt{\hat{r}_j^{(i)}} = \left(\frac{r_j^{(i)}}{1-\beta_2^i}\right)^{1/2} &= \left(\frac{\beta_2 r_j^{(i-1)} + (1-\beta_2)\left(g_j^{(i)}\right)^2}{1-\beta_2^i}\right)^{1/2} \\
&\overset{(\dots)}{=} \sqrt{1-\beta_2}\left(\frac{\sum_{k=1}^{i}(g_j^{(i)})^2\beta_2^{i-k}}{1-\beta_2^i}\right)^{1/2} \\
&\leq G_\infty\sqrt{1-\beta_2}\left(\frac{\sum_{k=1}^{i}\beta_2^{i-k}}{1-\beta_2^i}\right)^{1/2} \\
&\leq G_\infty\sqrt{1-\beta_2}\left(\frac{\sum_{k=1}^{i}\beta_2^{k}}{1-\beta_2^i}\right)^{1/2} \\
&\leq G_\infty\sqrt{1-\beta_2}\left(\frac{1-\beta_2^i}{(1-\beta_2^i)(1-\beta_2)}\right)^{1/2} \\
&\leq G_\infty
\end{aligned}
\tag{4.31}
$$

for all $j \in \{1, \dots, d\}$ and $i \in \{1, \dots, N\}$. With this estimate, we can proceed to finding an upper boundary for $\sum_{i=1}^{d}\left(\frac{\hat{r}_N^{(i)}}{N}\right)^{1/2}$. For this, we proceed as follows

$$
\sum_{j=1}^{d}\sqrt{N\hat{r}_j^{(N)}} \leq \sum_{j=1}^{d}\sqrt{T}G_\infty \leq dG_\infty\sqrt{T}. \tag{4.32}
$$

Dividing both sides of the equation by N yields

$$
\sum_{j=1}^{d}\left(\frac{\hat{r}_j^{(N)}}{T}\right)^{1/2} \leq \frac{dG_\infty}{\sqrt{T}}. \tag{4.33}
$$

Finally, further estimating inequality 4.28 by inserting (4.30) and (4.33) as well as taking the limit, we end up with

$$
\lim_{N\to\infty}\frac{R(N)}{N} \leq \lim_{N\to\infty}\left(\frac{1}{\sqrt{N}} + \frac{1}{N} + \frac{1}{\sqrt{N}}\right) = 0. \tag{4.34}
$$

This completes the proof. □

This corollary proves the convergence of the Adam algorithm in a convex setting under strong assumptions and a convergence speed of $\mathcal{O}\left(1/\sqrt{N}\right)$. We refer to Zou et al. ([195]) and Chen et al. ([34]) for convergence proofs of Adam-type optimizers in settings different to the one presented here.

After presenting a lot of theory, we want to summarize what is known about different optimization techniques, in particular about their convergence speed and quality, before we leave the field of optimization theory and turn to an applied example. Table 4.1, which follows and extends Table 11-2 in [60], gives an informal overview over this properties.

Table 4.1: A comparison of different optimization techniques.

Optimizer	Convergence Speed	Convergence Quality
SGD	-	+
SGD with Momentum	○	+
NAG with Momentum	○	+
AdaGrad	+	-
RMSProp	+	○/+
Adam	+	○/+
Nadam	+	○/+
AdaMax	+	○/+
AMSGrad	+	○/+
YOGI	+	○/+

4.3 Example: Feedforward Neural Networks

In this section, a simple example of a feedforward neural network (FFNN) is presented. A network of this type is used to predict the absolute log return of the US stock index S&P500 given the observations of the last ten days as an input. The period of interest begins on 1 January 2005 and ends on 1 October 2021, resulting in a total of 4217 observations. This task is similar to the one in Section 2.4 when fitting an ARIMA-GARCH model. It can be classified as univariate supervised learning with a regression task and batch learning approach since the data is labeled and there is no new data flowing in continuously. The features are the close prices of the previous ten days and the labels the close price of the current day. Like all neural networks it is discriminative. The FFNN used in this example consists of two hidden layers with 64 neurons in the first and 32 neurons in the second hidden layer. The input layer has dimension 10 while the output layer is of dimension 1. For both hidden layers, the activation function used in this example is the ReLU function.

We use the Python library TensorFlow which includes the deep learning API Keras and offers the basic architecture to create neural networks in Python. Again, the financial market data is downloaded from Yahoo!Finance and the code can be found in the iPython notebook `FFNN_for_FTS` in the appendix. The training data used are the first 80% of the data downloaded.

The procedure of fitting the FFNN to the market data can be summarised as follows:

1. Download of the market data,
2. Preparation of data by extracting the absolute log returns on the close prices and creation of a sliding window of past observations which serves as the input for the network,

3. Scaling of data,
4. Split data into training and test datasets,
5. Creation of a model: FFNN with an input layer consisting of 10 neurons, hidden layers with 64 and 32 neurons as well as an output layer with a single neuron,
6. Fit the model using Keras' fitting tool,
7. Evaluate fit for both training and test data, and
8. Plot comparison of actual data and predictions and the absolute prediction error.

Figure 4.5 compares the actual data observed in the financial markets and the predictions generated by the fitted FFNN and also plots the absolute error (bottom). The dashed line indicates the end of the training data set.

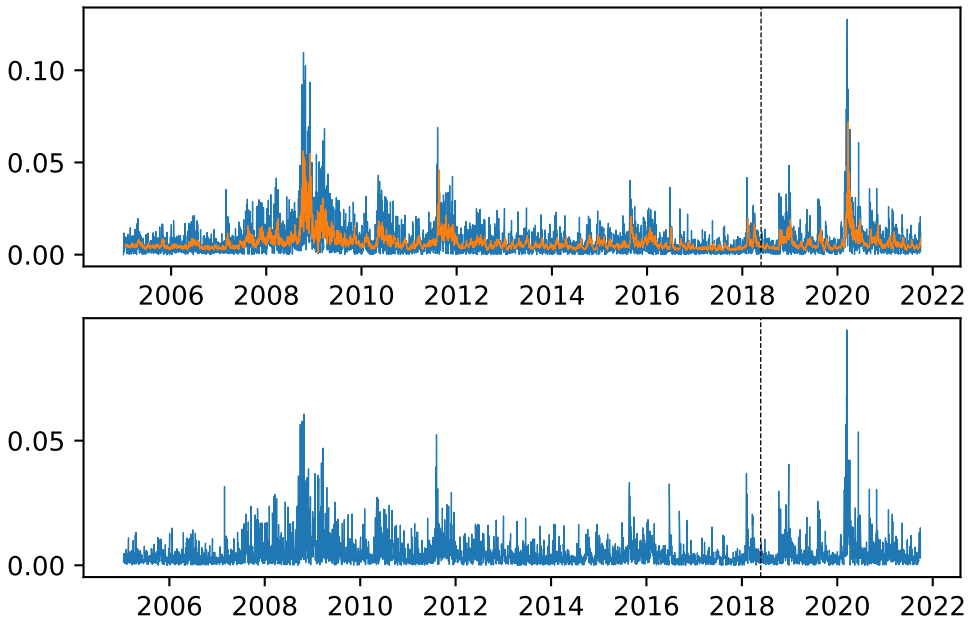


Figure 4.5: (Top) Prediction versus actual observations of absolute returns of S&P500 close prices using a FFNN with two hidden layers (64 and 32 neurons) and an input consisting of the last 10 observations. (Bottom) Absolute error of the prediction.

The root mean-squared error (RMSE) of the training data is $7.175 \cdot 10^{-3}$ while the error for the test data is $9.269 \cdot 10^{-3}$. It is plausible that the error is slightly higher for the test data set but it is not very far off indicating that the model neither over- nor underfits the data. However, both errors are high and and would have a significant

impact if applied in real world markets. In addition, we find that the absolute prediction error is highest for extreme returns and conclude that FFNNs are incapable of forecasting extreme events but useful for the detection of trends. Nevertheless, the test error is lower than the RMSEs for the ARIMA-GARCH model, $1.794 \cdot 10^{-2}$, and dynamic exponential smoothing, $1.453 \cdot 10^{-2}$. In addition, there are many ways to tune the performance of the model. Some of these tuning methods will be discussed and implemented in Section 6.1. Furthermore, it would possible to perform further hyperparameter tuning and cross-validation to find the best model.

It is important to note that in practical applications, feedforward neural networks are not used to predict financial markets since they implicitly assume iid observations. There exist other network structures such as *recurrent neural networks (RNNs)*, presented on the right in Figure 4.3, which provide a significantly better fit and prediction power. These networks will be presented in detail and implemented in Chapter 5 of this work.

4.4 Summary

Artificial neural networks are inspired by biological neural networks and the building blocks are similar. Neurons transform given input signals into a new signal that is handed over to the next layer. How the new signal is computed depends on the activation function used and we have seen that there is a variety of such functions of which ReLU is used often. The most simple form of a NN which only forwards information through the network is called a feedforward neural network. Even though the universal representation theorem states that a FFNN with one hidden layer can approximate any continuous function, it is very useful and efficient to have multiple hidden layers, i.e. deep NNs.

The training of FFNNs builds on the chain rule of differentiation to determine the gradient with regard to the different network weights/parameters and is known as backpropagation. In practice, initialization and automatic differentiation have proven to speed up the process. Networks more complex than the ones of feedforward type can be trained using the AdaGrad or the different versions and extensions of the Adam algorithm. For complex networks, strategies to further improve the performance like batch normalization, pre-training, gradient clipping or learning rate scheduling have been recommended.

To put the findings and strategies into practice, an example to predict the absolute log returns of close prices of the S&P500 index using a FFNN has been presented. The results show that the network outperformed the ARIMA-GARCH model and exponential smoothing. It is worth to keep in mind that in practice FFNN are not the network type of choice to predict future closing prices. Instead, more complex networks with smoothing, memory functions and gates are used which will be introduced in Chapter 5.

5 Advanced Neural Networks for Financial Time Series

In the previous chapter, artificial neural networks and several training and boosting methods have been introduced. We are now interested in networks more complex than the ones of the feedforward type which allow for a better performance when predicting asset prices in financial markets. In addition, wide single-layered networks usually require a lot of parameters while adding multiple layers reduces the amount of required parameters and thus makes the network less prone to overfitting. All networks presented in this chapter are discriminative models of supervised machine learning. First, we will turn to networks modeling stationary time series such as recurrent and convolutional neural networks. In this context, the connection to econometric models introduces in Chapter 2 will be pointed out. Unfortunately, these types of neural networks prove to have a rather short memory and are not suitable for the time series that indicate a dependence on long past observations. In what is Section 5.2, we will turn to networks that have a longer memory than the RNNs and CNNs and are able to take long-term effects into account (long short-term memory and gated recurrent units). To conclude the chapter, the reader will be introduced to transformer networks specialized for financial time-series which is able to handle large data sets as well as the shortcomings of LSTM and GRU.

Before we introduce our first **advanced neural network (ANN)** it is worth mentioning that the networks presented in this chapter can be used in a vast variety of applications such as voice recognition or image processing. Forecasting of financial time series is just a small niche of what is possible with the unlimited creativity of neural network designs. More often than not, the networks were developed for other applications and only later transferred and adapted to financial time series. Moreover, for small sequences or limited amounts of data, regular feedforward neural networks can be sufficient and even show a better generalization performance than advanced neural networks in applications. The network designs in this chapter have been developed for predicting tasks that take a large amount of input data.

This chapter builds on CHapter 8 in [48] that initially motivated the author to write this thesis.. Chapter 15 in [60] is another bubbling source of information about sequence processing with advanced neural networks. The interested reader will find a broad and rather informal overview of the variety of neural network design on this website ([183]).

Of course, there exist many other network designs for applications in financial markets

but we will focus on a limited amount of design classes for supervised learning.

5.1 Recurrent Neural Networks (RNN)

Recall that if the data $\mathcal{D}$ is autocorrelated observations of the inputs and outputs over time, predictions can be made taking lagged observations as the input of a non-linear predictor whose parameters are to be determined in the process of the training. Furthermore, we have already learned that stationary time series are of great advantage since stationarity means that the series shows the same statistical behaviour for all periods of time and thus predictions of future developments can be made on a strong basis. In this section, the architecture and mathematics of *recurrent neural networks* and its modifications will be introduced.

5.1.1 Basic Recurrent Neural Networks (RNN)

So far, the spotlight has been on feedforward neural networks in which the information, i.e. the activation signals, is only forward-passed through the network starting at the input layer and ending at the output layer. **Recurrent neural networks (RNNs)**, originally introduced in the context of machine learning in 1986 ([150]), differ in that an activation signal z_{t-1} depending on the output y_{t-1} of the previous time step, some weights and a bias, is sent back to the neuron itself and thus the input at time t consists of the standard input x_t and activation signals z_t from the previous time step $t-1$. Of course, these activation signals recursively depend on even earlier time steps and the lag h gives the maximum number of past time points that are relevant for the recurrent neuron at the current point of time. For the very first time step $t = 0$ under consideration, there is no output from previous time steps and thus the input only consists of x_{t-h} and $z_{t-h-1} = 0$. Neurons of this type are called **recurrent neurons** which can be regarded as another set of hidden neurons over time. A sketch of the architecture of a RNN is shown in the left part of Figure 5.1, where recurrent neurons are marked by gray arcs. The dependency on previous time steps is shown in the right part of the figure. One says that the RNN is an **unfolding** or **unrolling** through time of hidden neurons and layers.

It is worth mentioning that for layers of recurrent neurons with more than one neuron, both the standard input x_t as well as the information from the output of previous time step z_{t-1} are vectors. In addition, there are two new sets of weights for the information from previous steps. Here, $w^{(l)}_{z,t^*}$ denotes the set of weights associated with the new second hidden layer's outputs z_{t^*} at previous time steps $t^* = t-h, \ldots, t-1$ while $w^{(l)}_{x,t^*}$ denotes the weights associated to previous network inputs x_{t^*}. Moreover, the weights of a layer l are collected in the weight matrices $W^{(l)}_x = \left(w^{(l)}_{x,t-h}, \ldots, w^{(l)}_{x,t-1}\right)$, $W^{(l)}_z = \left(w^{(l)}_{z,t-h}, \ldots, w^{(l)}_{z,t-1}\right)$ and $\tilde{W}^{(l)} = \left(W^{(l)}_x, W^{(l)}_z\right)$. Analogous to FFNNs there also

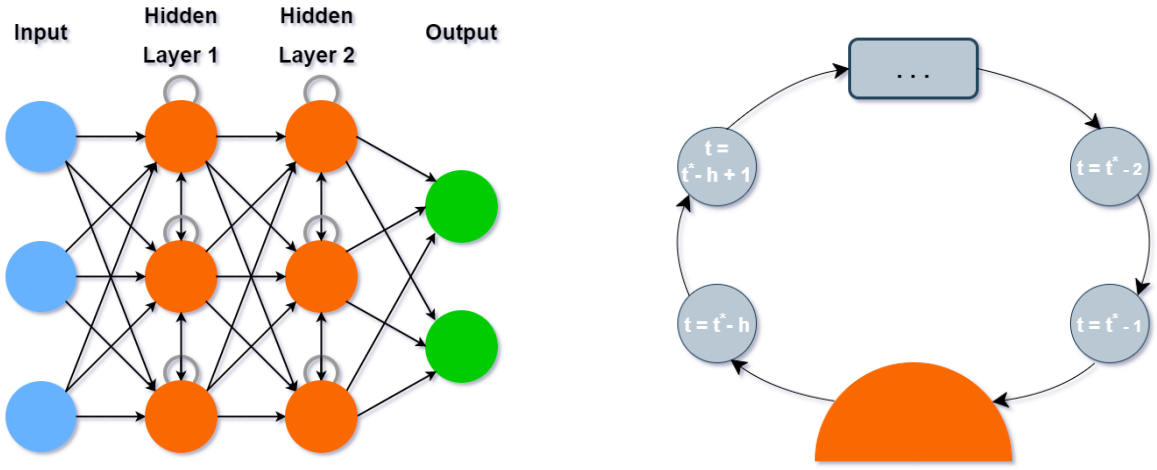


Figure 5.1: (Left) Architecture of a recurrent neural network with a single hidden layer following [183], where gray arcs above the neuron indicate recurrent neurons. (Right) Recurrent neuron structure at time $t = t^*$.

exist the standard weight matrix W and the standard bias b.

We are now in the position to formally define what a RNN is.

Definition 5.1 (Recurrent Neural Network)**.** *Let $\mathcal{D} = (X, Y)$ be real-valued and labeled data, where the input X can be imagined as a list of vectors x_t. A neural network F with L layers is called a* **recurrent neural network (RNN)***, if the mapping F is given by*

$$Y \approx F_{W,b}(X) := \left(f^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}} \right)(X), \tag{5.1}$$

where the activation signals of the hidden states at time t and for a second activation function σ_z are a function of the current layer input and the activation signals of the hidden states at one time step before. Formally, they are determined by

$$z_t^{(l)} := z_t^{(l)}(x_t^{(l)}) := \sigma_z^{(l)} \left(W_z^{(l)} z_{t-1}^{(l)} + W_x^{(l)} x_t^{(l)} + b_z^{(l)} \right) \tag{5.2}$$

and the response at time t by

$$f^{(l)}_{W^{(l)},b^{(l)}}(x_t^{(l)}) := y_t^{(l)} := \sigma^{(l)} \left(W^{(l)} z_t^{(l)} + b^{(l)} \right), \tag{5.3}$$

both for all network layers $l = 1, \ldots, L$ and for $z_t^{(l)} := z_t^{(l)}(x_t^{(l)})$. The weight matrices and biases are shared temporally and the activation functions do not necessarily need to be identical. Moreover, applying the functions to a (mini-)batch X can be done by applying $f^{(l)}_{W^{(l)},b^{(l)}}$ to every instance of X for layers $l = 1, \ldots, L$.

In the above definition, the dependence of z_t from x_t and z_{t-1} is pointed out. Due to the recursive approach of RNNs, this directly implies the dependence of z_t from $x_t, \ldots, x_{t-h}$. Again, it is assumed that the dimensions of the matrices and biases are appropriate so that the matrix multiplications are well-defined and this will be assumed for all other neural networks unless explicitly stated otherwise.

In the context of RNNs, most researchers like Vu Pham et al. ([135]) prefer the hyperbolic tangent activation function for σ_z rather than the ReLU. The latter one can also be chosen (Le et al., [107]) and the main reason is to avoid vanishing gradients. Furthermore, σ is often chosen to be either a softmax activation function or the identity (cf. [48], p.242).

Recurrent neurons compute their outputs depending on the information from previous time steps which is why we say that they have a **memory** and a layer of recurrent neurons is called a **memory cell**. This memory typically goes back about ten time steps depending on the task. With this, we are satisfied for the time being but we will introduce other network designs in Section 5.2 that allow for a dependency on longer past information. In addition to limited memory, other notable issues are higher computational effort due to the need of optimizing even more parameters, the exact number of hidden layers and neurons as well as the inability of the network to take future inputs that might be already known into account. While the computational efficiency can be increased using methods introduced in previous chapters, the bias-variance tradeoff is the most important consideration when determining the quantity of hidden layers and neurons. Beyond these issues, the user has to deal with the question how many times the network should be unfolded through time, i.e. how many past time steps are relevant for the computation of the activation signal at the current point of time. One possible approach to a solution is testing for autocorrelation using the methods presented in Section 2.2.

On the other hand, among others some advantages of RNNs are the possibility to process inputs of any length, the recycling of historical computations and information as well as the time-invariance of the weight matrices. We shall later see in Chapter 6 how this network type performs for predictions of financial time series in comparison to other designs.

RNNs are flexible in which input they take and which output they produce. If the network is unfolded only once through time, i.e. the layer input and output are each a single vector, the RNN is simply a FFNN. Contrarily, if it is unfolded multiple times, the network takes a sequence as its input. A sequence can be a sequence of stock prices or a linguistic sentence. Like the input, the output can be either a vector or a sequence. Table 5.1 gives an overview of different RNN types and Figure 5.2 illustrates the architecture of each type. Missing inputs in the figure indicate that it is equal to 0 and missing outputs are discarded outputs.

The **encoder-decoder design** basically is a sequence-to-vector network followed by a vector-to-sequence network. Encoder-decoders have proven to be much better in lan-

Table 5.1: Different designs and applications of RNNs.

Network Type	Application
Sequence-to-sequence	Stock price prediction based on previous observations
Sequence-to-vector	Sentiment classification (e.g. of a review)
Vector-to-sequence	Creation of a caption for an image, music generation
Encoder-decoder	Translation of texts into different language

guage translation than word-by-word translations and the full text translater DeepL is a great example of a practical implementation.

Recurrent neural networks can be regarded as a generalization of the ARIMA model. The following remark, which is derived from Example 8.1 in [48], shows how a simple RNN is in fact a generalized AR(p) model:

Remark 5.2. *Let us consider a RNN with a single hidden layer with one hidden neuron and a one-dimensional real-valued input. Moreover, we choose both activation functions to be the identity and set $b_z = 0$. For this network, the weight matrices reduce to scalars and we assume $W_x = 1$ and $|W_z| < 1$. For reasons of notational simplicity we dropped the layer indicator in the exponent since we only have one hidden layer.*

From Definition 5.1, we get

$$\begin{aligned} z_{t-h-1} &= 0 \\ z_{t-h} &= W_x x_{t-h} \\ z_{t-h+1} &= W_z z_{t-h} + W_x x_{t-h+1} \\ \vdots \quad &= \quad \vdots \\ z_t &= W_z z_{t-1} + W_x x_t \end{aligned} \tag{5.4}$$

with $W_x = 1$ and for the response

$$y_t = W z_t + b. \tag{5.5}$$

If we plug in z_t from (5.4) in (5.5) and iterate backwards through time, we obtain

$$\begin{aligned} y_t &= W z_t + b \\ &= W\left(W_z z_{t-1} + x_t\right) + b \\ &= W\left(W_z\left(W_z z_{t-2} + x_{t-1}\right) + x_t\right) + b \\ &= \quad \vdots \\ &= W\left(x_t + W_z x_{t-1} + W_z^2 x_{t-2} + \cdots + W_z^{h-1} x_{t-h}\right) + b \\ &= \sum_{i=1}^{h} W W_z^{i-1} x_{t-i} + W x_t + b. \end{aligned} \tag{5.6}$$

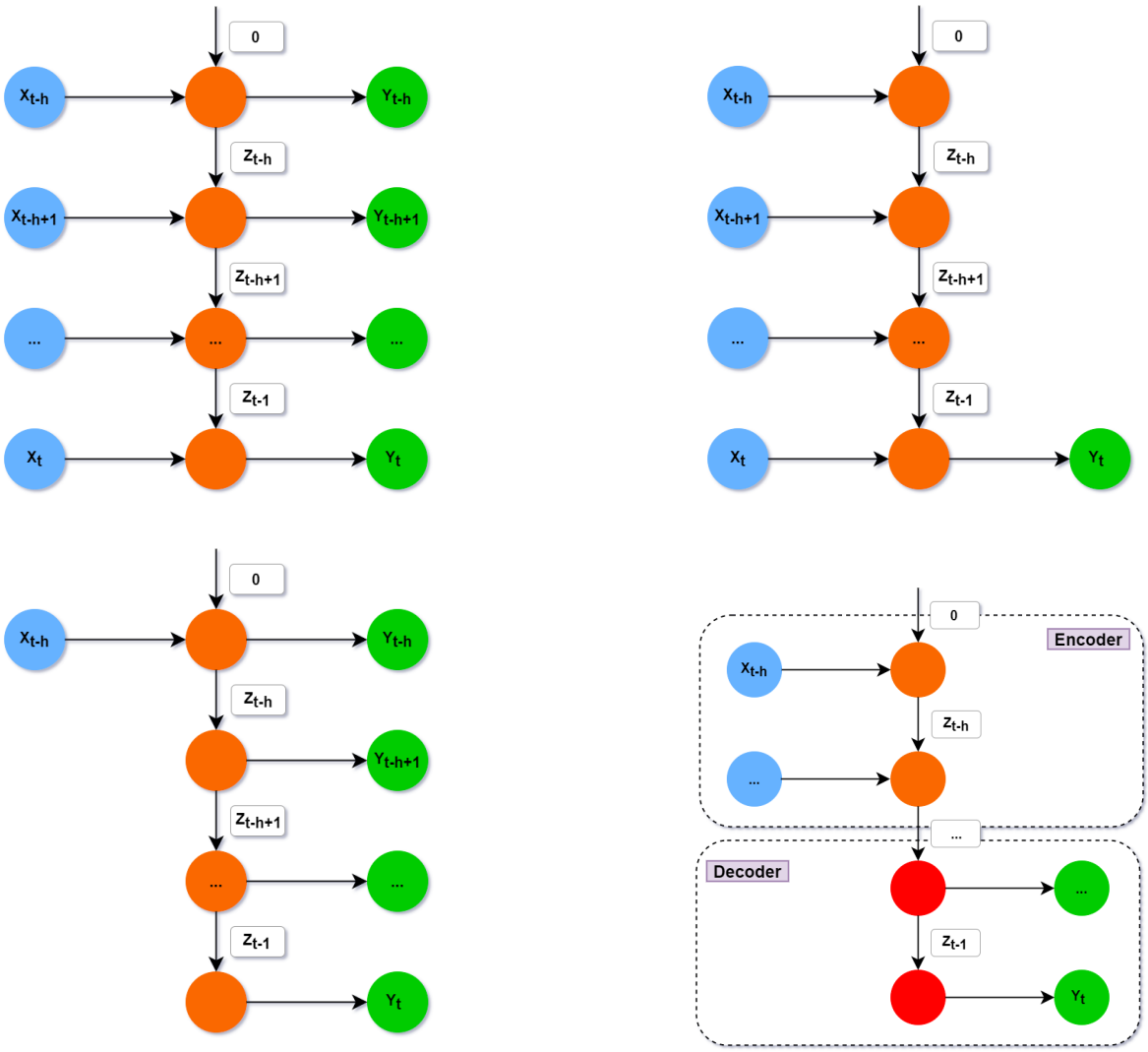


Figure 5.2: Different types of recurrent neural networks unrolled over time. (Top left) Sequence-to-sequence, (top right) sequence-to-vector, (bottom left) vector-to sequence and (bottom right) encoder-decoder network. Illustrations inspired by [9].

By setting $\Phi_i := WW_z^{i-1}$ we find the structure of an AR(p) process with $p = h$ as defined in Equation 2.7 plus an additional term Wx_t. The order can be identified through tests for autocorrelation, e.g. Ljung-Box. It is also possible to generalize the findings from above to lag-dependent weights allowing for more flexibility. Moreover, we can recover a result from the analysis of AR processes. It can be shown that a linear RNN with an infinite number of lags and no bias corresponds to exponential smoothing with

$$z_t = \alpha y_t + (1 - \alpha) z_{t-1}, \tag{5.7}$$

if $W_z = 1 - \alpha$, $W = \alpha$ *and* $W_x = 1$ *([48], p.243).*

Generally speaking, RNNs build on ARIMA models and generalize them. For ARIMA models, it is often required to first remove the trends and seasonalities to increase the efficiency of the fitted model. When using a RNN this is not required but it can improve the performance in some cases because the learning process is simpler with previously identified trends and seasonalities.

For models with non-linear activation functions, a general simple description of a RNN cannot be given. Nevertheless, the autocorrelation and partial autocorrelation functions provide an insight. The latter one has a cut-off at lag h and is independent of time (Section 8.2.1 in [48]). This allows us to identify the order of a RNN model using the estimated PACF in a similar fashion then we did for ARIMA models.

For some further insights into stability and stationarity analysis, that will not be analyzed here, the interested reader is referred to Sections 8.2.2 and 8.2.3 in [48].

So far, we have not specified how to train RNNs which contain two hidden structures: the normal hidden layers and the hidden layers to capture the information from previous pass-throughs. The training process consists of unrolling the RNN through time and then perform regular backpropagation. This is called **backpropagation through time (BPTT)**. We start with a forward pass of inputs through the unrolled network and evaluate the output vector or sequence using a loss function $\ell(Y, \hat{Y})$ which might ignore some outputs like in a sequence-to-vector model. The gradients of the loss function with regard to the weights are then determined in a backward pass, where the gradients do not necessarily have to "flow" through every output and the weights are updated using this gradients and a learning rate. All methods we have learned so far to improve the performance of backpropagation can be applied here as well. Since the weights and biases are time-invariant, backpropagation performs correctly and adds up over all time steps. When using TensorFlow and Keras, the packages automatically take care of this additional complexity and we do not have to take it into further consideration.

When training a RNN, unlike FFNNs, the network is able to deal with any number of previous time steps taken as the input but the dimension of the input tensor has to be fixed in the first layer of the model. One of the major advantages of RNNs over simple FFNNs is that the number of parameters is potentially significantly lower. On the other hand, in most applications and in particular forecasting of financial time series RNNs with a single hidden layer perform worse than high-dimensional FFNNs. For this reason, deep RNNs should be considered for prediction tasks, in particular if the goal is to predict several time steps ahead into the future. A deep RNN can be created by simply stacking layers of recurrent neurons. When implementing such a network using Keras, it is important to set `return_sequences = True` to ensure the correct handling of the activation signals produced by the hidden neurons when recurring. A trick to boost the network performance, convergence and flexibility is to use a standard Dense layer as the final output layer ([48], p.507f.).

When the intention is to predict several time steps into the future, it is recommendable to not simply train a RNN, but to make it predict the next time step, add the prediction to input values, predict the next time step and so on. Dixon et al. show in Chapter 15 of [48], that for an exemplary time series the model performs significantly better if the prediction task is adjusted to predicting the next N time steps at every point of time and train the network with regard to this time horizon. In other words, this means turning to a sequence-to-sequence model and even though the training takes longer, the result is much better than for simple NNs or the sequence-to-vector RNN. Even though these rather simple RNNs are quite good at forecasting time series, they run into problems when being forced to handle long time series.

In Section 4.2.2, the problem of vanishing or exploding gradients in deep neural networks has already been mentioned and addressed. This issue is even more pronounced for deep RNNs since the training requires to unroll the networks through time and thus creates a very deep network. To improve the training, we can apply some of the methods introduced in the course of Chapter 4:

- Parameter initialization,
- Improved optimization techniques such as Adam,
- Gradient clipping,
- Dropout,
- Saturating activation functions such as tanh,
- Batch normalization,
- And many more.

For RNNs, batch normalization only provides us with a small benefit when applied between the recurrent layers ([106]). **Layer normalization** is a concept similar to batch normalization that has been introduced by Lei Ba et al. in 2016 ([13]). Here, the normalization takes place across the features dimension instead of across the batch dimension, and this approach has proven to often perform better with recurrent-type networks. The issue of the RNN's limited memory of only a few time steps will be dealt with in Section 5.2, where we introduce network structures with gates and extended long-term memory.

5.1.2 Generalized Recurrent Neural Networks (GRNN)

As stated above, RNNs can be understood as a generalization of the ARIMA model. There is a modified approach called **generalized recurrent neural networks (GRNNs)** that generalizes this idea analogous to how the GARCH model generalizes the ARIMA model. The striking difference is that classical RNNs threat the error as iid, in other words RNNs follow a homoscedastic modeling approach. By modifying the loss function, we can generalize these networks to allow for heteroscedasticity. This aproach is still a

matter of research for applications in econometrics and will therefore not be discussed in detail here. However, the underlying concept will be presented briefly.

The concept is based on solving a modified minimization problem of the least squares-type:

$$\min_{W,b} L_{W,b} + \lambda\Phi(W, b). \tag{5.8}$$

Here, λ is a parameter and Φ is a penalty term, both for regularization. The function L is defined by

$$L(W, b) := \frac{1}{T}\sum_{t=1}^{T} \mathcal{L}_\Sigma(y_t, \hat{y}_t), \tag{5.9}$$

where y_t are the actual labels, $\hat{y}_t$ the predictions made and the loss function $\mathcal{L}_\Sigma$ is defined by

$$\mathcal{L}_\Sigma(Y, \hat{Y}) := (Y - \hat{Y})^T \Sigma^{-1} (Y - \hat{Y}). \tag{5.10}$$

Σ is the conditional covariance matrix of the residual and this matrix has to be estimated from the given data.

In contrast to GARCH, GRNNs solely rely on the empirical error distribution and do not make forecasts of future conditional volatilities. Moreover, this generalization is not implemented yet in Keras and we shall not use this approach when comparing different forecasting methods in Chapter 6.

Since the application to financial time series is not the focus, there is relatively little literature on this type of network. We refer to [93] for an example of a work that establishes a connection between ARMA-GARCH models and RNNs, builiding a RNN-adapted non-linear ARMA-GARCH model.

In the following, we proceed in the extension of RNNs by turning to smoothing techniques similar to what we have already seen in the theory and the analysis of time series.

5.1.3 α- and α_t-Recurrent Neural Networks (α-RNN, α_t-RNN)

We start by introducing a smoothed version of RNNs called **α-recurrent neural network (α-RNN)**. The modifaction compared to basic RNNs takes place by adding a smoothing parameter $\alpha \in [0, 1]$. This parameter guarantees for a longer memory of the network, which indeed is infinite for $\alpha \neq 1$. Formally, when computating the signals of the network, we adjust (5.2) in Definition 5.1 to

$$\begin{aligned} z_t^{(l)}(x_t^{(l)}) &:= \sigma_z^{(l)}\left(W_z^{(l)}\tilde{z}_{t-1}^{(l)} + W_x^{(l)}x_t^{(l)} + b_z^{(l)}\right) \\ \tilde{z}_t^{(l)} &:= \alpha z_t^{(l)} + (1-\alpha)\,\tilde{z}_{t-1}^{(l)}. \end{aligned} \tag{5.11}$$

To guarantee that this is well-defined, we initially set $\tilde{z}_{t-h}^{(l)} = y_{t-h}^{(l)}$. Chossing $\alpha = 1$ results in a standard RNN with limited memory of length h. Furthermore, when training

an α-RNN, we can treat the smoothing parameter α as yet another hyperparameter that can be learned and optimized.

Choosing a fixed α for all time steps t obviously limits us to stationary time series. Fortunately, there exists a dynamic version of exponential smoothing: **α_t-recurrent neural networks (α_t-RNN)**. Unlike for α-RNN, the smoothing parameters α_t are time-dependent and applied to the outputs $y_t^{(l)}$ instead of the hidden vectors to generate a smoothed output denoted by $\tilde{y}_t^{(l)}$. The convex combination for the smoothing factor α_t is given by

$$\tilde{y}_t^{(l)} := \alpha_t y_{t-1}^{(l)} + (1 - \alpha_t)\, \tilde{y}_{t-1}^{(l)} \tag{5.12}$$

for every layer. For $\alpha_t = 0$ the model ignores the forecasting error, repeats the current hidden state and the memory is lost. Contrarily, if $\alpha_{t-k} = 1$ the current hidden state activation signals are ignored at and beyond the lag k.

The class of α_t-RNNs is free in how we define the smoothing parameters α_t. One convenient option to determine the parameters is to use a recurrent layer or a complete RNN. To do so, we turn to a slightly more general setup where the smoothing is applied on the hidden state signals and contains a smoothing parameter $\hat{\alpha}_t \in [0, 1]^H$, where H is the depth of the recurrent layer. A filtered time series is then given by

$$\tilde{z}_t := \hat{\alpha}_t \circ z_{t-1} + (\mathbb{1} - \hat{\alpha}_t) \circ \tilde{z}_{t-1}, \tag{5.13}$$

where $\mathbb{1}$ is the H-dimensional vector consisting of ones and $\circ$ denotes the elementwise Hadamard product of vectors. This form of smoothing, known as **exponential smoothing**, can be understood as a vectorized form of what has been presented before in Section 2.1.4. The parameters can be determined as the outputs of a standard plain RNN, G, with weights W_G and biases b_G given the current input x_t, i.e. $\hat{\alpha}_t = G_{W_G, b_G}(x_t)$. The process of determining the smoothing parameters can be interpreted as learning the necessary error correction rate.

When comparing the three different RNN models introduced, one finds that standard RNNs lose their memory after a few time steps whereas α-RNN show longer and smoother memory. In addition, α_t-RNNs enable us to fit a model to non-stationary data ([47], pp.22-23).

We will not go into further depth, but refer the interested reader to [49] and [47] for more detailed insights into the theory of α- and α_t-RNNs. In general, there exist two approaches how to extend the memory of a recurrent neural network which differ from smoothing approaches. One of them is called **dilation** and will be introduced in Section 5.3.4 in the context of a new class of networks. The second approach is based on *gating* and will be discussed in the next section.

5.2 Models with Long-Term Memory and Gating

In the previous section, we mentioned that the main issues of RNNs are potentially unstable gradients due to high dimensionality and the rather short-term memory caused by the loss of information when the data is sent through the network. The primer one can be satisfactionally cured by a combination of several methods presented in Chapter 4 including dropout or layer normalization among others. To be able to extend the memory of neural networks of the recurrent type, we will now turn to two improved RNN structures called **long short-term memory (LSTM)** networks and a simplified version with growing popularity called **gated recurrent units (GRU)**. Recall that the idea of smoothing has already been introduced in Section 5.1 and LSTMs will generalize this idea. Here, we present an extended abstract of the theory that can be found in Chapter 15 of [60] and Section 8.3 in [48].

5.2.1 Long Short-Term Memory Networks (LSTM)

When introducing neural networks in Chapter 4, the main components were the neurons and the connections between them which correspond to the biological neurons and the dendrites in a biological network. For advanced NNs we have seen that the neurons can contain inner hidden states, e.g. including information from previous time steps in RNNs, and that activation functions, as well as smoothing, play an important role. With reference to the biology, the combination of these buliding blocks for a single neuron is called a **cell**. A cell takes some input x_t combines it with any existing inner hidden states z_t which can be thought of as the **short-term state** and *somehow* generates and output y_t. Here and through the entire section, we dropped the indication (l) of the layer for reasons of simplicity of notation. This basic design is illustrated in the left part of Figure 5.3. The matter of this section is how to design these cells to provide the whole network with a long-term memory. Fortunately, the implementation of different cells in Keras is straightforward and they can be treated as black boxes. However, our initial motivation was to avoid black boxes which is why we will take a detailed look at the interior of the cell.

In 1997, the **long short-term memory (LSTM)** cell design was proposed by Hochreiter and Schmidhuber ([80]) and later improved by numerous reseachers such as Sak et al. who added a recurrent projection layer for improved vocabulary speech recognition ([152]). LSTM-based designs have proven to be very successful in diverse application tasks, e.g. the forecasting of electric loads in grids (Zheng et al., [194]), due to the faster convergence and improved detection of long-term patterns.

Let us now venture a look inside the cell which is illustrated in Figure 5.3 and find out what allows the LSTM to provide the network with a longer memory. The inner hidden state z_{t-1} as well as the newly added **long-term state** denoted by c_{t-1}, serve as an input

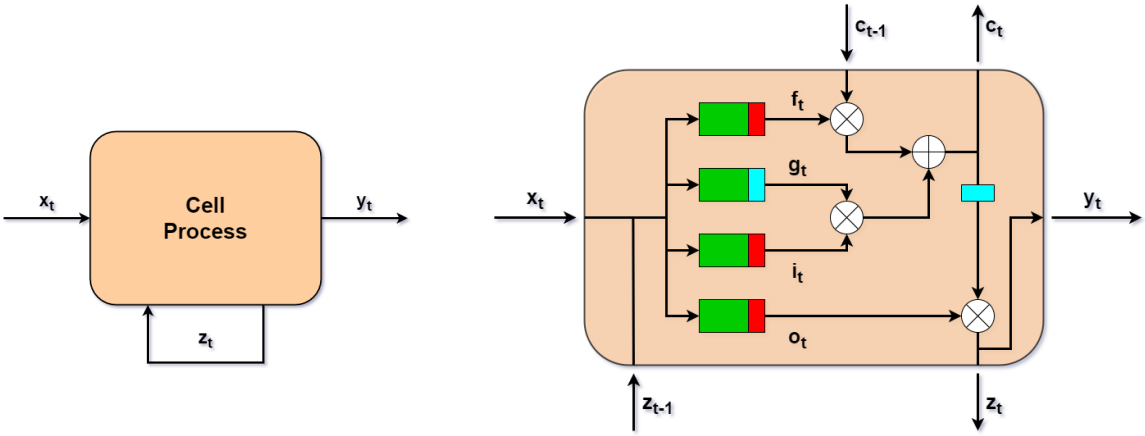


Figure 5.3: Cell designs of (left) a basic cell and (right) a LSTM cell. The right figure is inspired by Figure 15-9 in [60].

together with the standard input x_t. In total, the cell produces an output y_t, a short-term state z_t and a new long-term state c_t.

First, the inputs x_t and z_{t-1} are fed to four different fully-connected layers (green boxes in Figure 5.3). The main layer which is also present in basic cells, is the one generating g_t by analyzing its inputs and applying an activation function σ_g, usually the hyperbolic tangent tanh (denoted by light blue rectangle in the figure). In addition to basic cells, LSTM cells contain three additional layers called **gate controllers** which in combination with a sigmoid activation function σ_{sig} (red boxes) produce outputs f_t, i_t and o_t each ranging from 0 to 1. All of these outputs are connected to gates via a Hadamard product indicated by $\otimes$. The **input gate** controlled by i_t specifies which information of the newly generated g_t is important and should be added to the long-term state while the **forget gate** influenced by f_t determines which information is no longer relevant and should be erased from c_{t-1}. Adding the signals exiting from these two gates provides us with a new and modified long-term state c_t which is sent out of the cell. The third gate called **output gate** is controlled by o_t and selects which information from the long-term state that has been passed through some activation function σ_{out}, usually tanh, is handed over to the output y_t and the short-term state z_t respectively.

This procedure at a time step t for a single instance x_t can be summarized in mathematical

terms by

$$g_t := \sigma_g\left(W_{xg}x_t + W_{zg}z_{t-1} + b_g\right), \tag{5.14}$$
$$i_t := \sigma_{\text{sig}}\left(W_{xi}x_t + W_{zi}z_{t-1} + b_i\right), \tag{5.15}$$
$$f_t := \sigma_{\text{sig}}\left(W_{xf}x_t + W_{zf}z_{t-1} + b_f\right), \tag{5.16}$$
$$o_t := \sigma_{\text{sig}}\left(W_{xo}x_t + W_{zo}z_{t-1} + b_o\right), \tag{5.17}$$
$$c_t := f_t \otimes c_{t-1} + i_t \otimes g_t, \tag{5.18}$$
$$y_t := z_t := o_t \otimes \sigma_{\text{out}}\left(c_t\right). \tag{5.19}$$

The matrices $W_{xg}, W_{xi}, W_{xf}, W_{xo}$ and $W_{zg}, W_{zi}, W_{zf}, W_{zo}$ are the weight matrices for each of the four fully connected layers in connection with x_t and z_{t-1} respectively. Moreover, the vectors b_g, b_i, b_f, b_o are the corresponding bias terms of the layers where b_f is usally set to a vector of ones to prevent excessive forgetfulness at the beginning of the training. Since the gate controllers range from 0 to 1 it is possible to overwrite the different memory states and the particular choice of $f_t = 0$ makes the network not taking the long-term memory into account. Contrarily, if i_t the cell will not be interested in currently generated signals.

There exist numerous variants of LSTM cells and one possible extension is to allow the short-term state z_{t-1} to peek at the long-term state through additional connections to give the short-term memory some additional context information ([61]). These connections are called **peephole connections** and improve the performance in some settings. Alternatively, improved LSTM cells which reduce the impact of bias on the pattern detection are helpful when e.g. forecasting the wind power output ([72]). However, arguably the most popular LSTM-based cell is the **gated recurrent unit (GRU)** even though it is a simplified version.

5.2.2 Gated Recurrent Units (GRU)

Gated recurrent units have been introduced by Cho et al. in a rich 2014 paper ([35]) which additionally suggested the usage of Encoder-Decoder networks, presented in Figure 5.2. A GRU cell simplifies the LSTM cell by merging the short-term state and the long-term state into a single vector z_t while offering a similar performance ([67]). Moreover, this cell type only contains a single gate controller f_t for both the forget and the input gate which can be regarded as a sensitivity parameter of the smoothed inner hidden state z_{t-1} to the input x_t. The primer gate is controlled by f_t and the latter gate by $1 - f_t$. Contrarily to LSTM cells, an output cell is not included in GRU cells. As a consequence the complete state vector is output in every time step. However, an additional gate controller r_t selects which part of the information in z_{t-1} is handed over to the main fully-connected layer which generates g_t.

This procedure is illustrated in Figure 5.4 where green boxes represent the fully connected layers, the light blue box the hyperbolic tangent activation function and the red boxes

indicate the application of the logistic activation function.

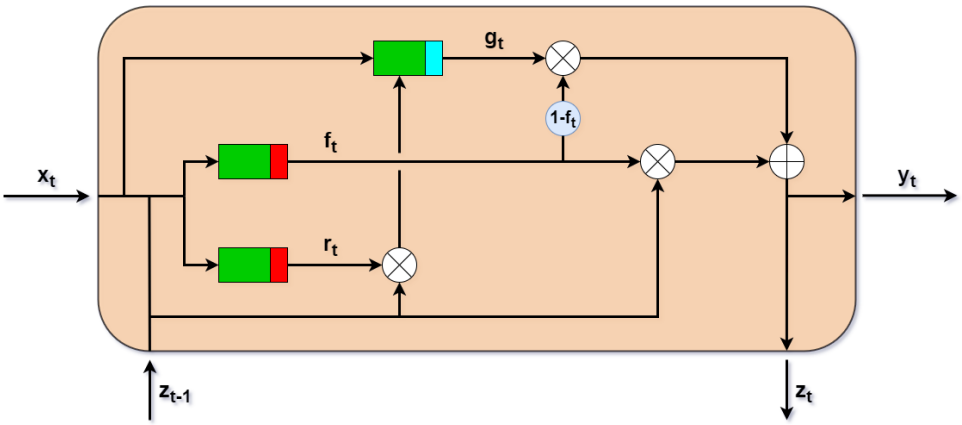


Figure 5.4: Cell designs of a GRU cell. The illustration is inspired by Figure 15-10 in [60].

Again, we can express the inside actions of the cell with mathematical equations:

$$r_t := \sigma_{\text{sig}}\left(W_{xr}x_t + W_{zr}z_{t-1} + b_r\right), \tag{5.20}$$
$$f_t := \sigma_{\text{sig}}\left(W_{xf}x_t + W_{zf}z_{t-1} + b_f\right), \tag{5.21}$$
$$g_t := \sigma_g\left(W_{xg}x_t + W_{zg}\left(r_t \otimes z_{t-1}\right) + b_g\right), \tag{5.22}$$
$$y_t := z_t := f_t \otimes z_{t-1} + (1 - z_{t-1}) \otimes g_t. \tag{5.23}$$

The matrices W_{xr}, W_{xf}, W_{xg} and W_{zr}, W_{zf}, W_{zg} are the weight matrices for each of the three fully connected layers associated with x_t and z_{t-1} respectively. Moreover, the vectors b_r, b_f, b_g are the corresponding bias terms of the layers and $\otimes$ denotes the Hadamard product of vectors.

Similar to LSTM, a GRU is capable of being a plain RNN or a FFNN for a certain choice of architecture and thus generalizes both network types. The additional layers allow GRU networks to learn long and complex dynamics. This is at the expense of increased requirement of computational resources and effort. Taking this into account, it is advisable to evaluate in advance whether a simpler model such as α_t-RNN would provide a sufficient generalization performance.

However, even though RNNs designed with LSTM and GRU cells generate improved network memory, they are limited and struggle when learning patterns in sequences of 100 or more time steps which are usual for long time series such as audio recordings. This issue can be tackled by adding *convolutional layers* to the network which apply filters on the sequences serving as the input and thus shorten the sequence by dropping unimportant details (WaveNet, [125]). This layer type will be in the focus of the next section and the idea of compressing the sequences can be taken further by making use of

dilated convolution that will be presented in Section 5.3.4. For processing audio and text sequences, the introduction of convolutional layers and dilation offered a performance boost and we shall see in Chapter 6 if this holds for financial time series as well.

5.3 Convolutional Neural Networks (CNN)

In this section, which is a wrap-up of the theory presented in Chapter 5 of [31], Chapter 14 in [60] and Section 8.5 in [48], we turn to an advanced neural network structure which aims at exploiting local spatial structures in the data given as the input. In other words, these networks called **convolutional neural networks (CNNs)** offer an architecture that reduces the complexity a FFNN would require for flattening or smoothing high-dimensional time series data. Thus, CNNs are able to detect specific patterns and exploit dependencies in the data. Before we turn in more detail to the usage in the context of financial time series, let us take a look at the history of the development of CNNs, which is representative of the interdisciplinarity of many innovations in deep learning.

5.3.1 Historical Development of CNNs and Basic Network Design

Originally, CNNs have been introduced and studied in the field of computer vision. They are inspired by our understanding of human and animal vision and build on the results of an experimental series analysing the visual cortex of cats and monkeys ([87], [88] and [89]). The authors David H. Hubel and Torsten Wiesel received the Nobel Prize in Medicine or Physiology in 1981 for their groundbreaking work. Their main findings were that many neurons in the brain specialized for vision only react to stimulation from a limited region of the visual field while the fields of different neurons may overlap. For example, some neurons react only to horizontal lines while others detect different orientations. This observation led to the idea that complex recognition tasks are based on the recognition of basic patterns by lower-level neurons. The outputs of lower-level neurons are processed in subsequent neuron layers allowing the brain to detect complex visual patterns.

With the increase of interest into neural networks and the breakthrough in training methods for deep structures, many researchers developed machine learning methods trying to detect patterns, objects and faces in images. Examples are the pioneering work by Paul Viola and Michael Jones ([186]) as well as the benchmark setting work by Perronnin et al. ([133]). By 2011, the leading algorithm by Perronnin et al. for face recognition tasks still had an error rate of about 25%. One year later, Alex Krizhevsky introduced an innovative network structure, called convolutional neural networks, which led to a major improvement in computer vision tasks with a reduced error rate of about 19% ([102]). In contrast to FFNNs, CNNs build on **convolutional layers** in which neurons are not connected to every pixel of the input data but only to these in their respective **receptive field**. In the following layers, the small features are step-by-step assembled

into larger and more complex features. Inspired by human vision, the CNN's layers are three-dimensional, having a width, height and depth. The network then applies so called **filters** which are sets of weights, detect simple patterns and create **feature maps** which highlight the areas that activate the filter the most. By combining the information from different feature maps and processing through multiple layers, the network is able to detect even complex structures like a human face. Feature maps tend to be very high-dimensional which is why **pooling layers** have been introduced to shrink the complexity and sharpen the features. Another upside is that the number of parameters is reduced significantly and thus overfitting is limited. Pooling layers usually take several input values for a visual field but only output a single value, e.g. the maximum value or the mean. Mean pooling layers are arguably the most popular version by now. The costs of the reduction of complexity is that some level of **invariance** to small translation appears, i.e. for similar but slightly different inputs the outputs of a pooling layer might be identical. On one hand, this can be very destructive but for some classification tasks it has shown to be helpful if the task is not focusing on the details.

CNNs only pass information forward and typically, every few convolutional layers are followed by a pooling layer which again is followed by several convolutional layers and so on. The image gets smaller and smaller through this process but also deeper and deeper (recall that the layers are three-dimensional in this setting). At the end of the network, a standard FFNN composed of fully connected layers with an activation function is added before ending with an output layer that usually supports classification tasks, e.g. a softmax layer. Figure 5.5 shows a typical design of a CNN.

Typically, the convolutional layers are followed by a ReLU activation layer before the signal is passed to the pooling layer. Tiny twists in the architecture can have a massive impact on the result, e.g. the accuracy rate in computer vision tasks. Major advances in computer vision have been achieved in recent years. Since we are focusing on forecasting financial time series, we will only sketch the progress made in broad outlines and give references for the interested reader.

5.3.2 Network Architectures for CNNs in Computer Vision Tasks

The ILSVRC ImageNet challenge and the progress of its winner's error rate are a nice indicator for the development in CNNs for recognizing objects and people in large images. A graphical overview over the improvement as well as many works that participated can be found on this website ([129]). Before the challenge was introduced for the first time in 2011, the **LeNet-5** architecture by LeCun et al. for hand-written digit recognition ([108]) was the state-of-the-art and still is widely known. LeNet-5 uses an architecture very similar to the one shown in Figure 5.5 but the hyperbolic tangent has been chosen as the activation function and average pooling was applied. In 2012, the much larger and deeper **AlexNet** by Alex Krizhevsky ([102]) dominated the challenge by using ReLU activation, stacking several convolutional layers when being in the deep network before

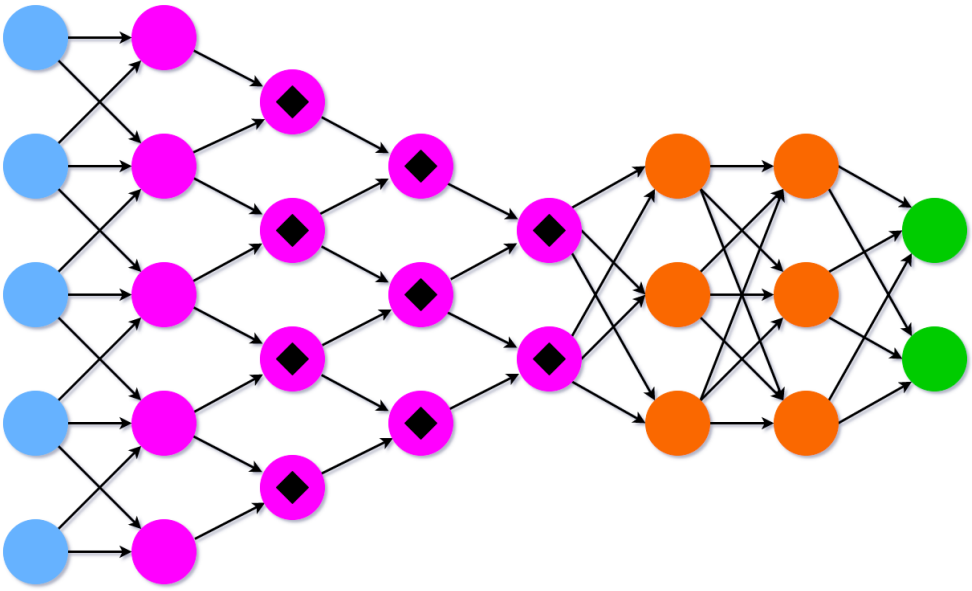


Figure 5.5: An example of the architecture of a CNN. Blue represents the input neurons, magenta the filters, magenta with a rhombus the convolution neurons followed by a pooling layer, orange fully connected neurons and green the output neurons. The illustration is inspired by [183].

proceeding with max pooling. Moreover, dropout, changes of the lighting conditions or flipping of the images in the process of training the network have been used as regularization techniques. In a next step that led to winning the 2014 challenge a research team from Google Research introduced **GoogLeNet** ([175]). The improvement was made possible by creating a much deeper network and sub-networks or **inception modules** which allow for a more effective use and the reduction of the quantity of parameters in the network. In 2015, a **residual network (ResNet)** design won the competition ([75]). In addition to designing an even deeper network, the creators used **skip** or **shortcut connections**. These connections additionally add the output signal to a much deeper layer and therefore allow for what is called **residual learning** which speeds up the training and increases the simplicity of the network. A well-performing example of a network architecture combining the ideas of GoogLeNet and ResNet is **Xception** ([36]). According to the website mentioned above and as of October 2021, the design with the highest top-5 accuracy is **EfficientNet-L2** which uses a scaling method that uniformly scales all dimensions using a *compound coefficient*. The variant presented by Pham et al. ([134]) shows an error rate of about 1.2% which demonstrates that great improvement has been made in computer vision (AlexNet in 2011 had a top-5 error rate of about 18.9%).

For top-1 accuracy, the best performance was achieved by combining convolution with a Transformer-type network (**CoAtNet-7**, [39]) of which the latter one will be studied in

Section 5.5.

A nice demonstration of how far computer vision has come, can be found in a paper of Redmon and Farhadi who use an improved **You Only Look Once (YOLO)** architecture for real-time object detection in videos ([143]). An example video of the network's real time performance can be found here.

We now turn to a more mathematical and numerical view on CNNs in a subsection that is inspired by Section 8.5 in [48] and Chapter 14 in [60].

5.3.3 Mathematics of CNNs

We already learned that CNNs can exploit local spatial structures in data given as the input. Their goal is to reduce the network size and this is achieved by the use of convolution and filters. For (financial) time series, convolution is useful to reduce the noise and takes place by applying a particular class of smoothers.

Definition 5.3 (Weighted Moving Average Smoothers)**.** *A* **weighted moving average (WMA) smoother** *for the local mean at time t of a time series $(X_s)_{s\in I}$ is given by*

$$\chi_t := \frac{1}{\sum_{j\in J}\omega_j}\sum_{j\in J}\omega_j X_{t-j}. \tag{5.24}$$

The coefficients $(\omega_j)_{j\in J}$ are weights to emphasize or deemphasize past observations of the time series.

This class of smoothers takes $|J|$ observations as the input and produce a single output (weighted average). The greater $|J|$, the smoother the output becomes and the less accentuations remain visible. In fact, we have already seen an example of a smoother of this type when turning to an exponential smoother while discussing a RNN with one hidden layer and neuron in Remark 5.2. Another example is the **Hanning smoother** $h(3)$ defined as

$$\chi_t := \frac{X_{t-1} + 2X_t + X_{t+1}}{4}. \tag{5.25}$$

More generally speaking, in a univariate setting the filtered time series is given by

$$\chi_{t-i} := \begin{cases} \sum_{j\in J=\{-k,\dots,k\}} K_{j+k+1} X_{t-i-j}, & i \in \{k+1,\dots,p-k\}, \\ X_{t-i}, & i \in \{1,\dots,k,p-k-1,\dots,p\}, \end{cases} \tag{5.26}$$

where the smoother or filter - also known as the **kernel** - is given by K. Using a notation similar to what has been introduced in Definition 5.1 for RNNs, we can give a formal definition of a convolutional layer.

Definition 5.4 (Output of a Convolutional Layer)**.** *For a convolutional layer l in a neural network with L layers, an univariate time series $X = (X_t)_{t\in I}$, a weight matrix $W^{(l)}$, biases $b^{(l)}$ and $b_\chi^{(l)}$, activation functions $\sigma^{(l)}$ and $\sigma_\chi^{(l)}$, a kernel K and a lag p, the output $y_t^{(l)}$ is given by*

$$y_t^{(l)} := \sigma^{(l)}\left(W^{(l)} z_t^{(l)} + b^{(l)}\right), \tag{5.27}$$

where

$$z_t^{(l)} := (\hat{\chi}_{t-1}, \dots, \hat{\chi}_{t-p})^T \tag{5.28}$$

and

$$\hat{\chi}_{t-i} := \sigma_\chi^{(l)}\left(\chi_{t-i} + b_\chi^{(l)}\right), \tag{5.29}$$

for $i = 1, \dots, p$ with χ_{t-i} given by Equation 5.26.

Similar to RNNs, the weight matrices and biases are time-invariant and thus CNNs are best suitable for stationary time series. Moreover, the size of the weight matrices increase if we increase the number of lags p or the number of kernels applied one after another. This leads to more complex networks which require more computational effort while being trained. Adding additional layers can reduce the dimensionality and hence avoid the well-known issue of overfitting. In deeper networks, it is common to immediately stack several convolutional layers before handing the signals to a pooling layer which allows us to achieve richer representations. On the other hand, deep CNNs need a large amount of memory, in particular for the backward pass of the training procedure (e.g. [162]).

To accelerate the training and make the generalization more stable, batch normalization is also used for CNNs ([90]). The normalization prevents significant shifts in the distribution of the inputs for every layer and thus increases the learning rate. Moreover, it acts like a regularizer and thus we do not have to additionally implement dropout or explicit regularization ([31], p.106).

Furthermore and in contrast to what we have been looking at so far, CNNs can even be extended to non-sequential models that ignore some of the intermediate lagged observations by e.g. setting the index $I := \{2^i\}_{i=1}^p$ which gives a maximum lag 2^p.

A very simple form of a CNN reminds us of a model that we have seen before:

Example 5.5. *For a one-dimensional CNN with a feedforward output layer and a single hidden layer containing one hidden neuron whose activation function is the identity, the output looks like*

$$y_t := W z_t + b, \tag{5.30}$$

where we dropped the layer indicator for reasons of simplicity. If in addition the weight matrix is given by a vector $W := (\Phi_1, \dots, \Phi_p)^T$ and the bias by $b := \epsilon$, we recognize an $AR(p)$ model (Equation 2.7) for a filtered or smoothed time series.

Many of the ideas and concepts presented above can be understood better in the case of 2D convolution which can be applied to image processing, among other things. For now, imagine an image whose pixels' gray-scaled values are collected in a matrix $M \in [-1,1]^{m\times n}$ where the entry m_{ij} refers to the gray scale ranging from -1 to 1 of the pixel in the ith row and jth column of the image. Note that for a colored image, we would face a matrix $N \in \mathbb{R}^{m\times n\times 3}$ with an additional dimension for the RGB values.

As described above, the neurons of a convolutional layer are only connected to their receptive fields, usually a small rectangle, and not to all neurons in the previous layer, i.e. a convolutional layer is not fully-connected to its predecessor. The convolution or pattern detection is then performed by moving a kernel $K \in \mathbb{R}^{o\times p}$ with $o < m$ and $p < n$ over the image matrix M. The kernel can be understood as the matrix containing the weights which are the subject of optimization for every layer. For all possible $(o \times p)$-sized clippings $\tilde{M}$ of M we compute the **feature map** $Y = (y_{i,j})_{i,j}$ for the next layer by Equation 5.26 modified for two dimensions which is

$$y_{i,j} := \sum_{a,b\in J=\{-k,\dots,k\}} K_{a+k+1,b+k+1} m_{i+a+1,j+b+1} \tag{5.31}$$

for $i \in \{1,\dots,m\}$ and $j \in \{1,\dots,n\}$.

Applying this over several consecutive layers allows the network to detect complex features and structures. If it is desired to have the same dimensions as in the previous layer, **padding** is applied. Padding increases the size of the image on the outerior edges. This can be done by periodic continuation or by reproduction of the values on the image edges. By far the most common approach is to add zeros around the initial image and this technique is called **zero padding**. In contrast, it is also possible to reduce the image size and thus the dimensionality by shifting the receptive fields by more than one step. The shifting step size is called **stride** and the stride is 1 if we want to have the same dimension in every layer. On the other hand, a stride of 2, which is a common choice, allows us to reduce a 5×7 layer expanded by padding to a 3×4 layer.

Every few layers a pooling layer is built into a CNN. This type of layer shrinks the input image to reduce the computational effort and the number of parameters required. They are similar to convolutional layers since each neuron is connected to a receptive filed in the previous layer. Instead of applying weights, pooling neurons use an aggregation function G such as the maximum or the mean. The pooling kernel size fixes the receptive field size and the stride once again tells us about the shift. It is also possible to apply padding to extend the size of the image in the output of the previous layer. The **pooling map** $P = (p_{i,j})_{i,j}$ is then computed by

$$p_{i,j} := G\left(\tilde{M}\right). \tag{5.32}$$

In order to gain a better understanding, we will now turn to an example of 2D convolution.

Example 5.6. *Let's assume that we have an input image M of size 4×6 that is about to be processed through a CNN with one convolutional layer followed by a pooling layer*

before being handed to a fully connected layer. Let the image be converted into gray scale $[-1, 1]$ *and then the image can be represented by a matrix* M_g*, which we assume to look like this:*

$$M_g = \begin{bmatrix} 0.9 & -0.1 & 0.1 & 0.0 & -0.5 & 0.2 \\ 0.1 & -0.2 & 0.8 & -0.4 & 0.8 & 0.4 \\ -0.3 & 0.4 & -0.2 & -0.7 & -0.8 & 0.2 \\ 0.5 & -0.2 & -0.7 & -0.8 & -0.5 & -0.7 \end{bmatrix}. \tag{5.33}$$

We expand the image by zero padding and then apply a kernel K *onto the* 3×3 *receptive fields. In this example we choose* K *to be defined by*

$$\begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}. \tag{5.34}$$

This is just one of many possible of a kernel design and the entries of the matrix are treated as parameters in the process of training a CNN. The kernel given in Equation 5.34 detects top horizontal edges and displays them as the brightest pixels. This convolutional layer outputs a 4×6 *matrix* N *which, thanks to zero padding, is of the same size as the input. An illustration of the procedure is shown in the left part of Figure 5.6 and applying the formula from Equation 5.31, the value* $n_{1,6}$ *is computed to be*

$$\begin{aligned} n_{1,6} =& (-1) \cdot 0 + (-1) \cdot 0 + (-1) \cdot 0 + 1 \cdot (-0.5) \\ &+ 1 \cdot 0.2 + 1 \cdot 0 + 0 \cdot 0.8 + 0 \cdot 0.4 + 0 \cdot 0 \\ =& -0.3. \end{aligned} \tag{5.35}$$

Completing this procedure for all receptive fields, we get that the output matrix N *is given by*

$$N = \begin{bmatrix} -0.8 & 0.9 & 0.0 & -0.4 & -0.3 & -0.3 \\ -0.9 & -0.2 & 0.2 & 1.6 & 1.1 & 1.5 \\ 0.2 & -0.8 & -0.7 & -2.9 & -2.1 & -1.8 \\ 0.2 & -0.3 & -1.2 & -0.3 & -0.7 & -0.6 \end{bmatrix}. \tag{5.36}$$

We proceed by applying pooling to create the pooling map. In this example, let the aggregation function G *be the* max *function and the pooling kernel be of size* 2×2*. This results in a* 3×2 *pooling map* P*, for which ,following Equation 5.32, the entry* $p_{1,4}$ *is computed to be*

$$p_{1,4} = \max\{n_{1,5}, n_{1,6}, n_{2,5}, n_{2,6}\} = 1.5 \tag{5.37}$$

and furthermore, the procedure is illustrated in the right of Figure 5.6.

The complete pooling map P *is then given by*

$$P = \begin{bmatrix} 0.9 & 1.6 & 1.5 \\ 0.2 & -0.3 & -0.6 \end{bmatrix}. \tag{5.38}$$

This matrix is now to be handed over to the fully-connected layers which conclude the CNN and generate the output of the network.

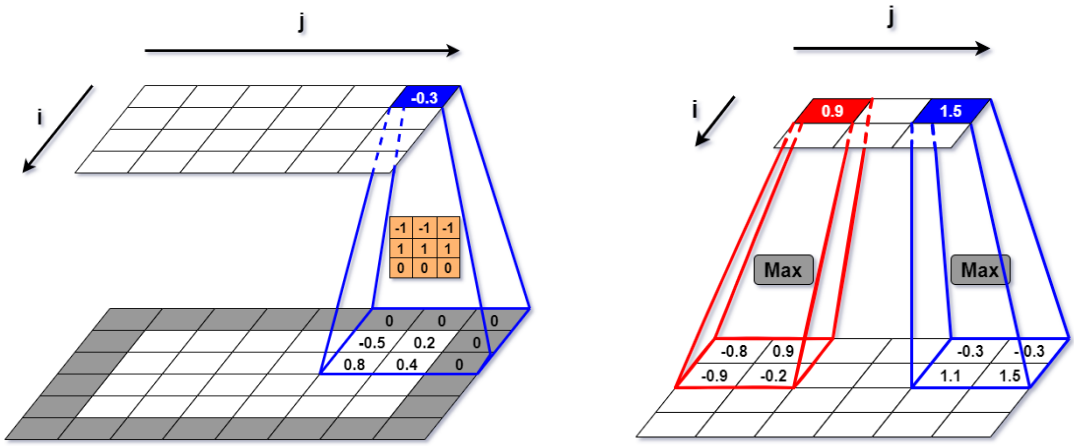


Figure 5.6: An example of (left) a convolutional layer with zero-padding and a kernel detecting top horizontal edges and (right) an example of a max pooling layer with 2×2 pooling kernel, a stride of 2 and no padding. The illustrations are inspired by the figures in Chapter 14 of [60].

It is worth mentioning, that this example is obviously much simplified and most CNN architectures like AlexNet add an activation function, often ReLU, after every convolutional layer and padding to the pooling layers. Moreover, if we would like to take a look at the filtered image, we could scale the matrix entries to $[-1, 1]$ and easily generate a gray-scale image.

5.3.4 Dilated Convolution

At the end of the previous subsection we focused on CNNs in the two dimensional setting. Since this work aims at comparing several types of neural networks for financial time series, let us turn back to the one-dimensional case. Although CNNs were primarily developed for image processing, they can also be applied very successfully to time series. A famous example is **WaveNet** which has been developed for audio processing ([125]).

Most time series such as audio or financial market data show long-term correlations and non-linear dependencies. The architectures of networks handling these two properties build on a non-linear p-autoregression of the form

$$Y_t = \sum_{i=1}^{p} \Phi_i(X_{t-i}) + \epsilon_t. \tag{5.39}$$

Φ_i are coefficient functions for $i = 1, \ldots, p$ which dependent on the data and are optimized through the training process of the network. To allow for a better and faster training of the long-term non-linear dependencies, the usage of **dilated convolution** has

been proposed by Borovykh, Bohte and Osterlee ([23]). In contrast to normal convolution, the dilated variant is only applied to every dth element in the input and thus increases the learning efficiency in particular for connections between data points at far-apart points of time. For a dilated network design with L layers, an exemplary choice for the dilation factor d is

$$d^{(l)} = 2^{l-1}, \quad l = 1, \ldots, L. \tag{5.40}$$

Figure 5.7 shows an exemplary architecture of a dilated CNN with three 1D convolutional layers that have been stacked on top of each other. Before, the receptive field of a convolution neuron was given by the neurons in the previous layer that influence its output. In addition, we define the size of the receptive field r of the whole network as the number of input neurons, e.g. the financial time series, which have an impact on the network's output. Thus, the network in Figure 5.7 has a receptive field size of $r = 8$, i.e. the output is influenced by eight input neurons.

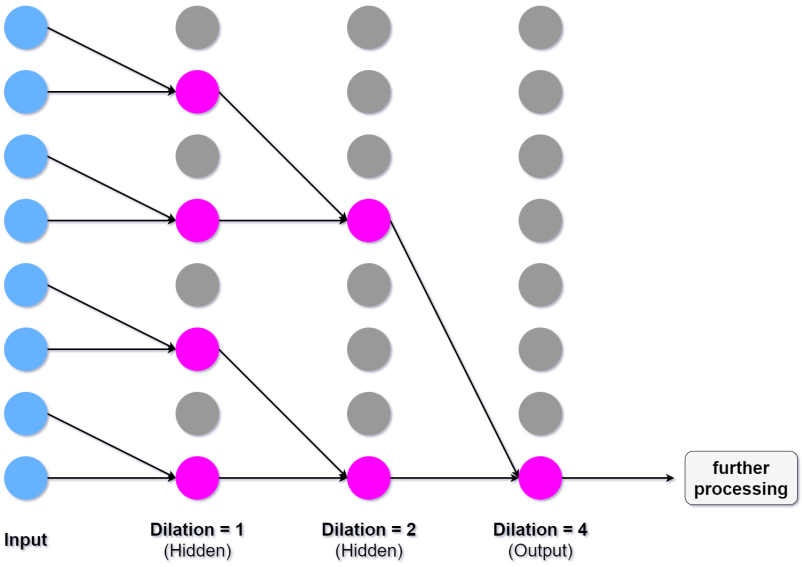


Figure 5.7: An example of a dilated convolutional neural network with three layers and dilation factor $d^{(l)} = 2^{l-1}$ for layers $l = 1, 2, 3$. Gray neurons indicate neurons that are not connected to other layers. The illustration is inspired by Figure 3 in [125].

For now, we will not go into further detail on CNNs and refer the interested reader to Chapter 5 in [31] and Chapter 14 in [60] for a more precise description of CNNs for image recognition. Theory focusing on applications in finance can be found in Section 8.5 in [48] where the authors also present conditions for stationarity and stability of simple CNNs.

5.4 Efficient Data Preparation: Autoencoders (AEs)

In many application tasks, and in particular in finance, we often face high-dimensional data from multivariate time series. The larger the input is, the larger the network and the more data is needed to train the model without overfitting. Convolutional neural networks presented in Section 5.3 offer improvements by reducing the number of parameters in the model while maintaining a high level of expressiveness. In most tasks, labeled data is scarce and it would thus be helpful to compress the information contained in the high-dimensional input data. A heavily used statistical method to achieve this form of dimensionality reduction is called **principal component analysis (PCA)** which aims at finding the dimensions or axes communicating the most relevant information and has been introduced in Section 2.1.5.

In practice, PCA is efficient for linear data relations but it is not capable to handle non-linear or even piecewise linear relations in complex data sets ([182]). There exist variants of PCA which allow for applications in non-linear setting and the rise of deep learning also provided us with a neural network structure called **Autoencoder (AE)** which is able to perform effective dimensionality reduction and shall be the focus of our interest in this section.

5.4.1 Autoencoders (AEs)

Autoencoders are different to most neural networks we have seen so far in a way that they belong to the field of unsupervised learning techniques and their intention is to prepare or simplify data. In contrast to CNNs, which also provide us with a low-dimensional representation but is paying attention to the inputs which are critical to solve a specific task, Autoencoders perform information selection which can be handed over to a wide range of tasks and models. An Autoencoder has the basic architecture presented in Figure 5.8 and consists of two building blocks: an **encoder** and a **decoder**. The encoder takes the input, produces an embedding and compresses it into a vector of lower dimensionality. Subsequently, the decoder inverts the computations from the first part of the network and tries to reconstruct and match the inputs. In other words, the network aims at extrapolating the identity map $F(X) = X$. The hidden layer in the very middle is called the **code layer** and often referred to as the bottleneck where the selection of information is performed. This basic design was introduced by Hinton and Salakhutdinov in 2006, usually has a shape similar to an hourglass and is able to capture structures that PCA would not detect ([78]). The researchers show that Autoencoders are more effective in classifying data such as the MNIST data set of handwritten digits and the implementation of multiple hidden layers helps the network to learn more complex structures (**stacked Autoencoders**).

Moreover, AEs can be regarded as the non-linear and non-parametric analouge to PCA

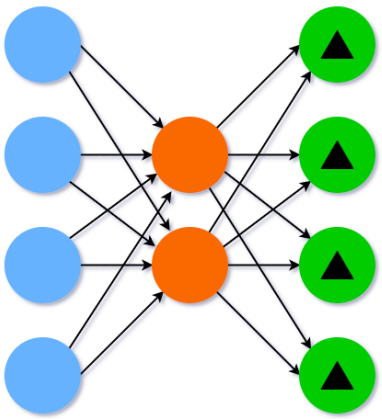


Figure 5.8: Basic architecture of an Autoencoder with match cells (green with triangle). The illustration is inspired by [183].

and therefore deep versions are well suitable for applications in finance ([76]), e.g. when monitoring risk or predicting future prices. A small set of so called **deep factors** can be determined which explain most of the movements and for Autoencoders with ReLU activation, deep factors can be interpreted as compositions of vanilla options.

To allow for a more robust generalization power, **Denoising Autoencoders** have been introduced by Vincent et al. ([185]). The architecture remains the same, despite that noise is added to the inputs to create a corrupted version $C(X)$ which is then handed over to the input layer. In this setting, it is useful to increase the number of neurons in the hidden layers in order to allow the network to handle the more complex task. From a mathematical perspective, the goal of Autoencoders is to find a low-dimensional manifold that unites all data points in a given category best, i.e. with the smallest error ([17]). Denoising AEs learn which information is generalizable and which should be regarded as noise which leads to an approximation of the manifold of interest. As a consequence, this network type is capable of replicating biased inputs similar to the human brain when being confronted with blurred or incomplete images. To further force the Autoencoder to favor sparsity and low-dimensionality, a form of regularization penalty can be added to the loss function ([139], [114]). Another extension is provided by **Variational Autoencoders** which transform the input into parameters describing a probability distribution ([99]). These networks have a similar structure to basic AEs while being closer to the basic Bayesian theory. By taking into account randomness, they provide a better generalization power at the cost of extensive mathematical complexity and increased computational effort.

With this, we conclude our study of Autoencoders and refer to the literature mentioned above, an article by Plaut ([136]) and Chapter 6 in [31] for more details and further discussions on Autoencoders.

5.5 Attention and Transformer Networks

When one of the masterminds of machine learning and artificial intelligence, Alan Turing, thought of a test to verify the intelligence of a machine, he chose a linguistic task in which the machine would have to convince a human interrogator that they is chatting with a human and not a machine ([181]). The test is called the **imitation game** and nowadays researchers and developers spend a lot of time and resources into the developments of chatbots. Until a few years ago, models to process natural language built mostly on RNNs which are able to generate text. A famous example is **Char-RNN**, a neural network able to generate novel texts, e.g. a Shakespeare imitation, after it has been trained with original texts ([96]). Another standard application of language processing is sentiment analysis of text. The 'hello word' example in this field is the classification of movie reviews and many variants of neural networks based on both recurrent and convolutional layers in combination with LSTM and GRU cells exist ([73]).

5.5.1 Attention Mechanisms

In even more advanced tasks, such as automated machine translation, which use an encoder-decoder design as presented for RNNs in Figure 5.2, it is noticeable that the networks sometimes become very complex and that representations are often dragged along unused for many time steps which can be unflattering, taking into account the short memory of RNNs ([174]). A 2014 paper by Bahdanau et al. paved the way for significant improvements in machine translation for long sentences ([14]). The authors propose an encoder-decoder structure including a decoder that is able to focus on the appropriate word at each time step (**attention**). Here, the decoder takes previous inner hidden states, the label from the previous pass and attention weights for the encoder outputs which are generated by a small neural network with a softmax activation function, called **alignment model** or **attention layer**. This procedure was improved and simplified by Luong et al. in 2015 ([113]) by using simple dot products to compute the attention weights. The benefits of attention networks are the reduced number of parameters which reduces the computational effort and the improved explainability of the process ([146]). Recall that one of the initial concerns about neural networks was that too much of the process is hidden in a black box. Taking this into account, the fact that attention models are easier to understand and to track cannot be overstated and thus increases the legitimacy of this variant of machine learning. Moreover, adptions of attention mechanisms have proven to be successful in tasks other than language processing, e.g. visual attention ([190]).

An example of a basic architecture of an attention network is illustrated in Figure 5.9. Using the softmax activation function and combining the encoder output with the previous hidden decoder states, as proposed by Bahdanau et al. and shown in the figure, is known as **concatenative attention**.

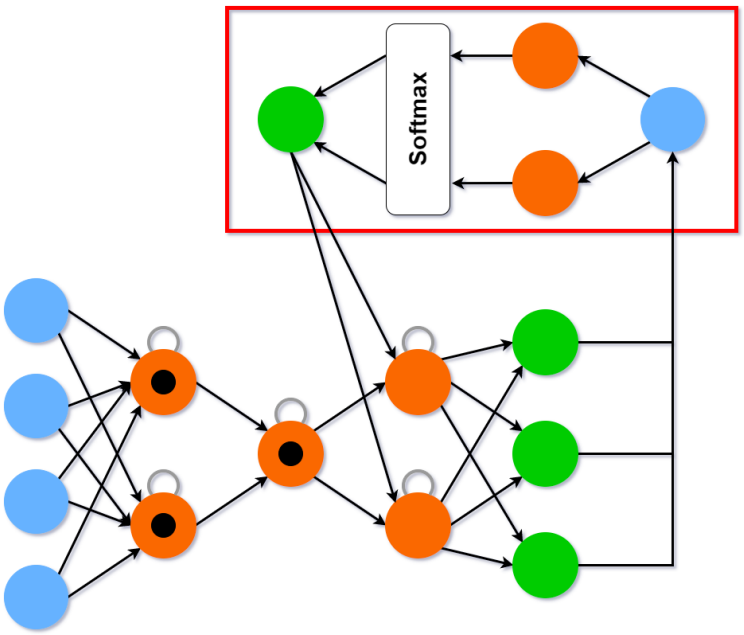


Figure 5.9: Basic architecture of an attention network of the concatenative type with recurrent cells with memory (orange with black circle) and an attention layer (red box). The illustration is inspired by [183].

5.5.2 Basics on Transformer Networks

The attention mechanisms presented in the previous subsection have been designed in conjunction with a recurrent network, sometimes amended with convolutional layers. In 2017, a paper from Google researchers, which is now known as one of the most influential in state-of-the-art machine learning, proposed a structure entirely relying on attention mechanisms while sticking with the approved basic encoder-decoder architecture ([184]). The researchers' intention was to boost the generalization performance of deep learning tasks like machine translation by replacing the recurrent layers to allow for a better parallelization and shorter paths between input and output positions which the signals have to travel when traversing through the network. Exclusive usage of attention mechanisms and in particular a method called **multi-head self-attention** which relates the different postions of the sequence led to a performance that was unrivalled at the time and at the same time meant less computational effort. Nevertheless, some other layers and pieces are also required to complete building the network which is called the **Transformer**.

Figure 5.10 shows the architecture of the Transformer, which will be explained in more detail below, just like the newly introduced components.

Like most high-performing neural networks designed for sequence processing, the Transformer network builds on an encoder-decoder structure. The encoder, which consists

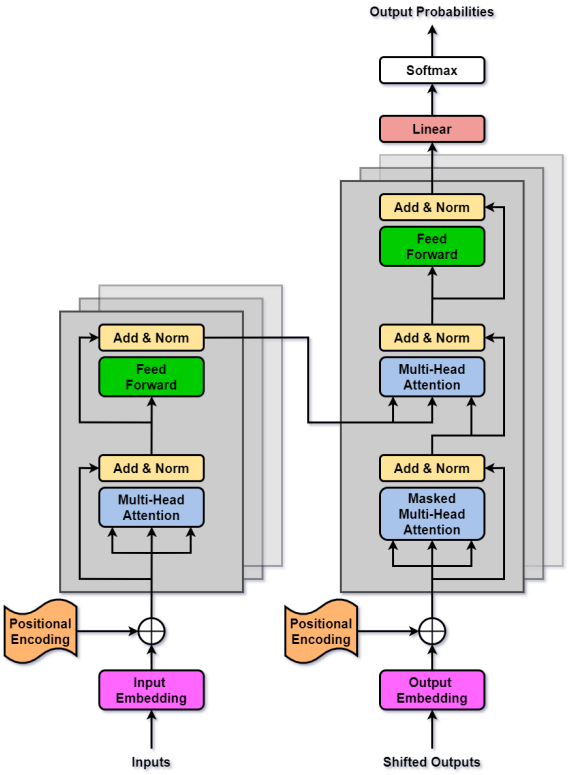


Figure 5.10: The Transformer architecture where the left part is the encoder and the right part the decoder. The illustration is a reproduction of Figure 1 in [184].

of a stack of N identical layers and is indicated by the left gray boxes in Figure 5.10, takes an input sequence $X = (X_1, \ldots, X_n)$ and transforms it into a coded representation sequence $C = (C_1, \ldots, C_m)$. Each of the encoder's layers has a **multi-head attention mechanism** followed by a fully connected FFNN. In addition, both sub-layers employ a residual connection ([79]), are normalized before the signals are passed on and produce an output of the same dimension which also is the same as the one of the output of the embedding layers. In more mathematical terms this can be written as

$$\text{LayerOutput} = \text{Normalization}\left(\text{LayerInput} + \text{LayerOperation}(\text{LayerInput})\right), \qquad (5.41)$$

where LayerOperation is the function implemented by the sub-layer and its coefficients. Layer normalization is an advancement of batch normalization, makes use of the empirical mean and variance to normalize the outputs and reduces the training time significantly ([13]).

The decoder which is also consisting of N identical layers, takes the sequence C and the previous outputs shifted to the right by one, i.e. $\tilde{Y} = (\cdot, Y_1, \ldots, Y_i)$, to determine the next element Y_{i+1} of the sequence $Y = (Y_1, \ldots, Y_m)$ in every step and is indicated by the gray

boxes in the right part of the illustration. In contrast to the encoder, the decoder contains an additional third sub-layer performing a masked multi-head layer attention mechanism on the previously generated outputs. **Masking** prevents the decoder from cheating, i.e. peeking at future positions when training, by setting them to $-\infty$. Moreover, C serves as the input for the standard multi-head attention mechanisms and once again residual connections and layer normalization are applied.

The input that is processed to both the encoder and the decoder is the result of an embedding, which is similar to convolution, and **positional encoding** of the original signals. Positional encoding is crucial for time series processing since the position of the feature within the series is very important and allows to capture the dependencies over time.

Several of the components of this architecture are well-known to us, but we have not studied positional encoding or multi-head attention yet. Before we turn to them in detail, let us deal with the mathematics of the fully-connected feedforward networks each composed of two dense layers. In the paper by Vaswani et al., they consist of two linear transformations with a ReLU and an identity activation, resulting in

$$y_{\text{FFNN}}^{(l)} = F_{\text{FFNN}}^{(l)}\left(x_{\text{FFNN}}^{(l)}\right) = W_2^{(l)} \max\left(0, W_1^{(l)} x + b_1^{(l)}\right) + b_2^{(l)}, \tag{5.42}$$

where $W_1^{(l)}, W_2^{(l)}$ are the weight matrices and $b_1^{(l)}, b_2^{(l)}$ the biases of layer $l = 1, \ldots, N$.

Finally, the linear transformation, which is usually determined by previously learned parameters, and a dense layer with softmax activation transform the decoder's output into predictions of the probabilities of the next element of the sequence Y, e.g. the next word in a sentence.

5.5.3 Time and Positional Encoding

In contrast to the networks presented in Section 5.5.1, the Transformer has no recurrence or convolutional layers which tell the model about the possibly highly relevant order of the sequence handed to the model. It is thus necessary to add information about the position of the word in the sentence or the time at which the prices of a stock were observed. This is called **positional** or **time encoding** and they aim at encoding the relative or absolute position or time by adding a positional component $P_{p,i}$ to the features of the embedded inputs. These position could be learned by the model in the course of the training, but often fixed encodings are used.

Example 5.7 (Fixed Positional Encoding)**.** *Let $X = (X_1, \ldots, X_n)$ be a sequence with features $X_i \in \mathbb{R}^d$. Then the simplest of all positional encodings is given by the map*

$$P : \mathbb{R}^{d \times n} \to \mathbb{R}^{(d+1) \times n} \quad , \quad P(X) := ((X_1, 1), \ldots, (X_n, n)) \,, \tag{5.43}$$

where we simply add the counted position starting from the first feature of the sequence.

In the Transformer paper, the authors favor a hand-crafted sine/cosine map, given by the positional encoding matrix P with elements

$$P_{p,i} := \begin{cases} \sin\left(\delta/10000^{\frac{i}{\delta}}\right), & i \text{ even}, \\ \cos\left(\delta/10000^{\frac{i-1}{\delta}}\right), & i \text{ odd}, \end{cases} \tag{5.44}$$

with i indicating the component of the encoding, p the position of the feature in the sequence and δ the dimensionality of the input and output of each layer in the model.

The first example of positional encoding simply adds the position of a word in a sentence to the vector representing the word while the second one focuses on relative positions and can be extended to arbitrarily long sentences. However, if other time series, such as financial time series, are fed to the Transformer, it is desirable to have notions of time instead of position, in particular if the sequence is asynchronous or not iid. These can be generated by applying the *Time2Vec* approach, which can be thought of as a vector representation like an embedding layer which ultimately even accelerates the training of neural networks significantly ([97]). In contrast to hand-crafted time features, Time2Vec is a general-purpose model-agnostic time representation suitable for many architectures by learning a vector embedding of time. Moreover, Time2Vec is designed to capture periodic (seasonalities, higher amount of sales on the weekend) and non-periodic events (increased risk of illnesses at a high age), be invariant to time scaling (hours, days, weeks) and be simple enough to be applied to a wide range of models.

Definition 5.8 (Time2Vec)**.** *Let τ be a scalar notion of time, $k \in \mathbb{N}$, $\omega = (\omega_i)_{i=0,\dots,k}$ and $\beta = (\beta_i)_{i=0,\dots,k}$ two sets of learnable parameters and H a periodic activation function. Then the $k+1$-dimensional vector* **Time2Vec** *of τ, denoted by $t2v(\tau)$, is given by*

$$(t2v(\tau))_i := \begin{cases} \omega_i\tau + \beta_i, & i = 0, \\ H(\omega_i\tau + \beta_i), & i \in \{1, \dots, k\}. \end{cases} \tag{5.45}$$

The vector $t2v$ is then concatenated to the input. Kazemi et al. ([97]) found that $H = \sin$ is a promissing choice for the activation function. For the sine, the ω_i and the β_i can be interpreted as the frequency and the phase-shift respectively. For example, a weekly periodicity can be achieved by setting $\omega_i = \frac{2\pi}{7}$ and the sine will peak at every 7th time step. Interestingly, the ReLU function performs worst when chosen as the activation function for the Time2Vec representation. The researcher demonstrate that at least for a variety of applications, learning the frequencies and phase shifts improves the model's performance compared to fixed encodings. Moreover, capturing both periodic and non-periodic patterns allows for an improved performance compared to encodings that only focus on periodic patterns.

Even though positional and time encoding are essential for some time series, there exists no implementation of an encoding layer in Keras, which is why this type of layer has to be implemented from scratch.

5.5.4 Self-Attention and Multi-Head Attention

We now turn to the second novelty: **multi-head attention**. First, we recall that attention mechanisms are able to determine the relation between two elements of a sequence. Thus, position or time encoding is absolutely necessary if the order within the sequence is of importance. There exist several attention mechanism that differ in how they compute their output. Formally, an attention mechanism is a mapping of vectors of **queries** $q_i \in \mathbb{R}^{d_k}$, **keys** $k_i \in \mathbb{R}^{d_k}$ and **values** $v_i \in \mathbb{R}^{d_v}$ that returns a weighted sum of the values determined by linear transformation of the embedded input x as follows ([151]):

$$q_i := W_q^{(i)} x + b_q^{(i)}, \quad i \in \{1, \dots, d_k\}, \tag{5.46}$$

$$k_i := W_k^{(i)} x + b_k^{(i)}, \quad i \in \{1, \dots, d_k\}, \tag{5.47}$$

$$v_i := W_v^{(i)} x + b_v^{(i)}, \quad i \in \{1, \dots, d_v\}, . \tag{5.48}$$

Here, $W_q^{(i)}, W_k^{(i)} \in \mathbb{R}^{d_k \times p}, W_v^{(i)} \in \mathbb{R}^{d_v \times p}$ and $b_q^{(i)}, b_k^{(i)} \in \mathbb{R}^{d_k}, b_v^{(i)} \in \mathbb{R}^{d_v}$ denote the weight matrices and biases respectively which can be learned by the model. These column vectors are each collected in matrices Q, K and V respectively and the names have their origin in linguistic machine learning tasks. The option that has been selected by the authors of the Transformer paper to compute the attention signals is called **scaled dot product attention** and is based on the unscaled version proposed by Luong et al. ([113]).

For the original Transformer, a sequence of length s and an embedded input vector $x \in \mathbb{R}^p$, the signal is determined by computing the dot products of the query and keys, scaling it by $\frac{1}{\sqrt{d_k}}$ to avoid saturating the activation function and thus vanishing gradients before applying the softmax function column by column and multiplying this with the values V. In mathematical terms, we compute the attention mechanism's output by the mapping

$$A : \mathbb{R}^{d_k \times s} \times \mathbb{R}^{d_k \times s} \times \mathbb{R}^{d_v \times s} \to \mathbb{R}^{d_k \times s}, \quad A(Q, K, V) := \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V. \tag{5.49}$$

Since the attention mechanism figures out the relations between the vectors within a sequence, this particular version of attention is called **self-attention**. It is useful to have multiple perspectives on the problem, i.e. applying h different learned linear transformations of the queries, keys and values in separate, parallel attention layers. This approach is called **multi-head attention**, is illustrated in Figure 5.11 and allows for parallelization yielding h different outputs which are then concatenated and linearly projected. Recall that masking an attention mechanism means preventing the model from peeking in the future while learning the parameters. Furthermore, applying dropout when training the model is recommended to achieve a higher level of robustness.

In mathematical terms, this can be represented as the multi-head attention function M with h different projections $A_1, \dots, A_h$ given by

$$M(Q, K, V) := \text{Concatenate}\left(A_1, \dots, A_h\right), \tag{5.50}$$

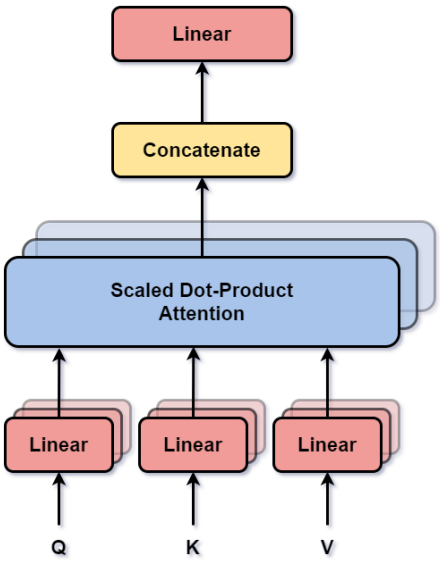


Figure 5.11: The architecture of a multi-head attention layer with $h = 3$ different parallel attention layers. The illustration is a reproduction of Figure 2 in [184].

where A_i for $i = 1, \ldots, h$ is the output from a single attention mechanism fed with linearly parametrized inputs $L_Q(Q), L_K(K), L_V(V)$.

By now, all pieces of a typical Transformer network have been explained and we are ready for the implementation in Chapter 6 using pre-implemented attention layers in Keras. There exist many variants of Transformers and the adaption of the model for specific tasks is an active subject of research. Before we conclude this chapter, we refer to further theory on Transformers and the attention mechanisms included in the model which can be found in [151], Chapter 4 in [184] and in the Bachelor's thesis of a fellow student ([62]). Moreover, an exemplary application of a Transformer type model for univariate time series is presented in [109] and in [191] the authors suggest a new design which only consists of the left decoder part of Figure 5.10 followed by a simple linear projection without softmax activation. The latter model is suitable for a broader range of tasks than the original Transformer architecture while offering a better performance for classification and regression. Pre-training and an adapted loss function have to be applied for this purpose.

Another currently very popular example of the use of transformers is a language representation model called **Bidirectional Encoder Representations from Transformers (BERT)** which combines pre-training with deep bidirectional representations to achieve state-of-the-art performance in language processing tasks ([43]).

5.6 Summary

At the beginning of this chapter about advanced neural networks, we introduced recurrent neural networks (RNNs). These networks can be regarded as an unfolding over time and have inner hidden states which provide the model with a memory. There exists different designs which take either a vector or a sequence as the input and output a vector or a sequence. By unrolling RNNs, the model can be trained with a variant of backpropagation called backpropagation through time (BPTT). For RNNs, as well as for the other advanced networks presented here, preparations methods such as parameter initialization, gradient clipping, dropout and batch or layer normalization are recommended to improve the training performance and avoid unstable gradients. A version of RNNs which allows for heteroscedasticity is called generalized RNN (GRNN) and generalizes the model similar to how the GARCH model generalizes the ARIMA model. Furthermore, α- and α_t-RNNs are the equivalent to standard and dynamic exponential smoothing respectively. Both, α- and α_t-RNNs, provide the model with a longer memory than the plain RNN and the latter networks enable us to fit a model to non-stationary data.

To tackle the issue of the limited memory of RNNs, long short-term memory (LSTM) cells have both a short-term and a long-term hidden state. Gates within the cell select which information is erased and added to the memory of the cell. Introducing this innovation into a RNN has proven to be very successful in diverse application tasks due to faster convergence and improved detection of long-term patterns. There exist several variants of LSTM cells, of which gated recurrent units (GRUs) are arguably the most popular one. GRU cells have only one hidden state and even though they are a simplified version of LSTM cells, they offer a similar performance. Combining LSTM and GRU cells with dilation to further compress the data yields an improved performance of the model.

Convolutional neural networks (CNNs), which have originally been developed for image processing, exploit spatial structures in the data and layers of convolutional neurons reduce the complexity of the data at hand. Each convolutional neuron is connected only to a part of the input, called receptive field, and applies filters to create a feature map which highlight the areas that activated the filter the most. Filters are able to detect certain patterns in the data, e.g. horizontal lines in an image. To reduce high-dimensional data after several convolutions, pooling layers are applied to shrink the complexity and sharpen the features. Mathematically, convolution is based on weighted moving average smoothers and simple CNNs can even be interpreted as an AR(p) model. To extend the horizon of the CNN's memory, dilated convolution has been introduced.

Before actually training a neural network, it can be helpful to reduce the dimensionality of the data. The standard statistical approach for this reduction is principal component analysis which finds the dimensions communicating the most relevant information and sorts out less important axes. The equivalent of PCA in deep learning are Autoencoders (AEs) which are based on a classical encoder-decoder architecture.

Finally, we studied attention networks. These network types are able to determine which features of the input are the most important, e.g. which word in a sentence or which part of an image. Stacking several attention mechanism and combining them with feedforward layers and positional or time encoding leads to the Transformer architecture. Transformer networks are currently state-of-the-art in many tasks connected to time series processing mostly because of the opportunity to parallelize the processes and thus reduce the computational time significantly. In contrast to RNNs and CNNs, Transformers are able to handle the whole input sequence at once and do not have to process every feature one after the other.

Table 5.2 compares the computational effort of an attention, a recurrent and a convolutional layer. Obviously, for long sequences, which are common in advanced deep learning tasks such as financial time series forecasting, the Transformer architecture building on mostly attention mechanisms is advantageous.

Table 5.2: A comparison of path lengths, complexity and number of sequential operations for recurrent, convolutional and attention layers. k is the kernel size of the convolution, s the length of the sequence and d the dimension of the layer output. The table is a reproduction of Table 1 in [184].

Layer Type	**Complexity per Layer**	**Sequential Operations**	**Maximum Path Length**
Recurrent	$\mathcal{O}(sd^2)$	$\mathcal{O}(s)$	$\mathcal{O}(s)$
Convolutional	$\mathcal{O}(ksd^2)$	$\mathcal{O}(1)$	$\mathcal{O}(\log_k(s))$
Self-Attention	$\mathcal{O}(s^2d)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

After having introduced the advanced neural networks of interest in this chapter, we will now turn to their implementation and numerical comparison.

6 Numerical Comparison of Different Machine Learning Methods for Stock Price Prediction

In the previous chapters, we introduced the main concepts of machine learning and focused on neural networks in particular. In this chapter, we aim at comparing the different methods introduced so far with regard to the prediction of financial market prices. First, the various tuning methods for improving the generalization power of the networks will be evaluated on a forecasting task for Bitcoin returns, which are more challenging to predict since the prices show patterns of extremely high volatility. Indeed, the modelling of Bitcoin is often used in the literature to verify the performance of a machine learning method (e.g. in [48]). The focus will be on standard feedforward neural networks (FFNNs) and the implementation of dropout, different activation functions, scaling, learning rate scheduling and many more to verify their effect on the prediction power for a rather simple one-step-ahead prediction task. Afterwards, the different advanced neural networks presented in Chapter 5 will be evaluated for a multivariate prediciton task of minutely returns for stocks and indices. We conclude by an interpretation of the results and mention aspects of the modeling that could be improved.

The implementation is done using iPython notebooks and build to run on most common computers. The author used Jupyter Notebook, Version 6.4.5, which builds on the Python programming language, Version 3.9. The most important library for neural networks is `TensorFlow`, Version 2.7.0, which contains the `Keras` library and will be used exensively in what follows. Not all of the results and plots will be presented in this work but the codes can all be found in the files `Comp_FFNN.ipynb` and `Comp_ANN_Final.ipynb` in the attachment. Care was taken to use as few black boxes as possible and to design the code independently and as comprehensibly as possible. The methods and implementation presented here are far from perfect or complete and the topic is still a matter of intense research. Moreover, the design of neural networks is depending on subjective believes and there exist no hard rules which one could follow.

6.1 Tuning Methods

This section deals with the methods than can be used to tune the performance of most neural networks. All methods will be evaluated and compared for a classical feedforward neural networks (FFNNs). The methods have been introduced in previous chapters in some detail and references can be found there. The goal is to present the implementation of these methods and verify whether they are useful for the particular example data used here. The data of interest are the daily and minutely log returns of Bitcoin over a given period. By performing the fitting of a FFNN for both daily and minutely data, we will be able to verify if some of the approaches work particulary well for these two time horizons. This might be of interest since we have seen that the reference period of autocorrelation can be long and most network types are unable to memorize many time steps. Of course, the algorithm is not interested in the actual time difference between the time steps at hand but for some prediction tasks it might require a subjective guess which time steps should be choosen for a good guess of future prices.

Since the fitting algortithms are of stochastic type, the code fits every model architecture ten times and determines the performance by taking the average of all ten prediction errors. Note that the results vary every time the code is ran because of stochasticity, different CPU or GPU capacities and since the data downloaded is the most recent one and not from a fixed time period. The notebook containing all the implementations is called `Comp_FFNN.ipynb` and can be found in the attachment.

The first training method is scaling of the data. This means that we normalize or standardize the data at hand before handing it to the network. This method has not been introduced explicitly in this work but is well-known in Data Science and commonly used over many mathematical disciplines. As a reference to stress the practicability and motivation of scaling, we refer to [6]. All of the other tuning methods that will be presented here are listed below alongside with a reference to the section where they have been introduced in this work:

- Batch normalization (Section 4.2.2),
- Dropout (Section 4.2.1),
- Initialization (Section 4.2.1),
- Regularization (Section 3.4.1),
- Different activation functions (Sections 4.1 and 4.6),
- Different optimizers (Sections 3.4.2 and 4.2.3),
- Different FFNN designs, i.e. number of layers and neurons,
- Variation of batch sizes, and
- Different learning rate schedules (Section 4.2.3).

In addition to these tuning methods, we will also present implementations of other methods which can help to simplify the task, speed up the training and improve the evaluation of the fit. These methods are

- Resampling,
- Cross-validation (Section 3.4.2), and
- Autoencoder for data preparation (Section 5.4).

The reader should always keep in mind that most neural network architectures and all of the tuning methods mentioned in this chapter have been developed for tasks completely different to (financial) time series forecasting. Thus, it is possible, if not likely, that some of the techniques harm the predicting power of the models. However, in this section we will only investigate the usefulness of the tuning methods for two time series which is why the results are not representative for other data.

The data of interest is downloaded once again using Yahoo!Finance and should thus be available for everyone who runs the code. Two data sets are downloaded, a smaller one ($n = 1518$) with daily close prices of Bitcoin in USD and a larger one ($n = 34608$) with minutely prices over the last 28 days. This data is given in a `Pandas` data frame and we proceed by computing the log returns for the period of interest before splitting the data into a training and a test set. For all methods, the generalization power will be measured by the MSE.

6.1.1 Normalization and Standardization

In addition to the standard log return data, we create a $(0, 1)$-normalized and a normally standardized data set which, as well as the original return series, are split into a training and a test data sets. Next, we create a basic FFNN model and fit it for every data set. Finally, the prediction power is verified by computing the mean square error (MSE) of the predictions generated by the fitted models with the actual observations. The results, as well as the time the fitting took, are summarized in Table 6.1. The values are average values of ten models with identical structure and properties trained and evaluated with the same data. In this table, as in all following tables, improvements are colored in blue-green, while deteriorations compared to the reference model - a FFNN with layers with 30, 15 and one neuron and without any tuning method and trained with unscaled data - are illustrated by a red coloring.

We find that standardization provides us with a better performance when training the model. On the other hand, the generalization power is significantly worse than for a model trained with the original data. This might indicate overfitting. However, we are not surprised by this result since standardization is recommended for data that follows a Gaussian distribution and this is not the case here. Normalizing the data seems to improve the prediction power of the model for the minutely data, even though the effect

Table 6.1: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin using unscaled and scaled data sets.

Time Horizon	Data Type	Time [s]	MSE Training	MSE Test
Daily	Unscaled	6.55	$1.593 \cdot 10^{-3}$	$1.678 \cdot 10^{-3}$
	Normalized	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	Standardized	7.63	$0.587 \cdot 10^{-3}$	$2.338 \cdot 10^{-3}$
Minutely	Unscaled	19.29	$7.791 \cdot 10^{-7}$	$3.428 \cdot 10^{-7}$
	Normalized	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	Standardized	49.69	$6.555 \cdot 10^{-7}$	$3.780 \cdot 10^{-7}$

is small but the difference between the MSE error for the training and the test set is not very pronounced indicating that the normalized data is useful at the cost of increased computational effort. Moreover, the negative effect of normalization on the daily data set is not very pronounced and might be effected by the rather small size of the data set. Recall that from previous theory and literature we know that normalization is useful and should be applied. This convinces us to use normalization techniques for the rest of this chapter and, in particular, we will use the model trained for normalized data as the new reference model for all upcoming modifications.

One possible cause for the small extend of the effect could be that log returns are already small-scale data and thus the scaling is not as effective as it is for large-scale data, e.g. if we would have analyzed the closing prices instead of their returns. On the other hand, having data with small values as we do, the model might be more prone to run into vanishing gradients which is why normalization is adequate for this setting.

Another point that deserves an investigation is the fact that for the minutely data set, the test error is lower than the training error which is rather unusual and counterintuitive. A likely reason is that the test data is 'simpler' than the training data, i.e. it has fewer extreme events since the Bitcoin price is highly volatile and heteroscedastic. Furthermore, the use of dropout in the course of training the model increases the error significantly in the training process while making the network more robust.

In a next step, we will apply normalization throughout the layers of the network.

6.1.2 Batch Normalization

Applying normalization to every single inner hidden layer is called batch normalization and this concept is well known to us. Even though Ioffe and Szegedy who introduced this concept in their work [90] suggest placing the normalization layer immediately after the inner layer and thus in advance of the activation, some researchers recommend placing it behind the activation (cf. this discussion). Here, we will perform normalization after

the activation took place in each layer.

The results when applying batch normalization can be found in Table 6.2 which summarizes the results for both daily and minutely data.

We find that batch normalization does not improve the generalization power of the model for the task at hand. In fact, the results are less accurate and the required computation time is higher. Our findings coincide with the results presented in [176], where the authors conclude that the use of batch normalization in deep networks of the feedforward type to generate forecasts of financial time series is not as efficient as the use in other contexts such as image processing. According to the authors, the application of batch normalization layers leads to large fluctuations and auto-correlations different from the ones in the original time series unable to capture the conditional variances.

Based on this observations, we will not use batch normalization in the following. However, before turning to the next tuning method (dropout), we refer to the work of Passalis et al., who propose a **Deep Adaptive Input Normalization Layer** which tackles the shortcomings of batch normalization and outperforms fixed normalization schemes by actively learning how to efficiently perform normalization ([131]).

6.1.3 Dropout

Recall that Dropout has been introduced to avoid overfitting and the basic idea is to keep a neuron active with a probability p and regard the neuron as inactive otherwise. In other words, some of the information is not processed through the network for some batches and thus forces the network to be accurate with incomplete information. Here, we choose a dropout rate of 20%, i.e. every fifth neuron in the inner hidden layers will be inactive. Fitting the model with normalized input data and use it to generate forecasts, we obtain the results summarized in Table 6.2.

The results show that dropout slightly improves the results for this time series and both daily and minutely data. Moreover, the training time is reduced which seems to be resonable since the training algorithm has to deal with fewer gradients for every batch since some of the neurons are inactive. Our finding coincide with the ones by Tang et al. ([178]) and from this paper we draw confidence that dropout will be very useful for LSTM and GRU network architectures that will be of our interest in the second part of this chapter.

It should be noted that the inclusion of dropout layers would not have been necessary if batch normalization had improved performance. The reason for this are the findings by Li et al. who present an explanation why combining both batch normalization and dropout in a neural network lead to a worse performance ([110]). Furthermore, the authors conclude that dropout can be dismissed if batch normalization is applied.

After focusing on the two techniques which are often regarded as two of the most powerful

ones, batch normalization and dropout, we now examine the effect of initialization.

6.1.4 Initialization

The problem of exploding and disappearing gradients is omnipresent in most machine learning tasks. In this context, it is substantial that the inital weights of the untrained network are different from each other. In particular, initializing all weights to the same value would force whole layers to act like a single neuron. To cure this, several forms of weight initialization have been introduced, among others Glorot, He and LeCun initialization. While the second one is recommended for ReLU and similar activation functions, we will focus on Glorot normal initialization here since the default activation function in our code is tanh. Any of these methods initializes the weights and biases before the network is trained.

The results, which are displayed in Table 6.2, indicate that initialization provides us with a similar performance while requiring comparable computational time and effort. The default initializer for Dense layers in Keras is the uniform variant of Glorot initialization, which in turn explains the almost non-existent differences in performance. We will regard initialization methods as useful for financial time series and this result coincides with the findings in the original paper proposing normalized initialization for tasks such as the recognition of handwritten digits ([64]).

Table 6.2: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin applying batch normalization after every inner hidden layer, a 20% dropout for every inner hidden layer or Glorot initialization for every inner hidden layer's weights and biases.

Time Horizon	Method	Time [s]	MSE Training	MSE Test
Daily	Basic	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	Batch Normalization	9.43	$3.229 \cdot 10^{-3}$	$4.813 \cdot 10^{-3}$
	Dropout (0.2)	7.87	$1.651 \cdot 10^{-3}$	$1.688 \cdot 10^{-3}$
	Glorot Initialization	7.92	$1.772 \cdot 10^{-3}$	$1.805 \cdot 10^{-3}$
Minutely	Basic	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	Batch Normalization	59.94	$1.071 \cdot 10^{-6}$	$6.446 \cdot 10^{-7}$
	Dropout (0.2)	42.61	$7.496 \cdot 10^{-7}$	$3.140 \cdot 10^{-7}$
	Glorot Initialization	47.04	$7.587 \cdot 10^{-7}$	$3.229 \cdot 10^{-7}$

6.1.5 Regularization

Regularization penalizies extensive use of parameters and thus helps to reduce the noise in the training process. There exist several variantes of regularization such as Tikhonov or

elastic net regularization but we shall focus on Lasso or ℓ_1-regularization here. Note that by default Keras layers do not include regularization. According to Kuhn et al., reasonable values for the regularization parameter range between 0 and 0.1 ([103], p.144) and we thus evaluate the predictive power of the model trained with regularization parameters $\lambda \in \{0.1, 0.01, 0.001\}$. Obviously, λ is a hyperparameter that could be trained for every task but this would be out of the scope of this work. The results are displayed in Table 6.3.

Table 6.3: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin applying ℓ_1-regularization for every inner hidden layer's weights.

Time Horizon	Regularization	Time [s]	MSE Training	MSE Test
Daily	No	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	$\lambda = 0.1$	4.68	$1.663 \cdot 10^{-3}$	$1.654 \cdot 10^{-3}$
	$\lambda = 0.01$	5.39	$1.657 \cdot 10^{-3}$	$1.646 \cdot 10^{-3}$
	$\lambda = 0.001$	7.69	$1.678 \cdot 10^{-3}$	$1.675 \cdot 10^{-3}$
Minutely	No	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	$\lambda = 0.1$	33.67	$7.522 \cdot 10^{-7}$	$3.155 \cdot 10^{-7}$
	$\lambda = 0.01$	22.24	$7.517 \cdot 10^{-7}$	$3.151 \cdot 10^{-7}$
	$\lambda = 0.001$	21.76	$7.515 \cdot 10^{-7}$	$3.149 \cdot 10^{-7}$

It seems like parameter regularization can help to improve the performance of the model at hand. All choices, $\lambda = 0.1$, $\lambda = 0.01$ and $\lambda = 0.001$, of the regularization hyperparameter are proper ones for both daily and minutely data, which is why regularization should be regarded as apowerful tool to improve the training. In addition, regularization reduced the training time in this setting. The interested reader is referred to [104] for a detailed overview over works on regularization techniques for deep learning.

6.1.6 Activation Functions

Another screw of the network we can easily adjust is the activation function applied on the signal after the information has been processed through a layer. Above, we introduced several activation functions which are used in practice. Here, we decided to compare the performance of five different functions: Sigmoid, tanh, ReLU, SELU and ELU. tanh has been used before as we fitted the standard models. Note that the famous softmax function is excluded since it is designed for classification tasks rather than for generative models like the one we are dealing with. The results can be found in Table 6.4.

We find that, as desired, all activation functions different to tanh offer shorter runtimes, in particular ReLU which proves to be fastest at quite high accuracy costs - at least when limiting the training through a maximum number of epochs and not in absolute time. This

Table 6.4: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin using different activation functions for every inner hidden layer.

Time Horizon	Activation	Time [s]	MSE Training	MSE Test
Daily	tanh	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	Sigmoid	2.15	$1.663 \cdot 10^{-3}$	$1.653 \cdot 10^{-3}$
	ReLU	5.54	$4.502 \cdot 10^{-2}$	$4.483 \cdot 10^{-2}$
	ELU	5.54	$1.793 \cdot 10^{-3}$	$1.832 \cdot 10^{-3}$
	SELU	5.85	$1.922 \cdot 10^{-3}$	$2.006 \cdot 10^{-3}$
Minutely	tanh	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	Sigmoid	24.32	$7.503 \cdot 10^{-7}$	$3.147 \cdot 10^{-7}$
	ReLU	41.71	$4.748 \cdot 10^{-5}$	$4.704 \cdot 10^{-5}$
	ELU	42.56	$7.984 \cdot 10^{-7}$	$3.624 \cdot 10^{-7}$
	SELU	36.00	$8.107 \cdot 10^{-7}$	$3.716 \cdot 10^{-7}$

coincides with the argumentation in Géron (p.338, [60]). However, the performance of the highly recommended SELU function is not better than the other functions. Sigmoid seems to be the best option in our setting, even though we would expect SELU and ELU to allow for a performance boost when being used in advanced network architectures such as LSTM or RNN.

6.1.7 Optimizers

The API Keras that we use for this work also allows us to easily change the optimizer used for the descent. Above we introduced the Adam algorithm as the most important optimization method which can be regarded as an improved descent method taking some momentum into account. In this subsection, Adam will be compared to some other famous algorithms that are available for Keras network training: Stochastic Gradient Descent (SGD), RMSprop, AdaGrad, Adamax and Nadam. While the primer three are predecessors of Adam, the latter two have been introduced to cure the shortcomings of Adam which has been used up to this point. The results are displayed in Table 6.5.

All optimizers require less time to complete the fitting. However, the performance of SGD, AdaGrad and Adamax is not convincing. On the other hand, RMSProp performs better than Adam on the smaller daily data set. However, as expected Nadam outperforms Adam, but the latter one also seems to be a descent choice for models and we will stick to it since we understand it better than the others except SGD and analyzed it in detail.

Table 6.5: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin using different optimization algorithms.

Time Horizon	Optimizer	Time [s]	MSE Training	MSE Test
Daily	Adam	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	SGD	5.15	$1.881 \cdot 10^{-3}$	$1.835 \cdot 10^{-3}$
	RMSProp	5.90	$1.651 \cdot 10^{-3}$	$1.632 \cdot 10^{-3}$
	AdaGrad	7.75	$2.152 \cdot 10^{-3}$	$2.103 \cdot 10^{-3}$
	Adamax	7.62	$1.738 \cdot 10^{-3}$	$1.749 \cdot 10^{-3}$
	Nadam	5.67	$1.630 \cdot 10^{-3}$	$1.649 \cdot 10^{-3}$
Minutely	Adam	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	SGD	37.16	$9.023 \cdot 10^{-7}$	$3.738 \cdot 10^{-7}$
	RMSProp	31.72	$8.205 \cdot 10^{-7}$	$3.833 \cdot 10^{-7}$
	AdaGrad	46.65	$1.270 \cdot 10^{-6}$	$5.301 \cdot 10^{-7}$
	Adamax	29.27	$7.591 \cdot 10^{-7}$	$3.240 \cdot 10^{-7}$
	Nadam	36.04	$7.506 \cdot 10^{-7}$	$3.159 \cdot 10^{-7}$

6.1.8 Different FFNN Designs

So far, we trained a model which contained of a first hidden layer with 30 neurons, a second one with 15 and a final output layer with a single neuron. We are now interested in the performance of feedforward network designs which are different to this standard approach. In the following, we investigate networks in which every inner layer has the same amount of neurons, networks with triangle or rhombus structures and deeper networks. We expect deeper networks to provide us with a better performance while the effect of adding more neurons to a layer does not necessarily have to show a positive effect. The results are once again compared in a table, which is Table 6.6.

For the data set containing daily returns, we find that all architectures need less time to complete the fitting, but this might be caused in reasons different from the design itself such as processes that happen in the CPU or GPU. It seems like for models with less data, most designs and in particular the ones with fewer neurons in one layer show improved performances and generalization quality. The best results in both the training and the test set are achieved by a deep network with only a few neurons per layer (10x10|1). The picture is different for minutely data. There is only one architecture which outperforms the standard approach: again this is the deepest network. Even though the other designs fail in improving the results, we find again that fewer neurons in a layer show the lowest errors.

Table 6.6: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin using different network designs.

Time Horizon	Optimizer	Time [s]	MSE Training	MSE Test
Daily	30|15|1	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	10|10|1	3.48	$1.689 \cdot 10^{-3}$	$1.685 \cdot 10^{-3}$
	30|30|1	4.63	$1.667 \cdot 10^{-3}$	$1.706 \cdot 10^{-3}$
	50|50|1	3.77	$1.736 \cdot 10^{-3}$	$1.780 \cdot 10^{-3}$
	50|30|1	4.90	$1.685 \cdot 10^{-3}$	$1.722 \cdot 10^{-3}$
	15|30|1	3.81	$1.694 \cdot 10^{-3}$	$1.716 \cdot 10^{-3}$
	10|10|10|1	3.43	$1.691 \cdot 10^{-3}$	$1.693 \cdot 10^{-3}$
	30|30|30|1	3.07	$1.693 \cdot 10^{-3}$	$1.719 \cdot 10^{-3}$
	30|15|10	5.38	$1.696 \cdot 10^{-3}$	$1.707 \cdot 10^{-3}$
	10|30|10|1	2.96	$1.694 \cdot 10^{-3}$	$1.701 \cdot 10^{-3}$
	50|40|30|20|10|1	2.28	$1.726 \cdot 10^{-3}$	$1.746 \cdot 10^{-3}$
	10x10|1	2.23	$1.675 \cdot 10^{-3}$	$1.667 \cdot 10^{-3}$
Minutely	30|15|1	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	10|10|1	35.05	$7.588 \cdot 10^{-7}$	$3.239 \cdot 10^{-7}$
	30|30|1	32.46	$8.036 \cdot 10^{-7}$	$3.657 \cdot 10^{-7}$
	50|50|1	30.02	$8.028 \cdot 10^{-7}$	$3.624 \cdot 10^{-7}$
	50|30|1	29.06	$8.296 \cdot 10^{-7}$	$3.918 \cdot 10^{-7}$
	15|30|1	26.24	$7.767 \cdot 10^{-7}$	$3.371 \cdot 10^{-7}$
	10|10|10|1	25.25	$7.592 \cdot 10^{-7}$	$3.230 \cdot 10^{-7}$
	30|30|30|1	29.27	$7.996 \cdot 10^{-7}$	$3.622 \cdot 10^{-7}$
	30|15|10|1	28.77	$7.646 \cdot 10^{-7}$	$3.291 \cdot 10^{-7}$
	10|30|10|1	26.47	$7.642 \cdot 10^{-7}$	$3.262 \cdot 10^{-7}$
	50|40|30|20|10|1	42.80	$7.723 \cdot 10^{-7}$	$3.356 \cdot 10^{-7}$
	10x10|1	21.70	$7.561 \cdot 10^{-7}$	$3.187 \cdot 10^{-7}$

6.1.9 Learning Rate Schedules

Learning rates can vary over the course of the training. Instead of learning the rate in every optimization step which would require another separate optimization, learning rate schedules provide the algorithm with a plan how the learning rate should be chosen at the i-th iteration. Here, we will implement networks and carry out the fitting with an Adam optimizer which is fed with different schedules. The schedules, we will investigate here, are given by

$$
\begin{aligned}
\gamma^{(i)}_{\text{constant}} &:= 0.01, \\
\gamma^{(i)}_{\text{increasing}} &:= 0.001i, \\
\gamma^{(i)}_{\text{piecewise constant}} &:= 0.1 \times 0.1^{\lfloor i/(0.2m) \rfloor}, \\
\gamma^{(i)}_{\text{exponential}} &:= 0.1 \times 0.1^{i/(0.2m)}
\end{aligned}
\tag{6.1}
$$

and 1cycle with a starting rate of 0.01, a maximum rate of 0.1 after 45% of the maximum epochs allowed and a decrease afterwards. In the equations above i is the iteration and m the maximum amount of epochs allowed for the training. Of course, the list is not complete and there exists a wide variety of other possible schedules. However, the results obtained after fitting the models can be found in Table 6.7, where we compare them to the standard model which uses a constant learning rate of 0.001.

Table 6.7: A comparison of the generalization power of FFNNs for the prediction of both daily and minutely prices of Bitcoin using different learning rate schedulers.

Time Horizon	Learning Rate	Time [s]	MSE Training	MSE Test
Daily	Default	8.53	$1.730 \cdot 10^{-3}$	$1.774 \cdot 10^{-3}$
	Constant	2.43	$1.692 \cdot 10^{-3}$	$1.692 \cdot 10^{-3}$
	Increasing	3.72	$1.716 \cdot 10^{-3}$	$1.726 \cdot 10^{-3}$
	Piecewise constant	4.84	$2.676 \cdot 10^{-2}$	$2.703 \cdot 10^{-2}$
	Exponential	3.67	$1.672 \cdot 10^{-2}$	$1.688 \cdot 10^{-2}$
	1cycle	3.27	$1.686 \cdot 10^{-3}$	$1.683 \cdot 10^{-3}$
Minutely	Default	48.39	$7.568 \cdot 10^{-7}$	$3.214 \cdot 10^{-7}$
	Constant	19.49	$7.670 \cdot 10^{-7}$	$3.303 \cdot 10^{-7}$
	Increasing	24.54	$2.341 \cdot 10^{-4}$	$2.337 \cdot 10^{-4}$
	Piecewise Constant	23.23	$1.563 \cdot 10^{-4}$	$1.559 \cdot 10^{-4}$
	Exponential	23.04	$7.854 \cdot 10^{-5}$	$7.811 \cdot 10^{-5}$
	1cycle	22.94	$1.563 \cdot 10^{-4}$	$1.559 \cdot 10^{-4}$

The data in the table yields that 1cycle schedule is the best option for daily data while the default and constant options perform best for minutely data. Taking a closer look at the step-by-step fitting progress shows us that the error is not decreasing monotonically for 1cycle when training the model for minutely data. This indicates that there is still room for improvement in the precise design of the schedule in order to take advantage of 1cycle which is the best choice in many applications. Some researchers, for example Smith et al. ([165]), argue that instead of reducing or varying the learning rate, one could increase the batch size and the momentum coefficient in the course of training the model to obtain a similar accuracy while requiring fewer parameter updates and allowing for better parallelization.

6.1.10 Some Notes on Extensions

In addition to what has been presented above, the notebook also studies the difference in the results when adapting the batch size of the training samples. We find that increasing the batch size for the daily data model from 32 to 64 leads to a slight performance improvement. The results presented here have been computed on a basic CPU. Rerunning the code on a GPU would allow us to increase the batch size even more while also adjusting the maximum amount of epochs allowed.

As the last section in the notebook shows, all the methods that individually contribute to improvement are, in general, not necessarily significantly more accurate when being combined in a single model.

The iPython notebook concludes by the implementation of an approach not presented so far, **resampling**, followed by two other methods introduced in previous chapters, cross-validation and Autoencoders for data preparation.

The idea of resampling is to list the data with regard to fixed steps of traded units instead of fixed time periods. Here, applying resampling to a data set as small as the one for daily returns is not very promissing, in particular since the trading volume increased massively over the last years. We find that the resampled data set is smaller and the fitting is less accurate when comparing the rates of MSE and average returns. Based on these findings, resampling will not be used further in the following. For detailed theory on this method, we refer to [22] where the interested reader will also find a successful application for investment decision processes in cryptocurrency markets.

Cross-validation has already been the matter of our interest in Sections 3.4.2 and 3.4.3 and aims at providing the user with a better evaluation of the model's predicitive power. When applying cross-validation, the data is split into several pairwise disjoint chunks of data. The model of choice is then trained with regard to all possible combinations of training and test sets to determine the average performance measure. Cross-validation has proven to be very helpful for a wide range of machine learning tasks ([144]). However, standard CV does not make use of all the information at hand since (financial) time series data show dependencies that can be very helpful when training a model. Moreover, the order of the data is crucial which is why the set should never be shuffled like it is for classification tasks such as image recognition. Bergmeir and Benitez evaluated the application of various variants of CV to time series data and suggest the use of **blocked cross-validation** rather than the standard version ([19]). Another popular approach is building on the concept of **forward chaining** which could be understood as a syntectic form of online learning. The model is first trained for an initial set of data observed at the most recent time points. Then, one or more observations are added to the data set and the model is trained again from scratch. This approach allows the influence of growing data sets to be traced and takes into account temporal dependencies. More detailed theory on forward chaining can be found in [5].

Even though researchers suggest methods slightly different to cross-validation, an implementation can be found in the notebook. The intuition is to provide the reader with a better understanding of this important method and demonstrate how a code could look like which can easliy be modified for tasks for which this method is more suitable.

Moreover, the notebook offers an implementation of an Autoencoder which has been introduced in Section 5.4. The purpose is to reduce the dimensionality of the (input) data, in particular in non-linear settings. We find that Autoencoders are not particulary useful in our setting of minutely data which is rather intuitive since the data includes dependencies and the samples are not very rich. To better handle anomalies we suggest the use of (exponential) smoothing methods. It could be hard to design efficient networks for predictions at exact points of time, but we are confident to build an advanced network which is better in predicting a more detailed trend in comparison to ARIMA-GARCH or exponential smoothing. However, the notebook demonstrates an exemplary implementation of an Autoencoder but the results will not be discussed here. Usually, a network with an Autoencoder structure is trained and only the first half, called the encoder, is used to reduce the dimensionality of the data. This can be very useful for clustering tasks such as customer segmentation ([7]).

6.2 Predicting Financial Markets with Advanced Neural Networks

In this section, the performance of different advanced neural network for a one-step-ahead prediction task for financial time series will be evaluated. Creating highly efficient models to predict the future would provide us with a chance to outperform the market and - way more important - would allow for significant improvements in risk management. Thus, this task has been a matter of active research since the first econometric methods have been introduced and the rise of deep learning made many reseachers apply advanced machine learning techniques on financial time series to either forecast the future or develope autonomously thinking trading bots. A comprehensive overview over the application of deep learning in financial time series forecasting can be found in [158]. A variety of exemplary approaches can be found in [48] and Jan Schmitz presented an implementation of the Transformer network for this task in [154]. In fact, these two references were the inspiration for this work and the codes used in this section are inspired by the authors' approaches. The aim is to build on them to find a model suitable for our purposes by comparing the performance of different ANNs. There are fundamentally different approaches, which is why the choice of our setting should be motivated first.

6.2.1 Model Setting

The most intuitive way to approach the task would be to focus on a single financial time series, e.g. a stock or an index, and predict the price at the next time step using past observations of the very same series. Thus, one would end up with a univariate model and for a price history which is long enough, even advanced models can be trained on the data. An exemplary prediction for the Apple stock log returns generated by a univariate RNN is shown in Figure 6.1. We find that the model can detect a trend with small magnitude, but has problems with periods of higher volatilities in the returns.

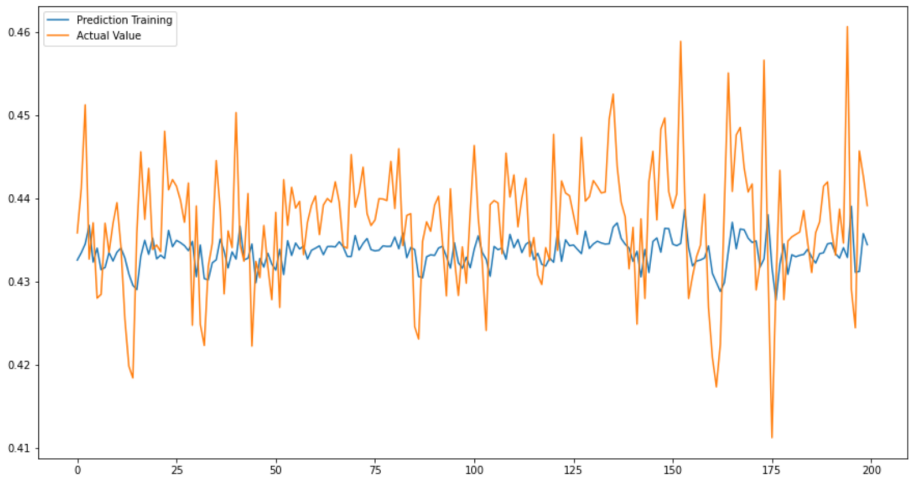


Figure 6.1: Forecasts of minutely Bitcoin returns generated by a univariate RNN model with one hidden layer of 30 neurons versus actual observations. The data set consisted of approximately 200,000 time points and the training was performed for 200 epochs and a batch size of 1000.

Another example can be found in Chapter 8 of [48] where the authors evaluated several recurrent and convolutional neural networks on predicting the USD price of Bitcoin. In the implementation of Dixon et al., α-RNN showed the best generalization performance, closely followed by GRU. However, the results look promissing, we are optimistic that including more data would improve the performance.

In a first step, the data is extended by adding more price information to the time series such as open, high and low prices as well as the trading volume in addition to the close prices that have already been taken into consideration. This leads to a multivariate model which takes five input time series to predict the returns in the next step. In his above mentioned blog post, Schmitz implemented a network of the Transformer type for a financial time series (IBM stock) and the results are summarized in Figure 6.2. For returns, the trained model generated constant predictions which is not a satisfying result. The author argues that for the rather small data set he used ($n \approx 14700$) even the most

advanced neural networks are incapable of finding more than a linear trend since they require larger data sets to be sufficiently trained. However, he suggests two approaches to cure the issue. The first is to increase the data set to several hundred or thousand time series, which is exactly what will be done in this work. The second option is to focus on moving averages of return data. By modifying the data to ten-day moving returns and retraining the network, Schmitz achieves a significantly better performance (bottom of Figure 6.2). If one trains other ANNs, such as RNN or GRU, with this data set, the predictions are once again constant almost without exception. For this reason, we will continue to adapt the setting to allow a comparison of all NNs introduced so far.

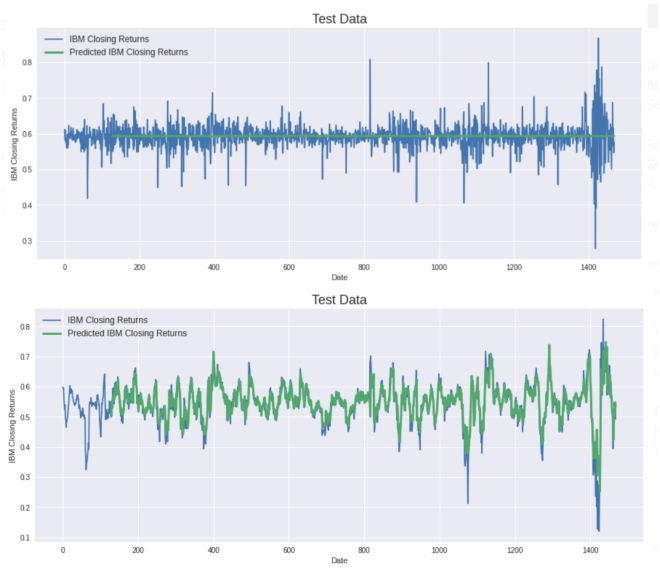


Figure 6.2: Forecasted IBM returns using a Transformer network for (top) standard return data and (bottom) ten-time step moving average return data. The figures are taken from [154].

In a next step, the data set is extended to a set of close price return time series of length $n \approx 98000$. The information from these time series will serve as the model input which then generates a univariate output, in this case the one-step ahead forecast of one of the stocks (Apple). Training the model with standard return data yields once again to a constant output, as shown for a deep RNN in Figure 6.3.

From Jan Schmitz's blog entry, we know that re-training the model with ten-step moving averages can reduce the error. Indeed, re-evaluation of the newly trained model shows a significantly better forecasting power. The results for a rather simple RNN with only one hidden layer are displayed in the graph in Figure 6.4. A closer look at the graph leads to the realization that the forecast lags behind the observations. However, since it is a simple model, we are optimistic that longer training and improved model design will lead to a more convincing result.

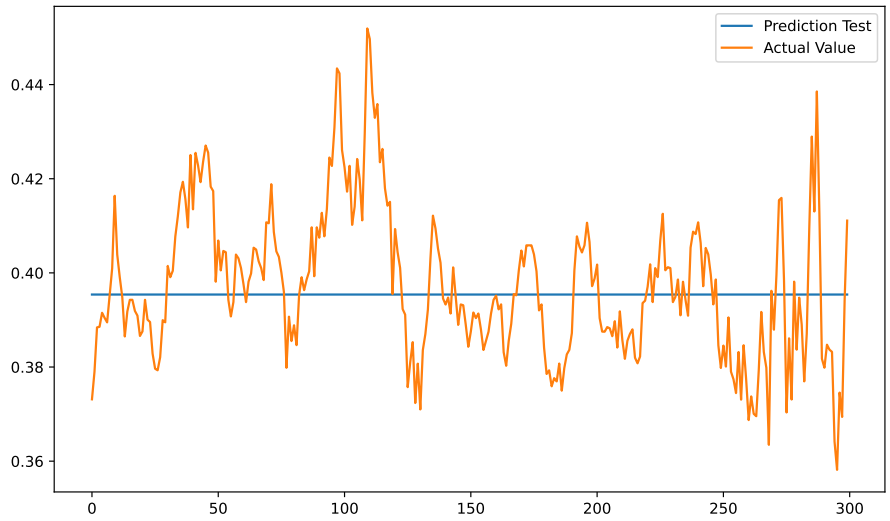


Figure 6.3: First 300 forecasts of the moving average of the Apple stock price returns generated by a RNN with three hidden layers with 20 neurons versus actual observations for a data set consisting of the minutely data for 100 stocks over one year.

Motivated by the promissing results when providing the model with a multivariate input to generate the one-step ahead forecast, it is desirable to forecast the other time series serving as a part of the input as well. This could be achieved by the creation of new networks which each produce univariate outputs. Alternatively, a model able to generate forecasts for all input time series would be advantageous since the training time would be less than training numerous models with univariate outputs one after the other and synergies could be used to our advantage. On the other hand, this approach will require a longer training time and higher computational effort compared to a single univariate model and the errors are expected to be slightly higher on average. Once again, the focus will be on ten-step moving averages instead of the pure returns in every time step. Training a RNN model of this type over 400 epochs and with a data set consisting of 100 US stocks results in predictions of which one is presented in Figure 6.5. We find that this model provides us with good predictions and is able to not only follow but also to forecast some of the events with high volatility such as the low peak at approximately $t = 670$. A drawback of this approach is that when adding regularization to the network, the forecasts will once again become constant. For this reason, regularization will be left out here and thus, the models might be more prone for overfitting. To counteract this, we add an early stopping criterion when fitting the networks which terminates the training if the model loss does not improve significantly anymore over 100 epochs and

Figure 6.4: First 200 forecasts of the moving average returns of the Apple stock price generated by a multivariate RNN with one hidden layer and 20 neurons versus actual observations for a data set consisting of the minutely data for 100 stocks over one year.

restores the best model.

The model type presented last in this section will be evaluated for an even larger data set in the following, including indices for US industry sectors. The exact procedure will be explained in the next subsection.

6.2.2 Procedure

Recalling that the aim of this section is to compare the predictive power of several advanced neural networks for multivariate financial time series, all models will be fed with the same data set. This data set contains minutely close prices for 100 US stocks included in the Nasdaq 100 index and the indices of 58 US industries by Standard & Poor's. The data ranges from 1 January 2021 to 31 December 2021, a total of 98,102 points of time. Commodities such as oil and gold are excluded from the data as well as exchange rates and currencies since, in contrast to stocks, they are traded throughout the day. Reducing the data to the standard trading hours of the New York stock markets, which stretch from 9:30 a.m. to 4:00 p.m. would lead to very high volatility between the last closing price of the previous day and the first close price the day after. Instead of prediciting the close prices themselves the focus will once again be on log returns. In the previous subsection we learned that deep but slim neural networks are advantageous in both fitting time and generalization performance and we will implement these kind

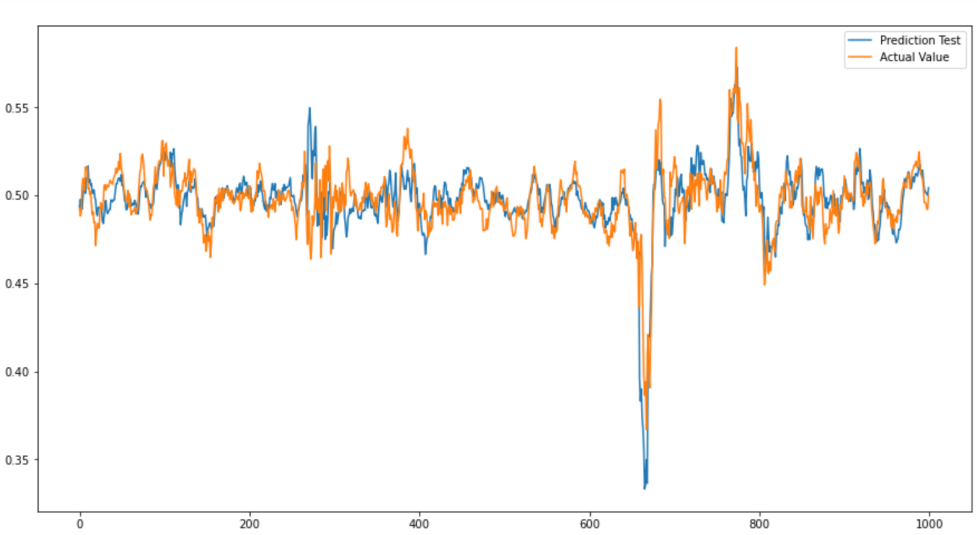


Figure 6.5: First 1000 forecasts of a stock price generated by a multivariate-to-multivariate RNN with three hidden layers and 20 neurons versus actual observations for a data set consisting of the minutely data for 100 stocks over one year.

of models whenever possible. Because of the way how the layers are designed, this is not possible for α- and α_t-RNN here. After the training has taken place, the forecasting accuracy of all models will be evaluated on a test set with regard to the root mean-squared error (RMSE). Due to limited computational resources, we will only train every model once and thus the stochasticity of the optimization algorithms and the initializations implies that the results might vary when running the notebook again. We expect the LSTM and GRU to outperform the standard RNNs while the performance of CNNs and the Transformer network is unclear, as they were developed for tasks different to time series forecasting. In addition we will train a basic feedforward model as a reference point that should be outperformed.

All in all, the procedure can be summarized as follows:

- Download of minutely data over a one-year period using the Barchart Excel plug-in;
- Import of data into a `.ipynb` notebook;
- Data preparation: replace NaNs by last close price available, compute log returns and 10-time step moving averages, scale data and transform to labeled data which can be handed to a supervised learning model, split data into training and test sets;
- Build neural networks;
- Fit and evaluate each of the neural networks; and
- Create table to compare performances.

As mentioned above, all networks will be fitted and evaluated using the same training and test set, each instance consisting of the previous 30 rolling averages and the next rolling average serving as the label for all 158 time series at hand. Furthermore, the models are compiled with regard to the ReLU activation function and the Adam optimizers. The networks also have in common that dropout (10%) and glorot initialization are applied to increase robustness. The models that shall be investigated here are: FFNN, RNN, α-RNN, α_t-RNN, LSTM, GRU, CNN, CNN with Dilation and Transformer.

A deep FFNN with five inner hidden layers of 30 neurons each will serve as a reference point for all other models. This network type has been introduced in Section 4.1 and is the most simple of them. By adding multiple layers, we hope to be able to predict more than just a linear trend but this seems rather unlikely.

The group of recurrent neural networks, introduced in Section 5.1, provides the neurons with a limited memory which helps to remember long-term dependencies. Here, we choose to fit a model with three hidden layers, each consisting of 20 recurrent neurons. α- and α_t-RNNs adapt the idea of smoothing techniques for neural networks and are thus able to handle non-stationary data. In contrast to most of the other network types considered here, the layers of α- and α_t-RNNs are not available in TensorFlow as of January 2022. Dixon et al. ([48]) implemented such layers by themselves and we will apply them at the cost of being unable to built a model with more than one hidden layer.

However, RNNs are limited in their memory's capacity. Long short-term memory network designs add a long-term hidden state and gates to the network architecture. These modifications allow for improved detection of long-term patterns and has proven to be useful in many applications such as audio processing. By simplifying the architecture of the neuron cell, GRUs offer a similar performance while theoretically reducing the computational effort in every epoch. We will implement both networks with three hidden layers consisting of 20 LSTM and GRU neurons respectively. The theory of these two network types can be reviewed in Section 5.2.

Convolutional neural networks apply filters and have been designed to detect patterns. In addition, pooling layers reduce the complexity of the processed data before it is handed over to the feedforward part of the network which is ultimately responsible for the generation of the output. CNNs extend the idea of simple autoregressive models to more complex data sets and the theory can be found in Section 5.3. By adding dilation, the learning efficiency can be increased for possible wide ranged data. The network design implemented here is based on Remark 2 in [24] which interpretes the findings in [83] and the architecture presented by Dixon et al. in Chapter 8 of [48] which consists of a one-dimensional convolutional layer with 32 filters and a kernel size of 5. This layer is followed by a pooling layer before the information is processed to the feedforward part. In contrast to the researchers, we extended the feedforward part by an additional hidden layer to make it deeper. Moreover, a similar model which uses dilation at a rate of 2 in the convolutional layer will be evaluated.

Last but not least, the focus will be on Transformer networks which build on attention mechanisms introduced in Section 5.5 and are able to process the whole input sequences at once. Thus, it is necessary to add a time encoding layer: Time2Vec. In the above mentioned blog post by Jan Schmitz ([154]), the author presents an implementation of a Time2Vec layer which we will be using here. The design of the Transformer network itself is adapted from a post by Theodoros Ntakouris ([124]) who adjusted the architecture from the original paper [184] to make it suitable for financial time series. Since Ntakouris' approach is designed for classification tasks, we change the output layer to consist of more neurons and exclude activation. Moreover, the number of attention heads is increased from 4 to 8. Due to the lack of computational resources, the training will not be carried out in a parallelized way, thus leading to a multi-hour training time.

All models except the Transformer network will be granted a maximum of 1000 epochs for training with batch sizes of 500. This choice of the batch size is adequate since the data set is very large. The first 85% of the complete data set will serve as the training data and the models will be validated during training by the last 20% of this set. In addition, early stopping with regard to the validation loss and model checkpointing are added to stop the training if a certain level of convergence is achieved and guarantee that the best model will be used effectively. The fitting of the Transformer will be limited to 50 epochs since it requires by far the most computational resources. To ultimately determine the performance of the networks, the predicitve accuracy of the model is tested for the hitherto unseen test set consisting of the remaining 15% of the complete data. It is not necessary to unscale the predictions to compare the results of each network.

The code can be found in the attached file `Comp_ANN_Final.ipynb`.

Now that the procedure and the network designs - which are far from perfect and can be modified in many ways - are explained, we can turn to the discussion of the results.

6.2.3 Results

Preparing the data as described above leads to normalized, ten-step moving averages of the log returns of the closing prices of a total of 158 financial time series. The prepared time series for the Apple stock which is one of the stocks included in the data set is displayed in Figure 6.6.

With all data being prepared, the code is run on an Intel Core i7-6500U CPU with two cores and a maximum of 2.5 GHz. Since parallelization is not applied and because of the CPU's limited capacities, the training will take a long time, in particular for the Transformer.

Table 6.8 summarizes the results of the training of all models. In this table, the number of parameters, the time needed for the training in minutes, the average of the training and test RMSEs over all time series as well as the maximum RMSEs for the test data of a single time series can be found. The plots of the first 500 predictions against the

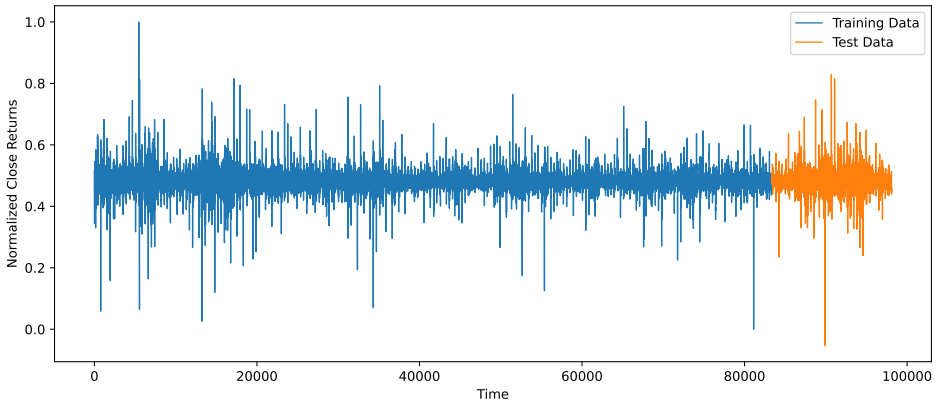


Figure 6.6: Normalized moving averages of minutely log returns of the Apple stock over a period starting on 1 January 2021 and ending on 31 December 2021. Blue indicates the data used for training and orange the test set.

actual values can be viewed as a proxy for the Apple stock in Figures 6.7 and 6.8. Here, the prediction is plotted in blue and the actual value in orange. Moreover, the history of validation losses over the stretch of the training process can be found in Figure 6.9.

Table 6.8: A comparison of the generalization power of several ANNs for the multivariate prediction of minutely returns of 158 time series.

Model	**Parameters**	**Time [min]**	**RMSE Train**	**RMSE Test**	**Max**
FFNN	150848	5.83	$2.755 \cdot 10^{-2}$	$3.087 \cdot 10^{-2}$	$6.799 \cdot 10^{-2}$
RNN	8538	83.45	$2.244 \cdot 10^{-2}$	$2.599 \cdot 10^{-2}$	$6.646 \cdot 10^{-2}$
α-RNN	18509	95.50	$1.621 \cdot 10^{-2}$	$1.901 \cdot 10^{-2}$	$5.750 \cdot 10^{-2}$
α_t-RNN	29058	146.41	$1.704 \cdot 10^{-2}$	$2.026 \cdot 10^{-2}$	$6.034 \cdot 10^{-2}$
LSTM	24198	206.97	$2.210 \cdot 10^{-2}$	$2.602 \cdot 10^{-2}$	$6.597 \cdot 10^{-2}$
GRU	19158	48.59	$2.533 \cdot 10^{-2}$	$2.810 \cdot 10^{-2}$	$6.799 \cdot 10^{-2}$
CNN	43790	26.5	$2.531 \cdot 10^{-2}$	$2.822 \cdot 10^{-2}$	$6.853 \cdot 10^{-2}$
Dil. CNN	42510	17.6	$2.749 \cdot 10^{-2}$	$3.076 \cdot 10^{-2}$	$6.792 \cdot 10^{-2}$
Transformer	5300902	1510.36	$2.240 \cdot 10^{-2}$	$2.573 \cdot 10^{-2}$	$6.658 \cdot 10^{-2}$

First of all, the results show big differences in the amount of parameters and the required time to train the model. The RMSEs for the test set stretch from $1.901 \cdot 10^{-2}$ to $3.087 \cdot 10^{-2}$, a difference of 38.4%. The fact that the test errors are slightly higher than the training errors show that we avoided overfitting to a certain extent.

Fitting the FFNN, which has a lot of trainable parameters compared to the other models,

with the data set at hand yields a constant output independent of the input data. This suggests that the network is not able to find a causality between the returns of the previous days and the next day and therefore reduces to forecast a constant trend. The training time might be the lowest of all networks, but on the other hand, the errors are the highest and this type of FFNN has to must be classified as inappropriate for the prediction of multivariate time series. However, the example in Section 4.3 showed that networks of the feedforward-type can outperform the exponential smoothing and ARIMA-GARCH approaches in a simple univariate setting.

In contrast to the first model, the RNN is able to identify causalities between the input and the labels which results in a reduced error for both the training and the test data. The plots of the predictions versus the observations show a clear pattern: the RNN seems to fit the training data well, but it struggles with extreme returns in the test data. This might be due to the well-known shortcomings of RNN, which struggle to identify long-term dependencies and are limited in their memory. However, these aspects have been addressed by more advanced neural networks which should offer a better generalization power. But there also exist more successful applications of RNNs in practice. Su et al. ([173]) propose a stochastic variant of the recurrent network type which performs well in anomaly detection for multivariate time series. Sirignano and Cont find that stacking recurrent neural networks leads to superior performance when combining intra-day data with technical market indicators ([163]). In another paper, Dixon finds that RNNs are able to outperform FFNNs for predictions of the limit order book data ([46]). The results show that RNNs outperform FFNNs in a wide range of applications and suggest that the results obtained here reflect the superiority of RNNs.

Generalizing smoothing methods results in networks called α- and α_t-RNN, where the latter one differs from the primer one in that the smoothing parameter is time-dependent. The results show that both approaches outperform the standard recurrent architecture. Surprisingly, the errors of the model with constant α are lower than the errors of the time-dependent approach. Moreover, α-RNN has the best generalization performance of all models implemented here with a test RMSE of $1.901 \cdot 10^{-2}$. Taking a look at the plots, the predictions look very promissing and both models are able to predict the correct tendency for the next day even though the magnitude is not always accurate and a few predictions, e.g. for $t \approx 450$, are far off, thus leaving room for improvements. Taking a closer look at the plots available in the notebook, we find that even though the models fit very well for most time series, there are several series which show test errors that are significantly higher than the training errors, yielding underfitting. This pattern is consistent for both stocks and indices. The evolution of the validation losses in Figure 6.9 suggests that both models are able to find ways to proceed in their descents. Furthermore, the performance boost compared to standard RNNs does not come at high costs since the training times are higher but they do not explode even though the number of paramters in this setting is a lot higher. The results of both networks are impressive, in particular when taking into account that they consist of only a single

hidden layer with 50 neurons. Adding additional layers or a feedforward structure at the end of the network could possibly provide the models with an even better prediction accuracy. Dixon and London demonstrate that smoothed RNNs are well-suited for the forecasting of financial time series while requiring, in contrast to the models evaluated in this work, fewer parameters than other modifications of recurrent networks. In addition, this approach allows processing non-stationary data and thus removes the requirement of pre-processing ([49]).

Contrary to our expectations, LSTM and GRU perform worse than the smoothed models, and the errors of GRU are even higher than the standard version. Comparing the plots for several time series shows that both models struggle with the prediction in volatile periods and for some series they are merely able to identify a trend, e.g. for the AMD stock. The plots in Figure 6.7 also indicate that the GRU is more prone for guesses that are far off. Moreover, the history of the validation loss shows that the error for GRU increases towards the end of the fitting process and thus, the performance could have been even worse without an early stopping criterion. On the other hand, the GRU could have simply run into a local minima and the process of moving towards a different minimum was interrupted by the criterion. The results presented in Table 6.8 show that, as expected, GRU, which is a simplified version of LSTM, reduces the computational time needed and the number of parameters required while offering a degraded performance. GRUs have proven to be comparable to LSTMs when the data sets are limited and not as large as the one used in this section which is a possible explanation for the deterioration. However, the results indicate that the design of the networks is not perfect and could be improved to at least outperform standard RNNs. In general, LSTM has not been applied successfully to financial time series prediction tasks. In fact, only a handful of researchers ever claimed that there network is working, e.g. in [163] where the authors trained a LSTM network with a massive, multiple TeraByte data set containing 1000 stock time series using approximately 500 GPU nodes. Moreover, the input data also included the public order book which is the basis of future prices. However, the amounts of data and the computational resources used by the authors is hardly available for private users and thus the results cannot be reproduced in the scope of this work. However, in a 2018 paper, Siami-Namini and Namin demonstrate that in a similar setting to the setting presented here, the application of deep learning and LSTMs leads to improved generalization power compared to ARIMA models ([161]). Another example of a successful application of LSTMs to financial time series forecasting can be found in [57]. The authors use LSTM to predict out-of-sample directional market movements of the stocks of the S&P500 index and the model yields improved results when being compared to deep feedforward networks. Obviously, the task of detecting a trend or direction in the market does not require as much detail as the forecasting of minutely returns and thus might be more suitable and easier for all types of recurrent neural networks.

For the models discussed so far, the results obtained here agree with Dixon, who tested them for a univariate prediction task in Chapter 8 of [48], in that α-RNN offers the

best performance, even if the improvement over the basic RNN is not as pronounced. In contrast to the findings obtained here, Dixon's GRU shows a significantly better result than LSTM and all modified recurrent-type networks outperform the basic RNN. Even though the model designed here have been inspired by Dixon's, the task is different since the focus of this thesis is on multivariate forecasting. This difference, together with the fact that Bitcoin prices are even more volatile than those of stocks, could be an explanation for the non-concordance of the results.

The convolutional neural network trained here shows a performance very similar to the GRU's. While being able to identify a basic trend and to adjust to longer periods of extreme returns, the model is unable to forecast the strong price fluctuations at all. This seems reasonable since the filtering that takes place in CNNs leads to the recognition of general patterns and thus, not every detail is processed by the network. According to this argument, CNNs might perform better in forecasting long-term trends in the markets, e.g. over a period of several months or years. Despite offering more parameters, the fitting proceeded faster than for GRUs. Adding dilation leads to deterioration in the generalization power. The performance of the dilated CNN is comparable to the FFNN, $3.087 \cdot 10^{-2}$ versus $3.076 \cdot 10^{-2}$, and we conclude that this model variant is not suitable for financial time series forecasting for multivariate high-frequency data. However, Borovykh et al. ([23]) designed a convolutional network with dilation based on the famous WaveNet paper ([125]) which offers a solid baseline for financial time series forecasting and can be regarded as a valid alternative to LSTM which they refer to as the state-of-the-art network for day-to-day time series forecasting. The authors point out that there is still a lot of room for improvement and that the main challenge, the trade-off between overfitting and adding many layers and filters to better detect non-linearities, should be in the focus when further improving the network design.

Last but not least, the results of the Transformer network will be evaluated. The training took more than 25 hours which is by far the largest amount of time needed of all networks. This can be explained by the high number of parameters and the more complex structure of the Transformer which results in the fact that instances require more time when being passed through the network. The errors are similar to the results for RNN and despite the intense training process, the Transformer network is not able to compete with the smoothed RNNs. However, the predictions that can be observed in the plots look more promissing than the forecasts generated by the FFNN, LSTM, GRU and CNNs since they are rarely completely off. Taking a closer look at the history of the validation loss, one recognizes that the loss function at a certain point suddenly increases and subsequently runs into a plateau. If one of the models of the later epochs would have been used to generate the forecasts, it would once again have produced constant outputs. This finding indicates that the architecture of the network is not mature and further tuning methods need to be applied to increase the stability of the training process. Although the basic idea of Transformer networks seems to be very promissing for financial time series forecasting, the design, which is almost identical to that for machine translation

from the original paper, does not seem to translate successfully to the task at hand. In a further step, the design would therefore have to be adapted to the task in order to be able to adequately exploit the advantages of the Transformer. To the best of the author's knowledge, there has not been a remarkably successful application of a Transformer to multivariate financial time series forecasting in a supervised learning setting which is often regarded as one of the hardest tasks for machine learning models. Surely, in the near future a modified model for this type of task will be presented and is thus still a matter of active research. However, Zerveas et al. ([191]) recently introduced a Transformer-based framework which included pre-training. The authors state that theirs is the first unsupervised method to push the frontiers of state-of-the-art performance for both regression and classification tasks. Since this approach is unsupervised, it can hardly be compared to a supervised Transformer. A different paper by Hollis et al. ([81]) found that adding attention to LSTM networks slightly improves the performance when predicting one-step ahead prices. However, their approach has several limitations, one of which is that online learning cannot be realized.

Before we conclude this chapter, the reader should be reminded that the results might not be representative since the training is based on stochastic optimizers and in contrast to the evaluation of tuning methods only one model has been trained here for each network type. Furthermore, predicting financial time series will always be a very hard task for machine learning models and the markets are driven by many factors some of which can be captured and quantified, but others cannot. In addition, there are interesting effects that can be observed in the markets and that should be included. Examples include the calendar month effect ([192]) and news sentiment, which are again the result of machine text processing ([66]). Generally speaking, intra-day forecasting is a lot more difficult than day-to-day forecasting and, on the other hand, predicting the value of a time series several steps ahead might be more relevant for trading and risk management.

6.2.4 Comparison with Univariate Approach

Since the results observed in the previous subsection differ from our expectations and most findings in the literature, the performance will be compared to a univariate modeling approach. The procedure is the same as described above despite the fact that instead of handing over a data set consisting of 158 time series, the networks will be fed with the ten-step moving average returns of a single stock, in this case the Apple stock. However, we will only evaluate the predictive performance of α-RNN, LSTM, GRU, CNN and CNN with dilation here. The implementation can be found in the attached file `Comp_Uni.ipynb`.

The results are displayed in Table 6.9 and we find that all models improved their performance when being trained only for a single time series. This result is surprising in that, although the models in the multivariate setting had more market information available, performance did not improve significantly. It seems like the trade-off between the

Table 6.9: A comparison of the generalization power of several ANNs for forecasting the Apple stock returns.

Model	Parameters	Time [min]	RMSE Train	RMSE Test
α-RNN (multivariate)	18509	95.50	$1.437 \cdot 10^{-2}$	$1.705 \cdot 10^{-2}$
α-RNN (univariate)	2652	24.89	$1.148 \cdot 10^{-2}$	$1.345 \cdot 10^{-2}$
LSTM (univariate)	8341	149.81	$1.744 \cdot 10^{-2}$	$2.024 \cdot 10^{-2}$
GRU (univariate)	6441	54.48	$1.382 \cdot 10^{-2}$	$1.612 \cdot 10^{-2}$
CNN (univariate)	8973	12.36	$1.211 \cdot 10^{-2}$	$1.455 \cdot 10^{-2}$
Dil. CNN (univariate)	7693	9.1	$1.310 \cdot 10^{-2}$	$1.542 \cdot 10^{-2}$

amount of information processed and the efficiency of information is not balanced in this setting and with the model architectures at hand. However, it should be noted that this is only the sample for one share and the training of the large models is significantly more complex, which points to the need for further development of these. A solution could be to add more neurons and thus more connections to the large network to cope with the amount of data and compensate for the accuracy lost. However, the α-RNN continues to deliver the best predictions in this new setting, while now the results for GRU and the CNNs appear significantly better. Figure 6.10 compares the predictions generated by α-RNN in a multivariate and a univariate setting and shows that the forecasts are very accurate for the latter. Moreover the history of the validation losses over the training epochs are displayed.

We conclude, that handing high quality data to the network and limiting its complexity can be useful to reduce the prediction error. The costs of having a single multivariate model to predict multiple time series are significant even though the training time is reduced compared to training hundreds of models. The time saved could be invested by more detailed training and evaluation of the model. However, the multivariate model might generalize better in that it should outperform the univariate models when being fed with time series of unknown stocks and indices. The development of methods and network designs to achieve the desired results will be left for further research.

6.3 Summary

In a first section, this chapter aimed at comparing the tuning methods introduced in the course of this work. They were compared for a feedforward network design and data sets containing of minutely and daily returns of Bitcoin. Normalization turned out to be helpful to avoid exploding gradients in the first layer, but the effect is not very pronounced since log returns are already within a small margin and thus the model is unlikely to run into issues. However, the model trained with normalized data will serve as

the model-to-beat in the following. In a next step, we find that applying normalization to every layer does not improve the generalization power and this finding coincides with the literature. Applying dropout in the course of training the network made it more robust und consequently reduced both the training and the test error for both data sets. On the other hand, initializing the layer weights and biases following Glorot's approach did not show a positive effect since the weights and biases are already initialized following a similar approach by default in Keras. By adding regularization when fitting the models, we achieved the desired improvement and reduced the probability of running into overfitting by penalizing the extensive use of parameters. However, in this setting the magnitude of the regularization parameter did not have a great influence on the results. Larger differences could be found when varying the different activation functions. It turned out that the hyperbolic tangent is an appropriate choice for the task at hand even though the training required the most time, while the often used ReLU activation performed significantly worse. Regarding the optimizers, the Adam algorithm proved to be the one of the better options even though Nadam, which was introduced to cure the shortcomings of Adam, slightly improved the results. Testing models with different amounts of neurons and layers yields that deep networks are advantageous over wide networks and this result coincides with recent trends in machine learning where deep learning is becoming more and more important. In a final comparison, the impact of modified learning rate schedules is evaluated. It can be seen that all schedules reduce training time, but also that the results are inconclusive and the supposed superiority of 1cycle, which struggles for minutely data, cannot be confirmed in this context.

In addition to these evaluations, the attached notebook also contains exemplary implementations of cross-validation, resampling and an Autoencoder. All of these methods can be useful to reduce the complexity of the training or improve the evaluation of model over the complete data set, but will not be investigated further in this thesis.

In the second section, the performances of different advanced neural networks are compared. Before the training takes place, the model setting is motivated. After testing univariate settings, evidence has been found that more advanced models are not able to detect more than a trend and ultimately producing only constant forecasts independent of the input data. We decided to proceed with a multivariate setting which requires more computational effort but includes more data and thus allows all advanced models to create non-constant outputs. Moreover, a multivariate model might generalize well to time series on which it has not been trained. However, this only holds true if the moving average of the return data at hand is fed to the networks. The networks are trained and evaluated for a data set containing of 158 financial time series, 100 US stocks and 58 US industry indices that have been downloaded. The models are trained and then evaluated on a test set in terms of the root mean-squared error. To avoid overfitting and bad model selection, an early stopping criterion and model checkpointing have been included in the implementation. After training each model once, the results have been compared and we find that smoothed RNNs provide us with the best generalization power

even though the networks designs include only a single hidden layer. Unlike some findings in the literature, LSTM, GRU, CNN and Transformer networks did not outperform the previously mentioned models. However, the models are based on simple designs that could be improved and adapted to this type of task. Moreover, the usefulness of several combinations of tuning methods for different network types would have to evaluated for every single model. Hence, the results are not representative for other applications and the application of advanced neural networks to multivariate financial time series forecasting is still a matter of active research and regarded as one of the hardest tasks in machine learning, in particular for intra-day data.

After covering the theory, comparing the impact of tuning techniques and evaluating the predictive power of a variety of neural networks, we will conclude this thesis by summarizing the most important learnings and by giving an outlook on challenges and the future of neural networks in finance.

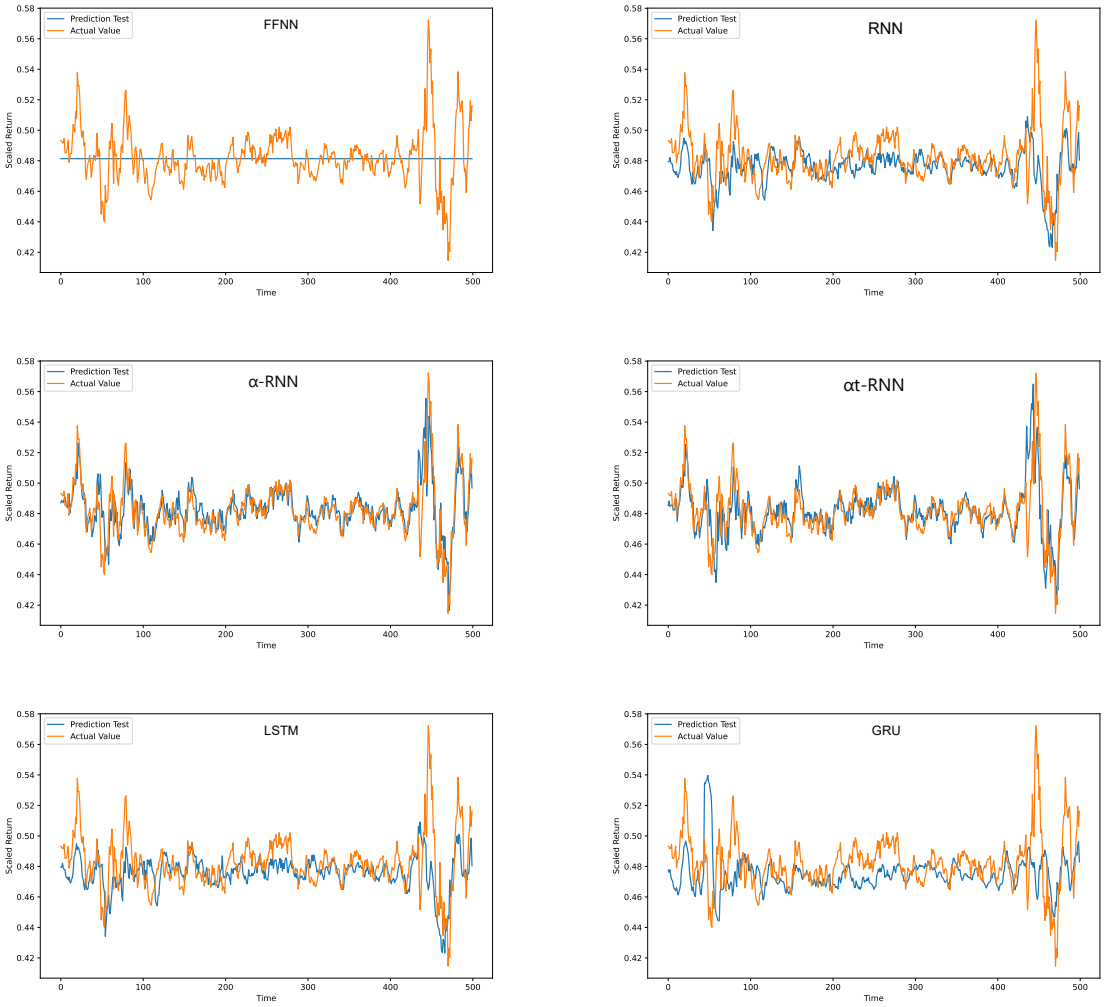


Figure 6.7: Comparison of the predicted versus the observed values at the first 500 time steps of the test set. From left to right and bottom to top: FFNN, RNN, α-RNN, α_t-RNN, LSTM and GRU.

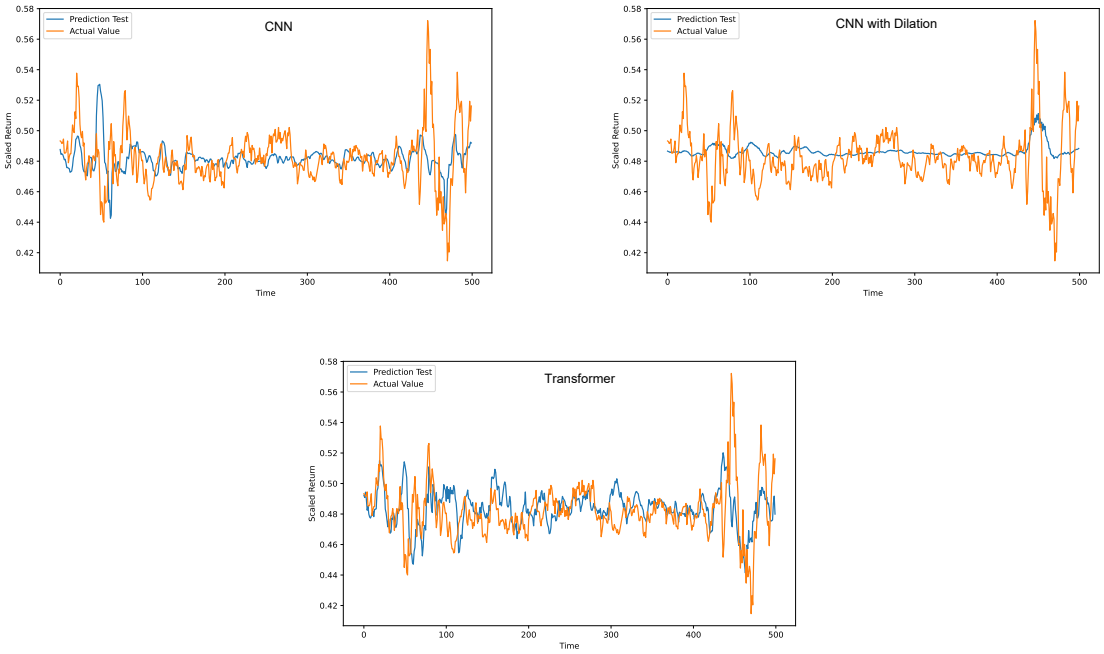


Figure 6.8: Comparison of the predicted versus the observed values at the first 500 time steps of the test set. The models are (left) CNN, (right) CNN with Dilation and (bottom) Transformer.

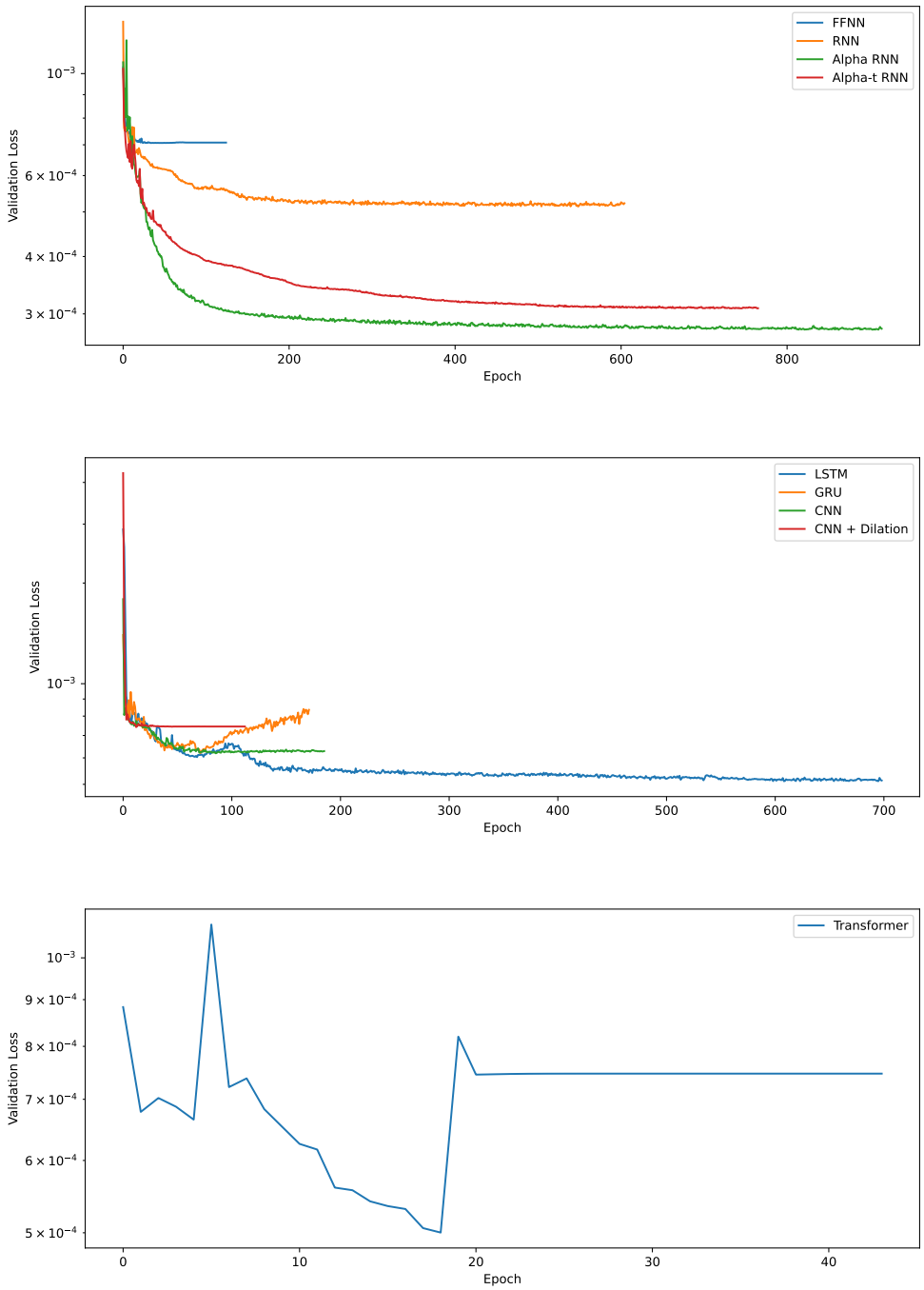


Figure 6.9: Comparison of the validation loss over the training epochs. Note that the *y*-axes use a logarithmic scale.

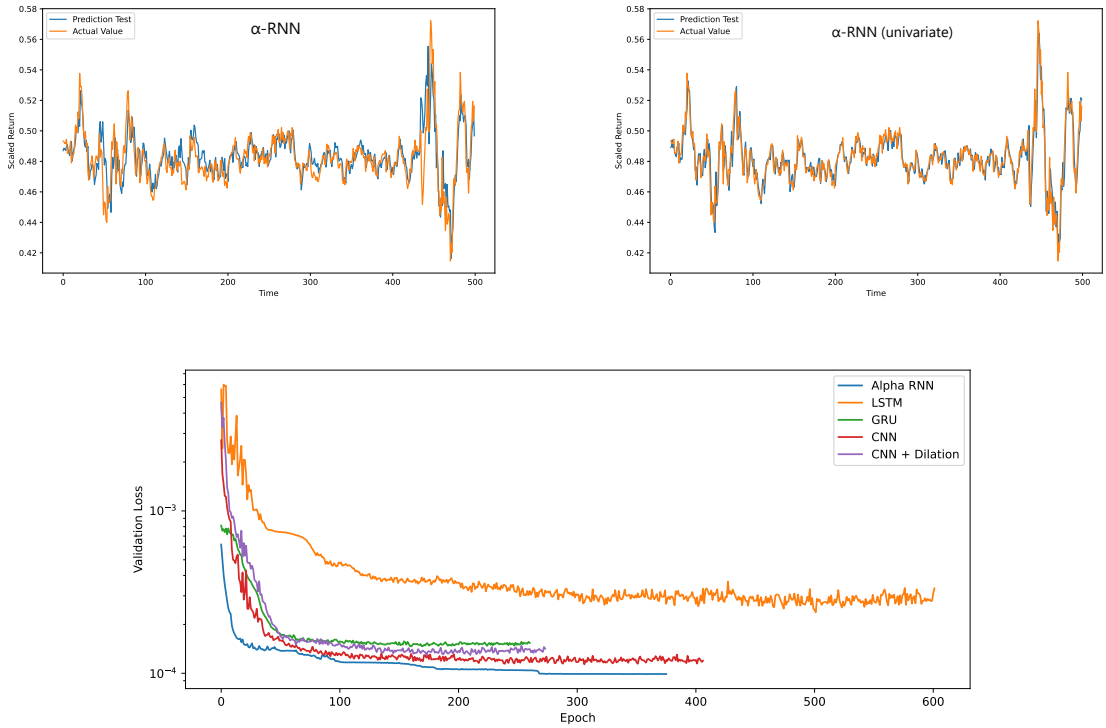


Figure 6.10: First 500 predictions of Apple stock returns generated by (left) a multivariate α-RNN and (right) a univariate α-RNN. (Bottom) History of validation loss for models trained with univariate data.

7 Summary and Outlook

At the beginning of this work, the basic concepts of financial time series have been introduced. The Wold representation theorem guarantees that every covariance stationary time series can be written as the weighted sum of past observations and a weighted stochastic time series. Building on this result, the two most common autoregressive models, ARMA and GARCH, have been introduced. In the following, the Box-Jenkins approach for the fitting of these type of models has been presented. An exemplary application of this approach showed that ARIMA-GARCH models are limited in their capability to detect more than a long-term trend and exponential smoothing turned out to be a more powerful alternative resulting in a lower mean-squared error on the test set.

In the following chapter, the motivation of machine learning has been framed and can be summarized as the extrapolation of a mapping. Machine learning can be classified in several ways with the most important being the differentiation between unsupervised, supervised and reinforcement learning. The bias-variance trade-off is identified as the core challenge of machine learning for both probabilistic and non-probabilistic modeling approaches. Probabilistic models can be fitted or trained by learning from the gradient of the loss function and applying a descent algorithm. Regularization is recommended to avoid running into overfitting which means that the model performs well on the training, but bad on the test set, indicating a limited generalization power. In a probabilistic context, maximum likelihood and maximum a posteriori estimations play a key role for the process of finding the best parameters which define the distribution of the posterior. When it comes to choosing the best of a set of models, cross-validation and Bayesian model selection can be helpful to determine the degree of generalization power over the whole data set.

In Chapter 4, the components of an artificial neural network and its neurons are explained. The networks are parametrized by the weight matrices for every layer and the signal resulting from the inputs and the weights is passed through an activation function before it is forwarded to the next neurons. Even though the universal representation theorm guarantees that any continuous function can be sufficiently approximated by a feedforward neural network with one layer, the use of deep networks promises higher efficiency and reduced training costs. The training of FFNNs builds on the chain rule of differentiation and backpropagation is introduced as the standard optimization method with regard to a loss function. The idea of backpropagation, which is based on gradient descent to iteratively adjust the model parameters, is expanded by the inclusion of momentum strategies and learning rate decay, ultimately resulting in the Adam algorithm. A

convergence result for this algorithm, which has proven to speed up the training process in a wide range of applications, is derived in the very specific setting of a convex loss function. The chapter concludes by an example of how feedforward neural networks can be applied in practice to predict the log returns of the close prices of the S&P500 index. When comparing the performance with ARIMA-GARCH and exponential smoothing, we find that the model outperforms both in terms of the test set's root mean-squared error.

Since this simple type of neural network has already shown better prediction power than classical econometric models, we turn to advanced neural networks with great interest. By adding inner hidden states to the network design, the models are enabled to develop a memory and incorporate knowledge from previous passes into the current signal generation process. These networks are called recurrent neural networks and can be trained with the optimization techniques derived from backprogatation by unrolling the layers through time. It is shown that RNNs generalize the concept of AR processes and that the networks can be generalized in a way how GARCH processes generalize ARMA processes. The application of exponential smoothing leads to α-RNNs for constant, and α_t-RNNs for dynamic smoothing parameters. These two models extend the network's memory and are able to handle non-stationarities in the data. Long short-term memory networks and its simplified counterpart gated recurrent units further improve the memory of the network by introducing gating. GRUs are arguably the most popular recurrent network by now and improvements in convergence and generalization power led to the two networks being considered as state-of-the-art for various time series processing tasks. A different type of network that exploits the spatial structures in the data and reduces its complexity is called convolution neural networks. Here, filters and pooling layers are applied to detect specific patterns in the data. Mathematically speaking, convolution is based on the concept of weighted moving average smoothers and can also be thought of as another generalization of an autoregressive model. CNNs have originally been introduced for image processing but can also be applied to time series. Moreover, adding dilation to the network increases the model's memory and is particulary helpful for long input sequences. Next, a network structure with a different purpose is introduced to the reader. The architecture of Autoencoders is inspired by the encoder-decoder structure of a class of RNNs and extrapolates the mapping $F(X) \approx X$. By passing data only through the first half of the network, the dimensionality can be reduced. This can be regarded as a generalization of principal component analysis to non-linear dependencies. In the final section, attention networks are introduced. These networks are able to determine which features of an input sequence are the most important. Combining multiple attention mechanisms, positional or time encoding and feedforward layers leads to the Transformer architecture. Transformers are regarded as the state-of-the-art for many tasks connected to time series processing while being able to handle a very long input sequence at once. Furthermore, the training can be parallelized easily which reduces the computational time required significantly. It should always be kept in mind that the networks presented here were developed primarily for classification tasks and not for the generation of precise predictions. For this reason, it is to be expected that the predictions will have a significant

error.

After having introduced a variety of concepts to improve the training process and generalization power of neural network, we are interested in how they perform in practice. For two data sets, one consiting of daily return data and one consiting of minutely return data of Bitcoin, we find that most tuning methods are helpful. Scaling the data helps in avoiding exploding or vanishing gradients, applying dropout when training the network and randomly initializing the weights and biases improve the generalization power and increase the robustness of the models while batch normalization, in accordance with the literature, is not helpful in this setting. Regularization can help the optimizer to avoid running into overfitting and an appropraite choice for the activation function applied after the layers also has a positive impact on the fitting time and loss function. The Adam algorithm performs well for the task at hand and only Nadam leads to a better result for both data sets. Testing several different network designs indicates that deep and slim neural networks should be preferred over short and wide models since the fitting time is reduced while the error is significantly lowered. In a final comparison, we learn that handing a custom learning rate schedule to the optimizer can improve the performance but the results are inconclusive and it is not possible to suggest the usage of a particular scheme. In addition to the discussion of these methods, the notebook also demonstrates exemplary implementations of cross-validation, resampling and Autoencoders. The second part of Chapter 6 deals with the comparison of different advanced neural networks for a multivariate prediction task. The data includes moving average returns of 158 time series from the US markets and the models are trained with regard to the root mean-squared error. Since regularization terms could not be added in the fitting process without limiting the models to generating constant predictions, an early stopping criterion and model checkpointing are implemented to reduce the risk of overfitting. The architectures of the models are inspired by the design presented in the literature and the Transformer architecture is almost identical to the design proposed in the original paper. We find that α- and α_t-RNNs perform best among the models which is surprising since in particular LSTM and Transformer networks are regarded as the state-of-the-art in time series processing. The designs offer a lot of room for numerous adaptations and the aim of this work is not to find the optimal design - and this task could prove too demanding anyway - but to demonstrate an overview of a possible application to financial time series. Thus, the results should not be considered representative. However, we can conclude that Transformers and CNNs should not be applied to financial time series without being revised and modified for this task which is still regarded as one of the most challenging assignments in machine learning. After all, most of the models seem to be able to identify long-term trends and perform better than the linear econometric models even though the error rate is still high and it cannot be recommended to solely rely on the forecasts of any of the models. It would be interesting to further investigate the performance of the models for day-to-day data and a forecasting horizon of multiple time steps into the future. Also, identifying trends and seasonalities in advance might lead to improved performances.

The results in Chapter 6 raise the question how a neural network for the specific task of time series forecasting should be designed. In the following, a proposal for the construction of such a network will be presented, taking into account the knowledge gained in this work. In general, this network should be large to be able to handle a huge amount of data. A large network could process more information and since the market prices depend on many factors such as other assets in the markets, international markets, commodities, sentiments and many more, this characteristic would be advantageous. The first building block of the proposed design illustrated in Figure 7.1 is the input layer. It is followed by an embedding layer which adds time or positional encodding information corresponding to the input features to the data. For example, one could use a Time2Vector layer here, as we have included in the architecture of the transformer in Chapter 6. To reduce the complexity and dimensionality of the data and to increase the depth of the information by applying filters and pooling layers, it is sent through an encoder consisting of convolutional layers. The encoder is trained by fitting a complete convolutional Autoencoder to the data set. Subsequently, attention mechanisms learn to detect the most relevant parts of the information and pass the signals to a layer of recurrent neurons in combination with smoothing techniques. This part of the network is the main part and inspired by the smoothed RNNs which performed best among our models for financial time series forecasting. Finally, the information is sent through a deep FFNN which then forwards a final signal on which a linear activation function has been applied to the output layer.

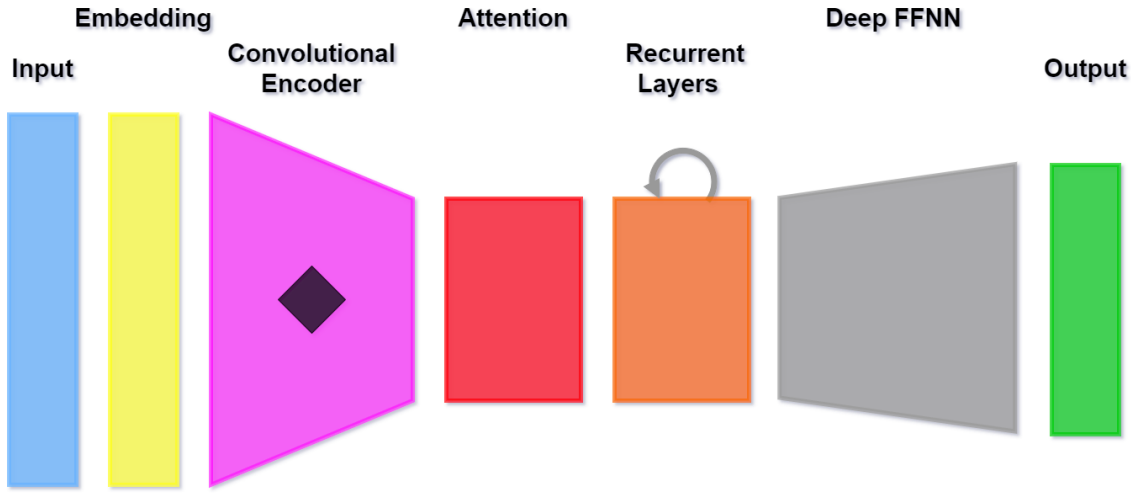


Figure 7.1: Proposed design of a neural network for financial time series processing.

The idea of this network is to combine the benefits of the advanced models presented in the course of this work. However, it is not guaranteed that this design provides any improvements when being trained for a practical application.

The model proposed above and the networks that have been evaluated in Chapter 6 are

all of the generative type, i.e. they perform regression to generate a new value. However, there exist tasks which ask for classifications instead of exact values. Models of this class are called discriminative. One example of a model of this class are algorithmic traders who make automated investment decisions based on neural networks. The theory and an implementation can be found in [159]. The networks presented in this work could be adjusted to make use of them for trading decisions as well. By expanding the reference period, drastically increasing the forecast horizon and equiping the output layer with softmax or sigmoid activation, the model is enabled to generate classifications and thus give investment advice. In this setting, the classification would not be binary and the output value y generated by the network should be interpreted as the model's advice with $y = 0$, or $y = -1$ in case of the softmax activation function, indicating that an investment should be avoided in any case and with $y = 1$ strongly suggesting the client to place an order to buy this asset. The first case thus also suggests selling a position from the depot, buying put options or, if possible, entering a short position. The outputs, i.e. the recommendations generated by this model, can then be handed over to a second algorithm which is basically a deep neural network trained to execute orders with optimal timing within a given time frame. Both networks should be built suitable for reinforcement learning and exemplary designs and application can be found in Chapters 9, 10 and 11 in [48]. Moreover, an example of a deep neural network based on recurrent layers and the reinforced learning approach can be found in [42].

The machine learning landscape changes quickly and what is state-of-the-art today will be outdated again in a few months or years. Nevertheless, a brief overview of current trends is given here. With the rise of improved traning algorithms and optimization techniques, the focus has shifted from rather simple to deep neural networks which offer improved performance while requiring less parameters to be trained and being able to handle larger data sets. The developments in this area of increasing attention are summarized in [158]. According to Ozbayoglu et al. ([127]), LSTM as well as its derivatives and modifications are the most dominant network among those suitable for deep learning. LSTM-type networks are able to detect the time-varying data representations and works well for regression-type models. An example for a successful application to the prediction of cryptocurrency prices can be found in [167]. Convolutional neural networks (CNNs) have also been in the focus of research and offered good predictive power for data that could be transformed to stationarity. In particular, filtering, and thus dimensionality reduction, have proven to be useful in financial applications, e.g. in [70]. Moreover, the popularity of deep reinforcement learning, in particular agent-based variants as shown in [33], and hybrid models, such as [85] that merge the benefits of two model types into one, in particular deep networks equiped with attention mechanisms, is continuing to rise.

The field of implementations areas for these network types is vast. In the context of finance, price or trend prediction and algorithmic trading are receiving the most interest (e.g. CNN for forex trading in [101]). Other important fields of research and practical applications are risk assessment tasks such as credit scoring ([120]), fraud detection

([148]), portfolio selection ([2]), asset and derivative pricing ([91]), cryptocurrency price prediction ([168]), sentiment analysis ([86]) and financial text mining ([4]).

In summary, it can be said that machine learning, and deep learning in particular, are becoming increasingly important. The future will probably be hybrid models that combine classical NNs with attention mechanisms and are fed with data generated by text mining and sentiment analysis. Whether these developments will lead to crises like the one in 2007/2008 being avoided and prosperity being better secured in times of crisis cannot be forecasted with certainty by the best and largest neural networks.

> In the end, everything will be okay.
>
> If it's not okay, it's not yet the end.
>
> (Fernando Sabino)

List of Abbreviations

ACF	Autocorrelation Function
AD	Automatic Differentiation
ADF	Augmented Dickey-Fuller Test
AE	Autoencoder
ANN	Advanced Neural Network
AR	Autoregressive Process
ARCH	Autoregressive Conditionally Heteroscedastic Process
ARIMA	Autoregressive Integrated Moving-Average Model
ARMA	Autoregressive Moving-Average Process
BN	Batch Normalization
DNN	Deep Neural Network
EGARCH	Exponential GARCH Process
ELU	Exponential Linear Unit
FFNN	Feedforward Neural Network
FTS	Financial Time Series
GARCH	General Autoregressive Conditionally Heteroscedastic Process
GARCH-M	GARCH Process in the Mean
GD	Gradient Descent
IGARCH	Integrated GARCH Process
iid	independent and identically distributed
MA	Moving-Average Process
MAE	Mean Absolute Error
MAP	Maximum A Posteriori Estimation
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MSE	Mean Squared Error
NAG	Nesterov Accelerated Gradient
NN	(Artificial) Neural Network
PACF	Partial Autocorrelation Function
PCA	Principle Component Analysis
ReLU	Rectified Linear Unit Activation Function
RL	Reinforcement Learning
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SP	Stochastic Process
SVM	Support Vector Machines
TGARCH	Threshold GARCH Process
UL	Unsupervised Learning

List of Attached Files

- `alphaRNN.py` (taken from [48])
- `alphatRNN.py` (taken from [48])
- `ARIMA_GARCH_FIT.ipynb`
- `ARMA_Examples.ipynb`
- `Comp_ANN_Final.ipynb`
- `Comp_FFNN.ipynb`
- `Comp_Uni.ipynb`
- `FFNN_for_FTS.ipynb`
- `FTS_Intro.ipynb`
- `GARCH_Example.ipynb`
- `Plot_Activation_Functions.ipynb`
- `Plot_Advanced_AF.ipynb`
- `Market_Data.xlsx`

Bibliography

[1] Peter Adby. *Introduction to optimization methods*. Springer Science & Business Media, 2013.

[2] Saurabh Aggarwal and Somya Aggarwal. Deep investment in financial markets using deep learning models. *International Journal of Computer Applications*, 162(2):40–43, 2017.

[3] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*, pages 199–213. Springer, 1998.

[4] Md Shad Akhtar, Abhishek Kumar, Deepanway Ghosal, Asif Ekbal, and Pushpak Bhattacharyya. A multilayer perceptron based ensemble technique for fine-grained financial sentiment analysis. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, pages 540–546, 2017.

[5] Ajlan Al-Ajlan. The comparison between forward and backward chaining. *International Journal of Machine Learning and Computing*, 5(2):106, 2015.

[6] Peshawa Jamal Muhammad Ali, Rezhna Hassan Faraj, Erbil Koya, Peshawa J Muhammad Ali, and Rezhna H Faraj. Data normalization and standardization: a technical report. *Mach Learn Tech Rep*, 1(1):1–6, 2014.

[7] Maha Alkhayrat, Mohamad Aljnidi, and Kadan Aljoumaa. A comparative dimensionality reduction study in telecom customer segmentation using deep learning and pca. *Journal of Big Data*, 7(1):1–23, 2020.

[8] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[9] Amidi, Afshine and Amidi,Shervine. Recurrent Neural Networks cheatsheet. Accessed on 04/10/2021, https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks#architecture.

[10] Theodore W Anderson. *The statistical analysis of time series*, volume 19. John Wiley & Sons, 2011.

[11] Tom M Apostol and CM Ablow. Mathematical analysis. *Physics Today*, 11(7):32, 1958.

[12] Larry Armijo. Minimization of functions having lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[13] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[15] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18, 2018.

[16] Edward J Bedrick and Chih-Ling Tsai. Model selection for multivariate regression in small samples. *Biometrics*, pages 226–231, 1994.

[17] Yoshua Bengio, Li Yao, Guillaume Alain, and Pascal Vincent. Generalized denoising auto-encoders as generative models. *arXiv preprint arXiv:1305.6663*, 2013.

[18] Fred Espen Benth, Jurate Saltyte Benth, and Steen Koekebakker. *Stochastic modelling of electricity and related markets*, volume 11. World Scientific, 2008.

[19] Christoph Bergmeir and José M Benítez. On the use of cross-validation for time series predictor evaluation. *Information Sciences*, 191:192–213, 2012.

[20] Sebastian Bock, Josef Goppold, and Martin Weiß. An improvement of the convergence proof of the adam-optimizer. *arXiv preprint arXiv:1804.10587*, 2018.

[21] Tim Bollerslev. Generalized autoregressive conditional heteroskedasticity. *Journal of econometrics*, 31(3):307–327, 1986.

[22] Tomé Almeida Borges and Rui Neves. *Financial Data Resampling for Machine Learning Based Trading: Application to Cryptocurrency Markets*. Springer Nature, 2021.

[23] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. Conditional time series forecasting with convolutional neural networks. *arXiv preprint arXiv:1703.04691*, 2017.

[24] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. Dilated convolutional neural networks for time series forecasting. *Journal of Computational Finance, Forthcoming*, 2018.

[25] Philippe Bougerol and Nico Picard. Strict stationarity of generalized autoregressive processes. *The Annals of Probability*, 20(4):1714–1730, 1992.

[26] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[27] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[28] Michael W Brandt and Christopher S Jones. Volatility forecasting with range-based

egarch models. *Journal of Business & Economic Statistics*, 24(4):470–486, 2006.

[29] Brett Szymik. Neuron Anatomy. Accessed on 08/09/2021, https://askabiologist.asu.edu/neuron-anatomy.

[30] Peter J Brockwell and Richard A Davis. *Introduction to time series and forecasting*. Springer, 2016.

[31] Nikhil Buduma and Nicholas Locascio. *Fundamentals of deep learning: Designing next-generation machine intelligence algorithms*. " O'Reilly Media, Inc.", 2017.

[32] CJC Burges. Data mining and knowledge discovery handbook: A complete guide for practitioners and researchers, chapter geometric methods for feature selection and dimensional reduction: A guided tour. *Kluwer Academic*, 1:5, 2005.

[33] Chiao-Ting Chen, An-Pin Chen, and Szu-Hao Huang. Cloning strategies from trading records using agent-based reinforcement learning algorithm. In *2018 IEEE International Conference on Agents (ICA)*, pages 34–37. IEEE, 2018.

[34] Xiangyi Chen, Sijia Liu, Ruoyu Sun, and Mingyi Hong. On the convergence of a class of adam-type algorithms for non-convex optimization. *arXiv preprint arXiv:1808.02941*, 2018.

[35] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[36] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[37] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[38] George F Corliss. Applications of differentiation arithmetic. In *Reliability in Computing*, pages 127–148. Elsevier, 1988.

[39] Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *arXiv preprint arXiv:2106.04803*, 2021.

[40] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.

[41] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[42] Yue Deng, Feng Bao, Youyong Kong, Zhiquan Ren, and Qionghai Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE*

transactions on neural networks and learning systems, 28(3):653–664, 2016.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[44] Anulekha Dhara and Joydeep Dutta. *Optimality conditions in convex optimization: a finite-dimensional view*. CRC Press, 2011.

[45] David A Dickey and Wayne A Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association*, 74(366a):427–431, 1979.

[46] Matthew Dixon. Sequence classification of the limit order book using recurrent neural networks. *Journal of computational science*, 24:277–286, 2018.

[47] Matthew Dixon. Industrial forecasting with exponentially smoothed recurrent neural networks. *Technometrics*, (just-accepted):1–27, 2021.

[48] Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine Learning in Finance*. Springer, 2020.

[49] Matthew Francis Dixon and Justin London. Financial forecasting with alpha-rnns: A time series modeling approach. *Frontiers in Applied Mathematics and Statistics*, 6:59, 2020.

[50] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[51] Harris Drucker, Chris JC Burges, Linda Kaufman, Alex Smola, Vladimir Vapnik, et al. Support vector regression machines. *Advances in neural information processing systems*, 9:155–161, 1997.

[52] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[53] Paul Embrechts, Rüdiger Frey, and Alexander McNeil. Quantitative risk management, 2011.

[54] Robert F Engle. Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica: Journal of the econometric society*, pages 987–1007, 1982.

[55] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[56] Benjamin Fehrman, Benjamin Gess, and Arnulf Jentzen. Convergence rates for the stochastic gradient descent method for non-convex objective functions. *Journal of Machine Learning Research*, 21, 2020.

[57] Thomas Fischer and Christopher Krauss. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, 270(2):654–669, 2018.

[58] Benjamin Friedlander and Boaz Porat. The modified yule-walker method of arma spectral estimation. *IEEE Transactions on Aerospace and Electronic Systems*, (2):158–173, 1984.

[59] Zhiqiang Ge. Process data analytics via probabilistic latent variable models: A tutorial review. *Industrial & Engineering Chemistry Research*, 57(38):12646–12661, 2018.

[60] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[61] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

[62] Fabian Gierens. *Bachelor's thesis: Mathematische Aspekte von Neuronalen Netzwerken vom Transformer-Typ*. PhD thesis, Department IV, Trier University, 2021.

[63] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995.

[64] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[65] Lawrence R Glosten, Ravi Jagannathan, and David E Runkle. On the relation between the expected value and the volatility of the nominal excess return on stocks. *The journal of finance*, 48(5):1779–1801, 1993.

[66] Namrata Godbole, Manja Srinivasaiah, and Steven Skiena. Large-scale sentiment analysis for news and blogs. *Icwsm*, 7(21):219–222, 2007.

[67] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[68] The Guardian. Deep blue computer beats world chess champion. Accessed on 12/08/2021 at https://www.theguardian.com/sport/2021/feb/12/deep-blue-computer-beats-kasparov-chess-1996, 1996.

[69] The Guardian. Alphago beats lee sedol in third consecutive go game. Accessed on 12/08/2021 at https://www.theguardian.com/technology/2016/mar/12/

alphago-beats-lee-sedol-in-third-consecutive-go-game, 2016.

[70] M Ugur Gudelek, S Arda Boluk, and A Murat Ozbayoglu. A deep learning based stock trading model with 2-d cnn trend detection. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

[71] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[72] Li Han, Huitian Jing, Rongchang Zhang, and Zhiyu Gao. Wind power forecast based on improved long short term memory network. *Energy*, 189:116300, 2019.

[73] Md Rakibul Haque, Salma Akter Lima, and Sadia Zaman Mishu. Performance analysis of different neural networks for sentiment analysis on imdb movie reviews. In *2019 3rd International Conference on Electrical, Computer & Telecommunication Engineering (ICECTE)*, pages 161–164. IEEE, 2019.

[74] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[76] James B Heaton, Nick G Polson, and Jan Hendrik Witte. Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1):3–12, 2017.

[77] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[78] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[79] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[80] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[81] Thomas Hollis, Antoine Viscardi, and Seung Eun Yi. A comparison of lstms and attention mechanisms for forecasting financial time series. *arXiv preprint arXiv:1812.07699*, 2018.

[82] Charles C Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International journal of forecasting*, 20(1):5–10, 2004.

[83] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[84] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[85] Ziniu Hu, Weiqing Liu, Jiang Bian, Xuanzhe Liu, and Tie-Yan Liu. Listening to chaotic whispers: A deep learning framework for news-oriented stock trend prediction. In *Proceedings of the eleventh ACM international conference on web search and data mining*, pages 261–269, 2018.

[86] Yifu Huang, Kai Huang, Yang Wang, Hao Zhang, Jihong Guan, and Shuigeng Zhou. Exploiting twitter moods to boost financial trend prediction based on deep network models. In *International Conference on Intelligent Computing*, pages 449–460. Springer, 2016.

[87] David H Hubel. Single unit activity in striate cortex of unrestrained cats. *The Journal of physiology*, 147(2):226–238, 1959.

[88] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

[89] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[90] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[91] Hitoshi Iwasaki and Ying Chen. Topic sentiment asset pricing with dnn supervised learning. *Available at SSRN 3228485*, 2018.

[92] Iberedem Iwok. Justification of wold's theorem and the unbiasedness of a stable vector autoregressive time series model forecasts. *International Journal of Statistics and Probability*, 6:1, 02 2017.

[93] Yongjin Jeong and Sangyeol Lee. Recurrent neural network-adapted nonlinear arma-garch model with application to s&p 500 index data. *Journal of the Korean Data and Information Science Society*, 30(5):1187–1195, 2019.

[94] Max E Jerrell. Automatic differentiation and interval arithmetic for estimation of disequilibrium models. *Computational Economics*, 10(3):295–316, 1997.

[95] Prajakta S Kalekar et al. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi school of information Technology*, 4329008(13):1–13, 2004.

[96] Karpathy, Andrej. The Unreasonable Effectiveness of Recurrent Neural Networks. Accessed on 19/10/2021, https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[97] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2vec: Learning a vector representation of time. *arXiv preprint arXiv:1907.05321*, 2019.

[98] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[99] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[100] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pages 972–981, 2017.

[101] Jerzy Korczak and Marcin Hemes. Deep learning for financial time series forecasting in a-trader system. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 905–912. IEEE, 2017.

[102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[103] Max Kuhn, Kjell Johnson, et al. *Applied predictive modeling*, volume 26. Springer, 2013.

[104] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[105] Sören Laue. On the equivalence of forward mode automatic differentiation and symbolic differentiation. *arXiv preprint arXiv:1904.02990*, 2019.

[106] César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2657–2661. IEEE, 2016.

[107] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[108] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[109] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyou Zhou, Wenhu Chen, Yu-Xiang Wang, and Xifeng Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *Advances in Neural Information Processing Systems*, 32:5243–5253, 2019.

[110] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony

between dropout and batch normalization by variance shift. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2682–2690, 2019.

[111] Greta M Ljung and George EP Box. On a measure of lack of fit in time series models. *Biometrika*, 65(2):297–303, 1978.

[112] Marcos López de Prado. Beyond econometrics: A roadmap towards financial machine learning. *Available at SSRN 3365282*, 2019.

[113] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[114] Alireza Makhzani and Brendan Frey. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

[115] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12, 2006.

[116] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[117] LE Melkumova and S Ya Shatskikh. Comparing ridge and lasso estimators for data analysis. *Procedia engineering*, 201:746–755, 2017.

[118] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[119] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 18(6):275–285, 2004.

[120] Victor-Emil Neagoe, Adrian-Dumitru Ciotec, and George-Sorin Cucu. Deep convolutional neural networks versus multilayer perceptron for financial prediction. In *2018 International Conference on Communications (COMM)*, pages 201–206. IEEE, 2018.

[121] Daniel B Nelson. Conditional heteroskedasticity in asset returns: A new approach. *Econometrica: Journal of the Econometric Society*, pages 347–370, 1991.

[122] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2003.

[123] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[124] Theodoros Ntakouris. Timeseries classification with a transformer model. https://keras.io/examples/timeseries/timeseries_transformer_classification/. Accessed: 2022-01-12.

[125] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[126] Genevieve B Orr and Klaus-Robert Müller. *Neural networks: tricks of the trade*. Springer, 2003.

[127] Ahmet Murat Ozbayoglu, Mehmet Ugur Gudelek, and Omer Berat Sezer. Deep learning for financial applications: A survey. *Applied Soft Computing*, 93:106384, 2020.

[128] Ioannis Panageas and Georgios Piliouras. Gradient descent only converges to minimizers: Non-isolated critical points and invariant regions. *arXiv preprint arXiv:1605.00405*, 2016.

[129] Papers With Code. Image Classification on ImageNet. Accessed on 11/10/2021, https://paperswithcode.com/sota/image-classification-on-imagenet.

[130] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

[131] Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. Deep adaptive input normalization for time series forecasting. *IEEE transactions on neural networks and learning systems*, 31(9):3760–3765, 2019.

[132] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[133] Florent Perronnin, Jorge Sánchez, and Yan Liu. Large-scale image categorization with explicit data embedding. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2297–2304. IEEE, 2010.

[134] Hieu Pham, Zihang Dai, Qizhe Xie, and Quoc V Le. Meta pseudo labels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11557–11568, 2021.

[135] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In *2014 14th international conference on frontiers in handwriting recognition*, pages 285–290. IEEE, 2014.

[136] Elad Plaut. From principal subspaces to principal components with linear autoencoders. *arXiv preprint arXiv:1804.10253*, 2018.

[137] Tomaso Poggio. Deep learning: mathematics and neuroscience. *A Sponsored Supplement to Science*, pages 9–12, 2016.

[138] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.

[139] Marc Ranzato, Christopher Poultney, Sumit Chopra, Yann LeCun, et al. Efficient learning of sparse representations with an energy-based model. *Advances in neural information processing systems*, 19:1137, 2007.

[140] Carl Edward Rasmussen and Zoubin Ghahramani. Bayesian monte carlo. *Advances in neural information processing systems*, pages 505–512, 2003.

[141] S Reddi, Manzil Zaheer, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. In *Proceeding of 32nd Conference on Neural Information Processing Systems (NIPS 2018)*, 2018.

[142] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[143] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[144] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. *Encyclopedia of database systems*, 5:532–538, 2009.

[145] Richard Restak. The secret life of the brain. 2001.

[146] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[147] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[148] Abhimanyu Roy, Jingyi Sun, Robert Mahoney, Loreto Alonzi, Stephen Adams, and Peter Beling. Deep learning detecting fraud in credit card transactions. In *2018 Systems and Information Engineering Design Symposium (SIEDS)*, pages 129–134. IEEE, 2018.

[149] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[150] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[151] Rush, Alexander and Nguyen, Vincent and Klein, Guillaume. The Annotated Transformer. Accessed on 21/10/2021, https://nlp.seas.harvard.edu/2018/04/03/attention.html.

[152] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recogni-

tion. *arXiv preprint arXiv:1402.1128*, 2014.

[153] Yosiyuki Sakamoto, Makio Ishiguro, and Genshiro Kitagawa. Akaike information criterion statistics. *Dordrecht, The Netherlands: D. Reidel*, 81(10.5555):26853, 1986.

[154] Jan Schmitz. Stock predictions with state-of-the-art transformer and time embeddings. towardsdatascience.com. Accessed: 2022-01-10.

[155] René Schneider. *Maschineller Erwerb lexikalischen Wissens aus kleinen und verrauschten Textkorpora*. Herbert Utz Verlag, 1999.

[156] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[157] Andrew Senior, Georg Heigold, Marc'aurelio Ranzato, and Ke Yang. An empirical study of learning rates in deep neural networks for speech recognition. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6724–6728. IEEE, 2013.

[158] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. *Applied Soft Computing*, 90:106181, 2020.

[159] Omer Berat Sezer and Ahmet Murat Ozbayoglu. Algorithmic financial trading with deep convolutional neural networks: Time series to image conversion approach. *Applied Soft Computing*, 70:525–538, 2018.

[160] Bin Shi and Sundararaja S Iyengar. *Mathematical Theories of Machine Learning-Theory and Applications*. Springer, 2020.

[161] Sima Siami-Namini and Akbar Siami Namin. Forecasting economics and financial time series: Arima vs. lstm. *arXiv preprint arXiv:1803.06386*, 2018.

[162] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[163] Justin Sirignano and Rama Cont. Universal features of price formation in financial markets: perspectives from deep learning. *Quantitative Finance*, 19(9):1449–1459, 2019.

[164] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[165] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[166] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*,

25, 2012.

[167] Bruno Spilak. Deep neural networks for cryptocurrencies price prediction. Master's thesis, Humboldt-Universität zu Berlin, 2018.

[168] Sashank Sridhar and Sowmya Sanagavarapu. Multi-head self-attention transformer for dogecoin price prediction. In *2021 14th International Conference on Human System Interaction (HSI)*, pages 1–6. IEEE, 2021.

[169] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[170] Gilbert W Stewart. On the early history of the singular value decomposition. *SIAM review*, 35(4):551–566, 1993.

[171] Josef Stoer and Roland Bulirsch. *Introduction to numerical analysis*, volume 12. Springer Science & Business Media, 2013.

[172] Gregory O Stone. An analysis of the delta rule and the learning of statistical associations. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1:444–459, 1986.

[173] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2828–2837, 2019.

[174] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[175] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[176] Shuntaro Takahashi, Yu Chen, and Kumiko Tanaka-Ishii. Modeling financial time-series with generative adversarial networks. *Physica A: Statistical Mechanics and its Applications*, 527:121261, 2019.

[177] Koji Takeuchi. Distribution of informational statistics and a criterion of model fitting. suri-kagaku (mathematical sciences) 153 12-18, 1976.

[178] Qi Tang, Ruchen Shi, Tongmei Fan, Yidan Ma, and Jingyan Huang. Prediction of financial time series based on lstm using wavelet transform and singular spectrum analysis. *Mathematical Problems in Engineering*, 2021, 2021.

[179] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for*

machine learning, 4(2):26–31, 2012.

[180] Ruey S Tsay. *Analysis of financial time series*, volume 543. John wiley & sons, 2005.

[181] Alan M Turing. Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer, 2009.

[182] Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. Dimensionality reduction: a comparative. *J Mach Learn Res*, 10(66-71):13, 2009.

[183] F. Van Veen and S Leijnen. The neural network zoo. Accessed on 12/08/2021 at https://www.asimovinstitute.org/neural-network-zoo, 2019.

[184] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[185] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[186] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. Ieee, 2001.

[187] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.

[188] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[189] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[190] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057. PMLR, 2015.

[191] George Zerveas, Srideepika Jayaraman, Dhaval Patel, Anuradha Bhamidipaty, and Carsten Eickhoff. A transformer-based framework for multivariate time series representation learning. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2114–2124, 2021.

[192] Cherry Y Zhang and Ben Jacobsen. Are monthly seasonals real? a three century perspective. *Review of Finance*, 17(5):1743–1785, 2013.

[193] Hongyi Zhang, Yann N Dauphin, and Tengyu Ma. Fixup initialization: Residual learning without normalization. *arXiv preprint arXiv:1901.09321*, 2019.

[194] Jian Zheng, Cencen Xu, Ziang Zhang, and Xiaohua Li. Electric load forecasting in smart grids using long-short-term-memory based recurrent neural network. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2017.

[195] Fangyu Zou, Li Shen, Zequn Jie, Weizhong Zhang, and Wei Liu. A sufficient condition for convergences of adam and rmsprop. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11127–11135, 2019.