Imperial College London

Department of Mathematics

Intraday Trading with Neural Networks and Deep Reinforcement Learning

Author: Hendrik Zimmermann (CID: 01950260)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2020-2021

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Signed Hendrik Zimmermann — 07.09.2021

Acknowledgements

I would like to express my gratitude to my supervisor Paul Bilokon for his constant support and insightful discussions throughout the writing of this thesis. Thanks also goes to David Finkelstein from DQR for his inspiration on the LSTM model. Additionally, I would like to thank all other lecturers and staff members of the MSc program to provide us with a great teaching experience despite the difficult circumstances.

Furthermore, I want to acknowledge the support received from my mother, grandmother and my brother, who made it possible for me to pursue this master degree in the first place and always provided me with an open ear.

Abstract

The search for trading strategies can be a highly time-consuming and quite difficult job. We show that using feed-forward neural networks and autoencoders the process can already be partly automated. By investigating the characteristics of mean-reverting trading signals, we design loss functions based on simple mathematical metrics such as the mean and the variance to train the networks to find trading signals. The results show a promising PnL for some configurations of the models. Application of neural networks optimising such special loss functions has already proven profitable in trading [1, 2] and we can confirm this for our simple loss function on mean-reverting signals.
We furthermore investigate the optimal trading behaviour on mean-reverting signals. Traditionally, mean-reverting signals are traded using Bollinger Bands which produce trading decisions by setting thresholds for the signal based on its standard deviation. We show that the application of reinforcement learning to find trading decisions can optimise trading and increase profit compared to the trading behaviour generated by Bollinger Bands. We will focus on the subgroup of Policy-based methods, especially the REINFORCE, the Advantage Actor-Critic and the Proximal Policy Optimisation algorithm. We put their performance into context by comparing them to one further machine learning trading model based on recurrent neural networks. All presented models outperform Bollinger Bands in terms of trading profits on simulated as well as historical data.

Key words: Mean-reversion, Artificial Neural Networks, REINFORCE, Advantage Actor-Critic, Proximal Policy Optimization

Contents

List of Figures

List of Tables

Introduction

"Buy low, sell high" — easier said than done. This can probably be confirmed by the many researchers in financial institutions whose job is to find profitable and stable trading strategies. A trading strategy is usually characterised by some indicator helping the trader decide when to best buy and sell certain assets. One of the first of these indicators was brought to wider attention by Gerald Bamberger during the 1980s and only focuses on two assets. The so-called pairs trading strategy also laid the foundation of statistical arbitrage and is still used and improved today [7, 8, 9]. The strategy suggests that by finding two assets that seem to react similarly in general market scenarios, one could make a profit by trading events at which the two assets seem to behave differently. The most commonly used metric to identify such pairs is the correlation coefficient. By tracking the residual of a linear regression between the two correlated asset prices, one obtains a process which is *mean-reverting* (varying around 0 in this case) and based on which trading decisions can be formed. Buying the asset which according to the mean-reverting process is underpriced, selling the asset which is supposed to be overpriced and reversing the trades when the signal swings back leads to a profit independent of market beta. We call such a strategy *market-neutral*. These steps describe the process of trading a mean-reverting signal. Over the past years, inspired by Bamberger's work, many more papers about the generation and extraction of mean-reverting trading signals have been published. Among these, the 2010 paper *Statistical arbitrage in the US equities market* by Avellaneda and Lee [10] is one of the most influential ones when it comes to traditional mean-reverting strategies. It extends the idea of pairs trading to larger baskets of assets and uses Principal Component Analysis to extract a mean-reverting signal. As an alternative to traditional approaches, Sarmento et al. [11] and Chang et al. [12] show that machine learning can also be used successfully in the process of generating trading signals.

One of the most versatile applicable machine learning models are *artificial neural networks* (ANNs) because of their flexible structure. This also makes them interesting for financial application. Neural networks can be separated into several subgroups based on their architecture and used layers. Regarding the approaches to extract trading signals from prices we focus on so-called *feed-forward neural networks* (FNN), which form the simplest group of ANNs. The application of neural networks in combination with a customised loss function offers numerous opportunities. By choosing simple metrics, we design loss functions that aim to reward characteristics of mean-reverting trading signals as the output of neural networks. The signals are tested for profitability using Bollinger Bands. Moreover, the usefulness of *autoencoders* as a tool for catching broader market movements is investigated. Autoencoders are neural networks with a special architecture which in the first place are used as a dimension reduction technique.

As a second step, we make an attempt to improve the trading decisions provided by Bollinger Bands with the use of *reinforcement learning* (RL). The success of reinforcement learning methods has been proven in a wide range of problems such as board games like Go [13], complex computer games like Dota 2 [14] and also in the DNA protein folding problem (one of the longest standing problems in bioinformatics) [15]. Driven by past success in other areas, reinforcement learning is nowadays an active part of research in the financial industry. In some isolated cases, reinforcement learning algorithms such as *Tabular Methods* and *Deep Q Networks* (DQN) have already been applied to the problem of pairs trading and to general mean-reverting trading strategies [16, 17]. This work further investigates the potential of policy-based RL algorithms, especially Actor-Critic methods, a subclass of RL algorithms, which use two Deep Neural Networks to find an optimal policy in the environment. We will focus our analysis on three algorithms with increasing complexity starting with the *REINFORCE* algorithm first introduced by Williams [18]. We furthermore apply the *Advantage Actor-Critic* (A2C) and *Proximal Policy Optimisation* (PPO), to the problem of finding trading decisions based on a mean-reverting signal. As mentioned before, both algorithms

use two neural networks with different tasks to solve the problem, with the PPO's learning algorithm approaching problems in a slightly more complex way compared to the A2C model. The number of networks used and what exactly is modelled by these networks distinguishes them from Tabular Methods and DQNs. A2C and PPO both have proven useful in solving problems related to investment optimization and trading [19, 20]. Hence, we expect them to also have some potential when it comes to trading a mean-reverting signal. To create a benchmark by another more traditional machine learning model, we will moreover train a LSTM network on a mean-reverting process and compare the results to those obtained by the reinforcement learning algorithms.
Chapter 1 introduces the basic concepts, which serve as a foundation for the remaining part of this work. Among other things, we provide our approach for modelling mean-reverting processes and give an overview of the primary principles of statistical arbitrage and trading signals. This is followed by a brief explanation of the use of Bollinger Bands on mean-reverting signals. Moreover, we will introduce the high-frequency data, used to test the introduced models.
The subsequent chapter provides an overview of the machine learning models that have been applied in this thesis. Leading with the explanations of simple neural networks and their general setup, we will continue the neural network section with outlining the deeper workings of LSTM networks. The chapter concludes with the introduction of the essential framework for reinforcement learning and the learning environment.
The third chapter contains a short introduction of three methods used to generate mean-reverting signals from asset prices. All methods are tested on the data explained in chapter 1 and we will use the first of these methods for a robust test of the trading agents introduced in chapter 4.
These agents will be a LSTM neural network and the reinforcement learning models mentioned earlier: REINFORCE, Advantage Actor-Critic and Proximal Policy Optimisation. Chapter 4 outlines the exact configurations of these models and shows the results of their training in a simulated trading environment based on an Ornstein-Uhlenbeck (OU) process. We analyse the results and conclude with give a brief comparison of the models.
The last chapter contains the results, of the more robust, historical data test of the reinforcement learning models. We will use the PCA signal generated in chapter 3 in the trading environment and analyse the trading results achieved by the agents. Moreover, we shows that for two agents a pre-fitting process on a simulated signal improves the performance.

Chapter 1

Statistical Arbitrage in an Intraday Environment

This section introduces the main concepts of statistical arbitrage and trading signals. We will put particular focus on mean-reverting signals and analyse their properties deeper. Finishing the chapter with an introduction of Bollinger Bands which will serve as benchmark trading behaviour.

1.1 Statistical Arbitrage and Trading Signals

Statistical arbitrage or StatArb describes a group of trading strategies that heavily rely on statistical methods and algorithmic modelling to extract trading signals from market data. A trading signal can be regarded as a value assigned to a single asset, security or basket of either, based on which trading decisions are formed. A common approach (when the signal represents a price of a basket of assets) is to hold a long position in the basket if the signal is low and a short position if it is high, changing positions accordingly over time. The data, used to find such signals and therefore the underlying strategy, can range from simple price series to large sets of fundamental data about the underlying assets. This also holds for the number of assets being included in one strategy. While the beginnings of statistical arbitrage were made by combining only two assets, nowadays baskets with several assets are traded in one single strategy [10].
As already mentioned, the first statistical arbitrage strategies were pairs trading strategies combining only two assets. These strategies were mainly introduced by Gerry Bamberger[1] in the 1980s who was working for Morgan Stanley at this time. The simple idea behind this group of strategies is finding two securities that have a causal and statistical relation such that a linear combination of the prices results in a mean-reverting time series. By now switching between long and short positions in the basket when this time series is low respectively, high, one can extract a profit. Thanks to the properties of mean-reversion, these strategies are also beta-neutral and hence independent of the drift the asset prices might carry. To identify such pairs via quantitative analysis, metrics like distance measures, correlation and cointegration are used [21]. Common examples for possible pairs are Goldman Sachs–JP Morgan Chase, Volkswagen AG–General Motors and Coca Cola–Pepsi. It makes sense to first look for pairs which also have a causal connection such as a common sector, as in the previously mentioned examples. One further milestone in the development of mean-reverting strategies was set in 2010 by Avellaneda and Lee with the publication of the paper *Statistical arbitrage in the US equities market* [10]. Avellaneda and Lee used the approach of factor modelling to replicate single asset prices using a set of Principal Components or sector exchange-traded funds (ETFs). In both cases, a trading signal is extracted by measuring the deviation of the asset time series from the basket time series. Provided the basket serves as a good approximation, this again results in a mean-reverting signal which can be traded profitably by buying/selling the asset/basket if the signal crosses predefined thresholds. The results obtained by this approach, motivated further analysis and the use of more complex methods, resulting in a universe of publications dealing with statistical arbitrage and mean-reverting trading signals using

[1]Appears to be common knowledge since no citation source could be found.

methods ranging from statistical metrics to deep reinforcement learning.
Although models became more and more sophisticated in recent years major events like the financial crisis in 2007/08 can still lead to significant losses for statistical arbitrage strategies. This is mainly due to the possible decorrelation between previously profitable trading pairs and baskets but could also be amplified through trading barriers and liquidity shortages during such events. The latter is also the reason why most strategies that incorporate mean-reversion are only applied in sufficiently liquid markets. With higher trading volume, the guarantee of completely executing the desired basket is increased and market risk can be reduced.

1.2 Mean-Reverting Signals

Although there are other statistical arbitrage strategies, methods to extract mean-reverting signals are widely used and also tend to prove profitable over long periods [10, 22]. Hence we as well are going to focus our analysis and implementations on mean-reverting trading strategies. Simply described, a mean-reverting process is a time series that tends to always return to a specific long-term mean. We will furthermore assume that this mean will be constant over time. Mean-reverting processes are best modelled using Ornstein-Uhlenbeck processes which we define through the following stochastic differential equation:

$$dx_t = \theta(m - x_t)\,dt + \sigma\,dW_t \tag{1.2.1}$$

on an interval $\mathcal{T} = [0, T]$ where $(x_t)_{t\in\mathcal{T}}$ is the mean-reverting/OU process, m its long term mean, θ the speed of reversion to the previously mentioned mean, σ the instantaneous volatility and W_t a standard Brownian motion. Using the ansatz $f(t, x_t) = x_t e^{\theta t}$ we can find a closed form solution to the SDE (1.2.1) which is defined as:

$$x_t = x_0 e^{-\theta t} + m(1 - e^{-\theta t}) + \sigma \int_0^t e^{-\theta(t-s)} dW_s \tag{1.2.2}$$

with $x_0 \in \mathbb{R}$ being the start point of the process. We will later use this solution to estimate OU–process–parameters of a mean-reverting signal generated from historical market data. Our main focus will be on improving the existing trading methods on such mean-reverting signals using machine learning, especially reinforcement learning. Despite this, we will start the analysis by investigating the potential of machine learning on the extraction of trading signals. The decision to use machine learning algorithms is rooted in the statistical properties of OU processes. Especially the fact that OU processes are stationary makes it quite attractive to leverage the power of machine learning algorithms. To run tests and simulations on the mean-reverting processes with any algorithm we will model a discrete version of the process using the procedure of algorithm 1.

Algorithm 1: Sampling discrete OU Processes

```
1: Set $T$ and $N$.
2: Calculate $\delta = T/N$.
3: Set process parameters: $\theta$, $m$ and $\sigma$.
4: Set $x_0 = m$.
5: Sample $Z_t$ for $t = 1, ..., N$ where $Z_t$ are independent and identically distributed random
   variables (iid) with $Z_1 \sim \mathcal{N}(0, \delta)$
6: for t = 1, ..., N do
7:     Calculate $x_t = x_{t-1} + \theta(m - x_{t-1})\delta + \sigma Z_t$
8: end for
9: return $(x_t)_{1,...,N}$
```

In the first line of the algorithm the time horizon and the granularity of the sampled process are set. In general, given T stays constant, the higher N is the more detailed and closer to the continuous version will the sample path be. Line 2 calculates the step size δ used to later determine the non-random change in the process. The following line sets the parameters characterising the OU Process, while in line 5, N iid normal random variables are sampled. The for-loop from line 6

to line 8 then produces the sample path of the process based on the parameters and the sampled random variables.
In the usual case, the signal represents a linear combination of several assets, and hence it is possible to directly buy and sell the signal (by trading the assets according to the weights of the linear combination). The logical consequence is that a trader seeking profit wants to buy such a signal low and sell it again higher, with the goal of taking as many trading opportunities as possible. To be able to compare several algorithms based on their trading performance on such signals we introduce the following base setup that will be used to generate the OU process for the simulation part in chapter 4.

OU Base Setup: $\theta = 10$, $m = 0$, $\sigma = 0.2$

A simple way to model trading behaviour on mean-reverting processes is using *Bollinger Bands* (BBs). This approach is also used by Avellaneda and Lee [10] in a slight variation. In the following section we will give a brief introduction of how BBs are used.

1.3 Bollinger Bands

Bollinger Bands, first introduced by John Bollinger in the 1980s [23], are a simple method to find trading decisions on a mean-reverting signal. They are still used today and will form the benchmark for other signal trading strategies. Given a signal representing a basket, the method encapsulates the idea of simply buying low and selling high using statistical metrics such as mean and standard deviation. Assuming we have a discrete trading signal, we obtain the buy and sell actions from Bollinger Bands as follows:

1. For each point in time calculate a moving average (MA) with look-back window of length N time points[2].

2. Calculate the standard deviation σ of the values over the same period as the moving average.

3. Obtain the upper and lower Bollinger Band as $MA + \sigma K$ and $MA - \sigma K$, respectively. K is the distance factor of the standard deviation usually set at 1.5 or 2.

4. Buy and sell events are characterised by the MA crossing the lower BB from above and the upper BB from below, respectively.

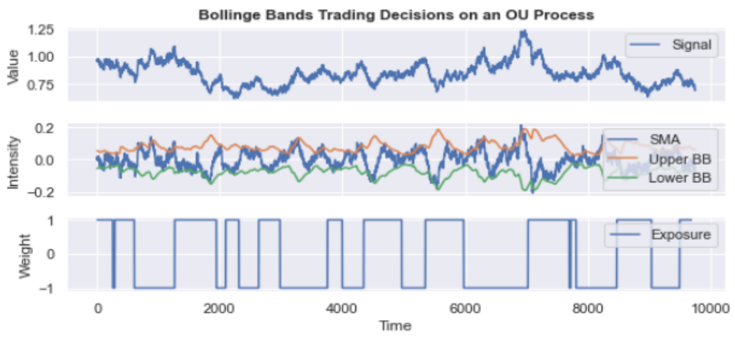

Figure 1.1: Top: Plot showing a possible trading signal. Middle: Simple Moving Average (SMA) over the last 250 time points of the signal and upper as well as lower Bollinger Bands also calculated over the same time horizon. The Bollinger Bands have a distance factor K of 1.5. Bottom: Plot showing a possible exposure time series if one would buy and sell two units of the basket according to the Bollinger Bands.

[2]Using the approach from the previous section this covers the time interval $[t - N\delta, t]$ at time t

For simplicity, most of the time only 2 units of the basket are sold or bought at the respective action events such that a trader would always have an exposure of either 1 or -1. Figure 1.1 shows exactly such a trading behaviour on a simulated OU process.
As one can see, in the middle plot of figure 1.1 the Bollinger Bands neatly picks up the idea of buying low and selling high. This results in the exposure pattern shown by the lower plot. The top plot of this figure is the extracted signal which here is just simulated for visualisation purposes. The trading actions and the respective trading performance will serve as a benchmark for the later introduced trading algorithms. We will vary the window size of the moving average and the distance factor to obtain an optimal PnL based on the underlying data. We will calculate the final PnL on a mark-to-market basis. This means the PnL process shown in the various figures in the work is the cumulative sum of all returns, made when swapping the position plus the current value of the position. we will close all positions at the end of the trading period.

1.4 Exploratory Data Analysis

This section serves as a short introduction to the asset data that is used to test the provided algorithms. We will focus on a high-frequency dataset consisting of four securities with the tickers BBG000XG00K4, BBG000XG00Y9, BBG00BFJSPJ1 and BBG00BFJSQ90. The first two tickers correspond to two Henry Hub Natural Gas futures, while the last two represent Light Sweet Crude Oil futures. The data covers the period from 01.02.2021 to 12.02.2021, and the four chosen futures are among the most liquid ones of the Oil and Gas futures during this time. We resample the data to a frequency of 0.5 seconds giving us approximately 1.7×10^6 price observations during this two-week period. It is essential to mention here that for all statistical and machine learning models the first 70% of the data are used as the training batch and that we use the whole dataset to test the models. This results in the cut-off point for in-sample to out-of sample at the 10.02.2021 00:00:00. Figure 1.2 shows the four security prices during the two-week period.

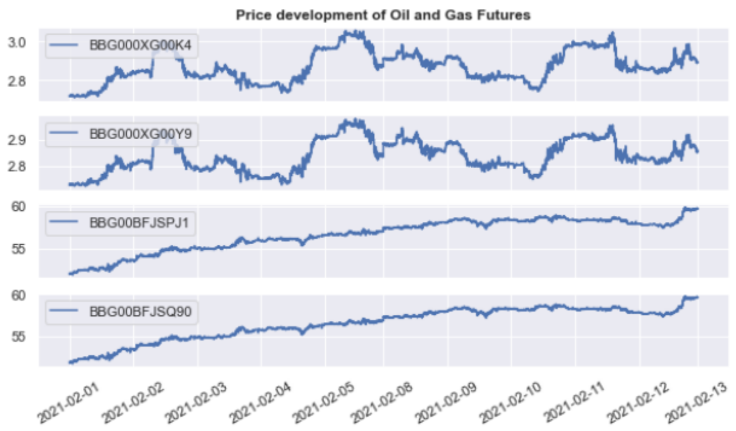

Figure 1.2: The four plots of this graphic show the prices of the oil and gas futures with tickers BBG000XG00K4, BBG000XG00Y9, BBG00BFJSPJ1 and BBG00BFJSQ90 over the two-week period between 01.02.2021 and 12.02.2021. Prices are indicated in USD.

Just by analysing the plots of the prices, one can already appraise that the correlation between each future pairs with the same underlying should be relatively high and indeed in both cases the correlation is above 0.95. The correlation between the futures of different underlyings lies around 0.4 to 0.45. Moreover we observe that for both Oil futures the change is almost constantly positive while for the Gas futures the price seems to vary strongly between 2.7 and 3.0 USD.

Chapter 2

Theoretical Overview

This chapter provides an overview of the main machine learning models used in this thesis. We will begin with a detailed explanation of neural networks, which play an essential role in many of the used algorithms. The remainder of the is chapter is dedicated to the reinforcement learning models used for the trading decision optimisation.

2.1 Artificial Neural Networks

2.1.1 General Structure

Different architectures of neural networks will play a vital part in the remainder of this thesis. First introduced by McCulloch and Pitts in 1943 [24], artificial neural networks have evolved to one of the best-known machine learning models. Their broad applicability and good approximation quality make them also quite attractive to use in the financial markets. Hence, offering many more new possibilities, neural networks are nowadays used in almost every company in the financial sector, with their applications ranging from price prediction to pattern analysis.

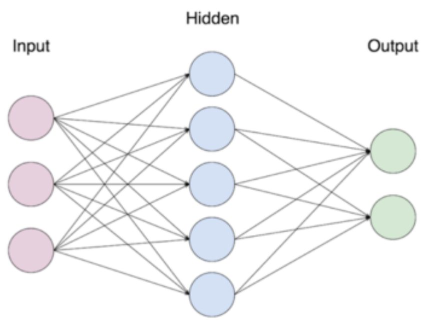

Figure 2.1: Simple feed-forward neural network architecture with an input layer of dimension 3 and two hidden layers (last one output layer) of dimensions 5 and 2. The network is fully connected. Source: [3]

We will begin by introducing the main concepts and notations of *feed-forward neural networks* (FNNs). Most neural networks (including the ones we will use) are structured by and comprised of several layers as shown in figure 2.1. Starting with the input layer, a network has $r - 1$ hidden layers, including the final output layer. r therefore is the total number of layers including the input layer. A layer consists of a number of neurons or *nodes*, which form the smallest computation unit of a neural network.

Formally, an ANN can be written as a function f which maps from the input space $\mathbb{R}^I$ to the output space $\mathbb{R}^O$:

$$f : \mathbb{R}^I \to \mathbb{R}^O,\ x \mapsto (\sigma_k \circ L_k \circ ... \circ \sigma_1 \circ L_1)(x)$$

where we will refer to the number of units or nodes in the i-th hidden layer as d_i with $i = 1, ..., r-1$ and I and O being the size of the input and output layer, respectively (d_0, d_r). The functions σ_i with $i = 1, ..., r-1$ represent the *activation functions* which we will assume to be the same for each node within one layer. We can therefore model these functions as $\sigma_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$. We will later introduce the activation functions, that we use in our models. The layer functions L_i are the building blocks of the actual neural network and can be written as affine functions:

$$L_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i},\ x \mapsto W^i x + b^i$$

where $W^i \in \mathbb{R}^{d_i \times d_{i-1}}$ is the weight matrix of the i-th layer and $b^i \in \mathbb{R}^{d_i}$ the vector of biases. The weight matrices and bias vectors are referred to as the parameters of the neural network, while the number of layers r and the dimensions of each layer d_i, $i = 0, ..., r$ are known as the *hyper-parameters* of the network. The hyperparameters are usually set before the training of the model and will not be changed during the process. The learning of the network only happens through adjustments in the weight matrices and bias vectors. Figure 2.1 shows the architecture of a feed-forward neural network. The network is *fully connected* meaning a connection exists between ever node in consecutive layers. Mathematically, this means that every element in the weight matrix W^i is non-zero for all $i \in [r-1]$. We call a layer *dense* if it is fully connected to the previous layer. Before describing the training process further, we will introduce the used activation functions.

2.1.2 Activation Functions

To keep a neural network fully linear, we can use the identity $g(x) = x$ as our activation function in all layers. Moreover, it has features wished for in a useful activation function such as continuity and a derivative on its whole domain. We will use the identity in cases where we want to come up with a suitable linear combination of parameters. Despite its simplicity, one of the best known and widely used activations is the *Rectified Linear Unit* (ReLU) function which is defined as:

$$g(x) = \max\{0, x\}, \quad g'(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases} \tag{2.1.1}$$

It is common practice to assume $g'(0) = 0$ since the derivative of the ReLU function is undefined for $x = 0$. The ReLU activation will be the main function used in the neural networks that are part of the reinforcement learning algorithms. Using this function, one also needs to be aware of the drawbacks the simplicity has. In the case of the ReLU function, a problem know as the *dead ReLU problem* can arise where the function only receives negative inputs from the attached layer function. This is turn results in a constant all zero output, which can jeopardize the learning process when using gradient-based learning. Solutions to this problem have been found in the form of the *Leaky ReLU* and the *Parametric ReLu* which allow negative values to get through the layer. An additional one-dimensional activation function we are going to introduce is the *Hyperbolic Tangent* (tanh) defined as

$$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad g'(x) = 1 - g(x)^2 \tag{2.1.2}$$

One main difference of this function, when compared to the previous ones, is that it scales its output to an open interval from -1 to 1. Hence, one can use this function to map the output value to the desired range and avoid large values in the hidden layers. We will use this function mainly in the later introduced LSTM trading agent. The last one-dimensional activation function used in the workings of this thesis is the *Sigmoid* activation function which is defined as

$$g(x) = \frac{1}{1 + e^{-x}}, \quad g'(x) = g(x)(1 - g(x)) \tag{2.1.3}$$

and scales its input in the interval $(0, 1)$.

All activation functions introduced so far are applied piece-wise to each element of the output of the layer. Especially when working with probabilities, a function to scale the output values into $[0, 1]$ and force them to sum to 1 is required. To achieve this, we will use the *Softmax* function, which is defined as follows:

$$g_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}}, \quad \frac{\partial g_i(x)}{\partial x_j} = \begin{cases} g_i(x)(1 - g_i(x)), & i = j \\ -g_i(x)g_j(x), & i \neq j \end{cases} \tag{2.1.4}$$

This function uses all the input parameters to calculate the value for one output and hence is multi-dimensional in the input compared to the previously introduced functions, which are only one-dimensional. The Softmax function is a common choice when modelling discrete distributions in reinforcement learning and we as well will use the function to model action probabilities in the reinforcement learning section.

2.1.3 Learning

For the classical supervised learning problems the weights and biases of the network are update using the loss calculated on the model output and the provided target. This loss, for example the mean squared error between output and target, usually serves as input to the stochastic gradient decent procedure, which calculates the adjustments of the parameters. These adjustments represent changes in the parameters which result in the largest reduction of the loss function given the current observation and target pair. The extend to which these adjustments are applied to the parameters is controlled by the learning rate α, which belongs to the group of hyperparameter for neural networks. An efficient way to realise this procedure is using the Adam optimiser, which was introduced in 2015 by Kingma and Ba [25]. This optimiser represents an extension of the standard stochastic gradient descent algorithm and will be used by us in all neural networks.
A problem one faces in reinforcement learning is that no label is provided for the optimisation of the neural networks used in the models. Therefore, the loss will be calculated by approximating the desired target value of one observation. The calculation of these losses will be described in more detail when we introduce the used algorithms.

2.1.4 Long-Short Term Memory Neural Networks

A particular group of ANNs is formed by recurrent neural networks (RNN). RNNs allow leveraging of the time dependency among sequential observations. As shown by McKinlay and Lo [26] time dependency exists for time series such as asset returns and prices. While FNNs ignore possible relationships between consecutive observations by assuming the samples are independent and their order in time is random, RNNs attempt to use previous input values when calculating the subsequent output value and hence make use of the ordering of the data. This advantage also requires adjusting the input data where each input feature in an observation will now be a time series. In the case of figure 2.2 this series is t+1 steps long.

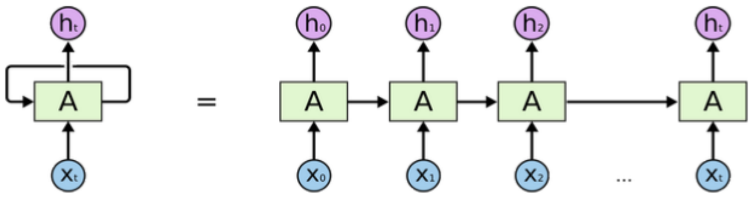

Figure 2.2: RNN cell unfolding for a time series of length $t + 1$. Source: [4]

Figure 2.2 describes the inner architecture of a single RNN cell and intuitively explains the definition of the hidden state function as

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1}) \tag{2.1.5}$$

where h_t is the hidden state at time t, x_t is the input value at time t and h_{t-1} is the hidden state of the previous layer at time $t-1$. W_{ih}, W_{hh}, b_{ih} and b_{hh} are the weight matrices and the bias vectors for each layer part, respectively. This calculation is performed for each input in every layer of the RNN.
A further improvement, tackling some of the problems arising when fitting RNNs, is provided by so-called *long-short term memory* (LSTM) neural networks. The LSTM cell structure is characterised by different gates which regulate the information flow in the node. This allows the networks to handle long term dependencies with more diligence and does not require long chains of matrix multiplications to do so. Figure 2.3 gives a detailed overview of the mechanics and information flow incorporated in a LSTM cell/node.

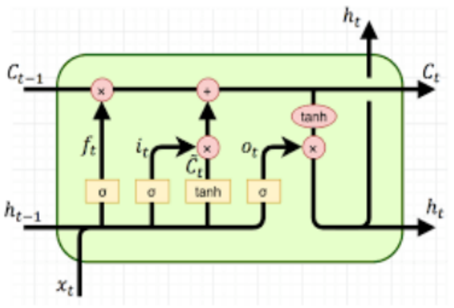

Figure 2.3: LSTM cell information flow and calculation diagram. Source: [5]

As before h_t and h_{t-1} are the hidden states at time t of the current and at time $t-1$ of the previous layer, c_t and c_{t-1} are the cell states of before mentioned layers. The different memory gates of the LSTM network are represented by f_t, i_t, $\tilde{c}_t$ and o_t and we will refer to them as input, forget, cell and output gates in the given order. The calculations performed in one cell can be summarised with the following sequence of equations:

$$\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
\tilde{c}_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

where W_{jk} and b_{jk} with $j \in \{i,h\}$ and $k \in \{f,g,i,o\}$ again represent the weight matrices and bias vectors in the cell. The first four equations are used to calculate the different gate values, which subsequently are used to obtain the final cell state and the hidden state at time t. The $\odot$ symbol represents the Hadamard product [1]. As shown in figure 2.3 both serve as input to the next hidden layer for time $t+1$. Depending on the output design, h_t might also already be used as part of the final output. As mentioned before, the characteristics of RNNs in general and LSTMs, in particular, provide a good foundation for an application in time series analysis and trading. Therefore, we will later design a trading agent based on a LSTM neural network that we will train to make profitable trades based on a mean-reverting signal. To produce such mean-reverting signals a special structure of neural networks, namely *autoencoders* is of particular interest.

2.1.5 Autoencoders

Autoencoders are dimension reduction algorithms in the form of neural networks and use layers of smaller dimension than the input layer dimension to condense the data to a chosen number of features. We will only focus on feed-forward autoencoders although it is also possible to use LSTM

[1]see equation (A.0.1)

and *convolutional* layers in autoencoders. A structure typical for an autoencoder is shown in figure 2.4. Such models usually consist of two parts, with the first part being the *encoder* and the second part the *decoder*. The encoder is build by the first layer up to (and including [2]) the layer with the smallest dimension in the whole autoencoder, with layers in between having a monotonously decreasing number of nodes in the direction of the data flow. The decoder entails the remaining layers of the network this time with monotonously increasing number of neurons in each layer going forward until the last layer has the same dimension as the input vector. The two parts pasted together constitute the final autoencoder model. In figure 2.4 an autoencoder with input dimension 10 and encoding dimension 3 is shown. It, moreover, illustrates a common choice for the decoder layers, which is just mirroring the the encoding layers without the smallest layer. Hence we obtain a symmetrical network with equal input and output dimensions.
Autoencoders are then trained in a special way: Since we want the model find the best low dimensional representation of the input the target will be equal to the input data. Consequently, to find the loss, we then compare the raw input with the input pushed through the network. Intuitively, the task of the network is then to find the best representation of the data in the smallest layer to best recover the original input values.

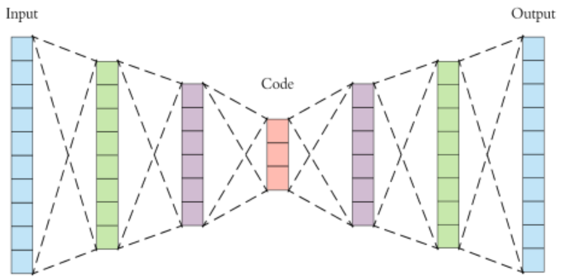

Figure 2.4: Standard architecture of an autoencoder with seven layers. The encoder layers are of size 10, 8, 6 with 3 being the encoding dimension. These layers are followed by the decoder which mirrors the layer dimensions of the encoder. Source: [6]

2.2 Deep Reinforcement Learning

We will start with a short introduction of the reinforcement learning setting, which is used for the agents. Reinforcement learning is based on the simple idea that an actor, also referred to as an agent, chooses an action A from a given set to act in an environment in a given state S. This interaction results in a change of the environment and a reward R is given as feedback from the environment to the agent. The agent then uses this reward to determine how good or bad the action was concerning a certain goal that the agent aims to achieve. This assessment helps the agent to learn which action to take at which state of the environment. The learning should then lead to achieving the predefined goal. The overall interaction process between agent and environment leads to the following sequence of states, actions and rewards: S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, ... where it is conventional to assign the reward received by action A_t to time $t+1$. We will also refer to this sequence as the *trajectory*. This overall setup can be formalized with the concept of *Markov Decision Processes* (MDPs).

[2]Subject to individual preference. Sometimes the smallest layer is not included in the encoder.

2.2.1 Markov Decision Process

We will regard a MDP as a 4-tuple of three sets and one function:

- **State space** $\mathcal{S} \subset \mathbb{R}^s$. The state space contains all the possible configurations of the environments in which the agent acts (numerically encoded as vectors or matrices). This is, for example, the current price of the asset, the current exposure and the current cash in the account for a trading problem. (s is the dimension of the state space)
- **Action space** $\mathcal{A} \subset \mathbb{R}^a$. The action space includes all actions which are available at a certain state. For the problems we will attempt to solve, the action space will be the same for all states. In the case of a trading problem, actions can be buy, sell or hold a given asset. (a is the dimension of the action space, only not equal to one if several actions can be taken at one point in time)
- **Rewards** $\mathcal{R} \subset \mathbb{R}$. The reward set contains all possible rewards that the agent can obtain by interaction with the environment. In a trading environment this could be set of all possible returns of an investment strategy. For completeness, we mention that sometimes the reward is also modelled as a function $r(s, a, s')$ mapping into the reward space with $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$.
- **Dynamics function** p. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is defined as the conditional probability of transitioning to state s' and getting reward r when choosing action a at state s:

$$p(s', r|s, a) = \mathbb{P}(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a) \tag{2.2.1}$$

  where $s', s \in \mathcal{S}$, $r \in \mathcal{R}$ and $a \in \mathcal{A}$.

For $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$ to be a MDP one more condition is required. For the decision process to be *Markov* we need to guarantee that the dynamics function fully characterises the possible scenarios in the environment. This can be broken down to the states of the environment by making the *Markov property* essential for the states. This means that every state must include all information about the current environment and no further information would change the future scenarios. This can be formalized as:

$$\begin{aligned}&\mathbb{P}[S_{t+1} = s_{t+1}, R_{t+1} = r \mid S_t = s_t, A_t = a_t] = \\ &\mathbb{P}[S_{t+1} = s_{t+1}, R_{t+1} = r \mid S_t = s_t, A_t = a_t, ..., S_0 = s_0, A_0 = a_0] \quad \text{for all t} \in \mathbb{N}\end{aligned}$$

where $s_{t+1}, ..., s_0 \in \mathcal{S}$, $a_t, ..., a_0 \in \mathcal{A}$ and $r \in \mathcal{R}$. This completes the properties of the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$ which are required to make it a Markov Decision Process.
In addition, a MDP can be divided into *episodes* which are typically ending with a terminal state. A terminal state is every state s $\in \mathcal{S}$ which satisfies $\mathbb{P}[S_{t+1} = s|S_t = s] = 1$. This makes sense when learning to play a game such as chess where end-scenarios or *terminal states* are unavoidable or even desired (checkmate). To apply the concept of episodes to a trading environment, we need to cut of the processes after a predefined number T of discrete time steps.

2.2.2 Policy

A policy π can be informally understood as a rule book according to which an agent is choosing its actions. We will model this as a function mapping a given action $a \in \mathcal{A}$ to a probability conditioned on the current state $s \in \mathcal{S}$. $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, $(a, s) \mapsto \pi(a|s)$. An agent currently following policy π then chooses its actions according to those probabilities. Over the learning process, the agent should adjust the policy to reflect the feedback given by the rewards to increase them in the next episode. Therefore, a good policy returns a higher probability for actions that will give a higher reward in the future and lower probabilities for actions with smaller expected rewards. This leads us naturally to the concept of an *optimal policy* whereby using it, the future reward is maximized. We will formally introduce the concepts of an optimal policy in section 2.2.4 after having provided some fundamental functions of reinforcement learning.

In all reinforcement learning algorithms introduced in this thesis the policy will be modelled with a deep neural network. This allows the agent to work in an environment where the state lies within a continuous space ($\mathcal{S} \subset \mathbb{R}^s$). We will refer to all RL algorithms modelling the policy directly by a network, as *policy-based* algorithms. Through the learning process, the network should be improved such that it will come as close as possible to representing an optimal policy. As opposed to common problems solved by neural networks in reinforcement learning, it is not directly possible to train the network by providing a label that can be compared with the output. We have to approximate such a label with the help of the sampled state, action and reward trajectory from the environment.
We will furthermore call an agent *greedy* with respect to a policy π if it always chooses the action with the highest probability and does not sample from the policy distribution. This is especially important if one considers an agent done learning and wants to only exploit the environment. By not allowing the agent to sample the actions from the output of the policy, exploration is stopped and unfavourable decisions are avoided. We will provided more information about exploitation and exploration in section 2.2.8.
Finally, we will follow the convention of using a subscript $\pi_{1,2,...}$ to differentiate between different configurations of a policy. Each of the policies will also have an respective set of neural network parameters represented by $\theta_{1,2,...}$. Therefore we sometimes also write π_{θ_1} to refer to the policy modelled by the network with the parameters θ_1.

2.2.3 Reward and Return Setup

The choice and design of the environment-specific reward function has a significant influence on the learning ability of a reinforcement learning agent and is one of the most important steps in RL. We, therefore, need a robust theoretical setup for modelling rewards and returns. A reward can be seen as numerical feedback of the environment to judge the preceding action at a particular state resulting in the transition from s to s'. As mentioned earlier, a MDP realisation can be summarised as a sequence of states, actions, and rewards. Training the agent in the environment we will get samples of these sequences through its interactions. The agent's goal will be to maximise the expected *discounted return* which we define as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.2.2}$$

with $t = 0$ and $R_{t+1}, R_{t+2}, ... \in \mathcal{R}$ where $\gamma \in [0, 1]$ is the *discount rate* for the future rewards. There is of course the option to have $T = \infty$. The properties of this sum allow us to write the development of G_t with the following update rule:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{2.2.3}$$

which will be useful for the actual implementation. Maximising the return for all possible sequences in a MDP gives us the objective function $J(\cdot)$ depending on the policy π_θ as an expected return of the above equation.

$$J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[G_0] = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{T} \gamma^k R_{k+1}\right] \tag{2.2.4}$$

where G_0 are discounted returns of episodes created under the policy π_θ. The problem the agent should then solve can be summarised by:

$$\max_\theta J(\pi_\theta) \tag{2.2.5}$$

In the special case of a trading environment, we need to reward actions that support a positive PnL and punish actions, which make the agent lose money. When trading only one mean-reverting signal, which represents a basket we will define the reward for an action as the change in PnL occurring through that action. This can be formalized as:

$$r(s_t, a_t, s_{t+1}) = M2M_{t+1} - M2M_t \tag{2.2.6}$$

with $s_t, s_{t+1} \in \mathcal{S}$ and $a_t \in \mathcal{A}$ where $M2M_t$ is the mark-to-market position of the current trading sequence. Moreover, we take the score of one whole trading sequence to be the sum of all rewards received during the trajectory. In cases where we can directly trade a mean-reverting signal this will necessarily also be equal to the final PnL. For the simulation part we will not control the risk associated with the trading. As shown by Yeo and Papanicolaou [27] it is possible to control the risk inherited in mean-reversion strategies when building the signal. Therefore risk controlling measures have not necessarily to be implemented in the trading behaviour.

2.2.4 State and Action-Value Functions

In addition to the policy, the value functions play an essential role in reinforcement learning, especially for the Actor-Critic methods introduced in later sections. The *state-value* function, as its name suggests, maps a state to its value, which is the expected return following a visit to that state. We write the function as $v_\pi(s)$ and formally define it as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right], \text{ for all } s \in \mathcal{S}$$

where π denotes the policy the agent is currently following and t any point in time. A more detailed version of the value function is provided by the *action-value* function. The action-value function maps two inputs namely state and action to the expected return following the action at that state and is defined as:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right], \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}$$

Knowing the true state- or action-value function would allow an agent to always choose the best action. Hence some reinforcement learning algorithms such as Deep Q-Learning focus on finding a good approximation of these functions instead of learning an optimal policy directly. An important property of both value functions is their recursive structure which is illustrated by equations (2.2.7) and (2.2.8).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \,| S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,| S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \,| S_{t+1} = s']\Big] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)\Big[r + \gamma v_\pi(s')\Big] \end{aligned} \tag{2.2.7}$$

with $s \in \mathcal{S}$ and similar transformations of the action-value function resulting in

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t \,| S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma\, G_{t+1} \,| S_t = s, A_t = a] \\ &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \,| S_{t+1} = s']\Big] \\ &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r | s, a)\Big[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')\Big] \end{aligned} \tag{2.2.8}$$

with $s \in \mathcal{S}$ and $a \in \mathcal{A}$. In both calculations from line one to two, we used the update rule of the returns in equation (2.2.3). Furthermore, the properties of the dynamics function and the policy were used to extract the random reward from the expectation in line three. The last step in the first transformation consisted only of replacing with the definition while for the second the characteristics of the policy function are used again. Equations (2.2.7) and (2.2.8) are known as the *Bellman equations* for v_π and q_π, respectively. Both equations formalise the relationship between the values obtained for the current input (state or state-action pair) and the possible successor inputs taking the environment dynamics into consideration. Moreover, we can also find a relation between the state-value and the action-value functions, which can be formally written as

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\, q_\pi(s,a) \tag{2.2.9}$$

with $s \in \mathcal{S}$. As mentioned before, having introduced the two central functions in reinforcement learning, we shall briefly provide an overview of the concept of an optimal policy. To determine the quality of a policy π, we first need to be able to compare policies. For this purpose we will use the state-value function and write $\pi \geq \pi'$ (policy π better than or equal to policy π') if and only if $v_\pi(s) \geq v_\pi(s')$ for all $s \in \mathcal{S}$. It is guaranteed that there is at least always one policy that is better than or equal to all other policies, but it is also possible for more than one policy to satisfy this condition ([28], page 62). We will refer to this policy, and all others that are equal in terms of v_π, as the optimal policy π^*. All optimal policies necessarily also share the same state-value function v_* and action-value function q_* which are defined as:

$$v_*(s) = \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S} \quad q_*(s,a) = \max_\pi q_\pi(s,a), \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}$$

These functions can give us an estimate of the maximum reward to expect when following the optimal policy. Earlier introduced concepts like in equations (2.2.7), (2.2.8) and (2.2.9) also hold for these optimal state- and action-value functions.

2.2.5 On-Policy vs Off-Policy

An important differentiation between reinforcement learning algorithms is the way they generate the data, that they learn from. For *on-policy* algorithms, the trajectory is produced by the same policy that the algorithm is improving. Compared to that, *off-policy* algorithms have an additional policy to produce sample trajectories, which are then used to enhance another policy. This makes off-policy algorithms more sample efficient since all samples can be stored and reused. This is not the case for on-policy methods since after each policy update a different policy is used for the sampling and hence the old samples need to be discarded. Leading to a lower sample efficiency for on-policy algorithms. Despite this shortcoming, on-policy methods are at least as effective and we will focus on these methods in this thesis. Some of the on-policy algorithms rely on the concept of *policy gradients* which we are going to introduce in the subsequent section.

2.2.6 Policy Gradient

Bringing the concepts of the policy and the objective function in equation (2.2.5) together, we can introduce the central concept to improve the neural network that approximates the policy in policy-based algorithms. By sampling a sequence of states, actions and rewards with the current policy π_{θ_1} we can calculate G_t. This, in turn, allows us to derive $J(\pi_{\theta_1})$ as defined in equation (2.2.4) which we can use as a loss for our neural network. Using the gradient of this loss to improve the network parameters lead to the following update rule:

$$\theta_{new} \leftarrow \theta_{old} - \alpha\, \nabla_{\theta_{old}} J(\pi_{\theta_{old}}) \tag{2.2.10}$$

The derivation of the gradient $\nabla_\theta J(\pi_\theta)$ can be found in the appendix in equation (A.0.2). We find that

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T} G_t\, \nabla_\theta \log \pi(a_t|s_t)\right] \tag{2.2.11}$$

Since we are working in a random environment where we do not know the four-argument dynamics function p (2.2.1), therefore we need to approximate $\nabla_\theta J(\pi_\theta)$. As mentioned earlier, this can be done by sampling a trajectory from the environment and calculating the policy gradient based on this data. This process is also known as Monte Carlo sampling and gives us the following approximation of the gradient:

$$\nabla_\theta J(\pi_\theta) \approx \sum_{t=0}^{T} G_t\, \nabla_\theta \log \pi(a_t|s_t) \tag{2.2.12}$$

Where in G_t we are using the rewards obtained in a sampled trajectory. This trajectory will always be sample with the most up-to-date policy (recall on-policy, after each update step old trajectories are discarded).

2.2.7 Actor-Critic Methods - A Comparison to DQNs

As the name already suggests, Actor-Critic algorithms use two independent parts to solve a reinforcement learning problem. The actor part is responsible for modelling the current policy π_{θ_a} whereas the critic part of the algorithm is used to learn the state-value function of the environment. Each of the components is modelled by a deep neural network. A special approach, mentioned here for completeness, is that the first layers of both networks can be shared and only the last layers separate the actor from the critic model and vice versa. This thesis will only focus on Actor-Critic models where the two networks are independent and do not share any layers or learning experience. The actor-network and consequently also the policy π_{θ_a} is improved using not only the rewards of a sampled sequence as in a policy gradient method but also makes use of the output of the critic-network, which is used to judge the action chosen by the actor. Hence, we hope to see an improvement over the pure policy gradient method in the testing environments. Furthermore, the critic-network is enhanced by comparing the current output at one state to the possible future rewards in the sampled sequence and calculating the loss based on the difference.
Compared to DQNs, one of the most commonly used reinforcement learning algorithms, next to Q-Learning, the Actor-Critic methods differ in several aspects. Nevertheless, the most significant difference lies in the way the final action is chosen. While DQNs are value-based algorithms choosing their actions using an approximation of either the state- or action-value function, Actor-Critic methods are policy-based and hence form their actions based on an estimation of the optimal policy. Consequently, also the learning process differs in some aspects. The DQN algorithms try to find the true or optimal value function, whereas the Actor-Critic methods attempt to learn the optimal policy in a direct manner. A further difference worth mentioning is that DQNs are off-policy algorithms while Actor-Critic methods are on-policy. As mentioned before, this makes DQNs more sample efficient since they can reuse previously generated trajectories. Actor-Critic methods, on the other hand, need to resample trajectories every time a learning step is performed and hence require a lot more sampling. One actual advantage of the resampling is that less memory during the learning process is required since there is no need to store the trajectories after the learning step has been performed. A third difference becomes apparent when actually training the models and is related to how the agents get to know the environment. While exploration has to be added to the DQN, for Actor-Critic methods one can sample from the provided distribution π_θ modelled by the actor. We will further describe the difference in exploration and exploitation of the environment in the upcoming section.

2.2.8 Exploration and Exploitation for Policy-based Algorithms

The trade-off between exploring the environment and receiving the highest possible reward is especially in trading a quite important one. Given, for example, a dynamic environment where the state space does not stay within fixed boundaries over time, one still wants the algorithm to learn from its actions. In this case, a completely fitted algorithm that does not choose actions different from the ones which are to itself, optimal will not adjust to the new environment and, over time, even lose its profitability. A further problem of only exploiting the environment is the possibility of arriving at a sub-optimal model state. In the worst case, the algorithm will stick to this configuration, only choosing the, again to itself, optimal action and not exploring possibly more profitable opportunities. This leads to the idea that in most models a small amount of exploration should also be present after a long training period. Only if the state space stays fixed, it might make sense to change to a full exploitation algorithm after the model has been fitted.
The ways to guarantee exploration in general and late in the learning process depend on the chosen model. The ϵ-greedy method along with its improved version Upper Confidence Bound are examples of ways to achieve this for DQNs. Policy-based methods, directly learning the policy, offer further ways to guarantee exploration. Exploration at the beginning of the training can be easily realised by sampling the provided policy and not taking the action with the highest probability. Later in the process, when one action will be dominant at a particular state, one can guarantee

some degree of exploration by forcing the policy to have a minimum probability for each action at every state.

2.2.9 State and Action Space

This short section defines the state and action space for the simulation problem where the algorithms are going to be tested. The goal again is to successfully trade a signal represented by an OU process.
A state for all three of the following RL algorithms will be a three-dimensional vector consisting of the current signal value, the current exposure and the signal value of the last exposure change. Hence the state space $\mathcal{S}$ will be $\mathbb{R} \times \{-1, 1\} \times \mathbb{R}$ with the important addition that the signal will be mean-reverting with mean zero. This leads to values far from 0 becoming less and less probable. For the OU Base Setup the state space can be assumed to be $\mathcal{S} = [-0.2, 0.2] \times \{-1, 1\} \times [-0.2, 0.2]$ without significant loss in approximation of the environment. We will train the agents with a random start exposure of either -1 or 1 for each episode.
The action space will only consist of two actions which will be either to change the current exposure (buy two units or sell two units depending on the position) or do nothing and keep the position as is. Consequently, we get $\mathcal{A} = \{0, 1\}$ as our action space where 0 will be no action and 1 will be the position change.

Chapter 3

Extracting Trading Signals

Several methods to extract mean-reverting signals are provided, from which the PCA signal will later serve as input to the machine and reinforcement learning trading agents. Statistical methods, as well as machine learning algorithms are used to produce these signals.

3.1 Principal Component Signals and Results

The first method introduced will serve as a the testing environment for the RL agents. It is rooted in the previously mentioned paper by Avellaneda and Lee [10] and uses Principal Component Analysis (PCA) to extract an error term which is found to be mean-reverting. We will start by taking a set of data $(X^1, ..., X^N) \in \mathbb{R}^{N \times m}$ where each X^i for $i \in [N]$ is a vector of samples of a random variable (this will be the price series of one asset in our case). We calculate the sample covariance matrix as $\Sigma = (\sigma_{ij})_{1 \leq i,j \leq N}$ with

$$\sigma_{ij} = \frac{1}{m-1} \sum_{k=1}^{m} (x_{ki} - \overline{x}_k)(x_{kj} - \overline{x}_j)$$

where x_{ij} is the i-th sample of the random variable X^j and

$$\overline{x}_i = \frac{1}{m} \sum_{k=1}^{m} x_{ki}$$

This is followed by the calculation of the eigenvalues $(\lambda_1, ..., \lambda_N)$ and eigenvectors $(v_1, ..., v_N)$ of Σ. As a next step, the eigenvectors are sorted according to the decreasing order of the eigenvalues. Finally by normalising the eigenvectors to length 1 we obtain the Principal Components in order. We will refer to them in the remaining part as $(p_1, ..., p_N)$. Taking these Principal Components, we find that the first few components cover a large chunk of the variation in the data set. In most cases (if not all N time series are pairs-wise uncorrelated) N-1 components cover almost all variation that appears in the raw data set. Hence the last components end up behaving rather like white noise and will therefore be close to mean-reversion. We calculate the ratio of explained variation r^i_{var} for each p_i with $i \in [N]$ as defined in (3.1.1).

$$r^i_{var} = \frac{\lambda_i}{\sum_{j=1}^{N} \lambda_j} \tag{3.1.1}$$

For the data introduced earlier, the first two Principal Components of the price data alone explain 99.99% of the variance $(r^1_{var} + r^2_{var})$. We, therefore, take the sum of the remaining two components as our signal and test it for mean-reversion/stationary properties using the Augmented Dickey-Fuller test [29]. It turns out we can reject the null hypothesis with a p-value smaller than 0.00001 and hence conclude that the time series can be assumed to be stationary 'enough' for trading purposes and can be used as a mean-reverting trading signal. Furthermore, we find that the mean stays relatively constant at -0.48 when splitting the data in 10 equally sized chunks and calculating it on each of them. Figure B.1 shows the raw signal.

Having found a possible trading signal we test its profitability using Bollinger Bands with different window sizes and several distance factors. The results of the trading analysis are shown in figure 3.1

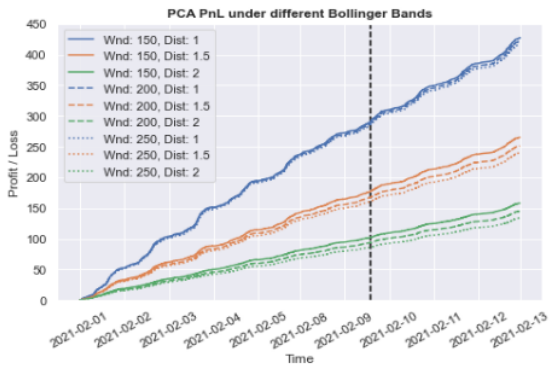

Figure 3.1: Different PnL results for the Principal Component trading strategy with window sizes of 150, 200 and 250 (indicated by different colors) and distance factors of 1, 1.5 and 2 (indicated by line type). PnL is in indicated USD.

We can observe that profit seems to be more dependent on the distance factor than on the window size of the Bollinger Bands. This also underlines our intuitive understanding since a more significant gap between the bands allows for fewer trades. Although those trades would lead to a higher return (since the traded gap is larger) the PnL is reduced by missing some of the opportunities the closer bands can pick up on. In addition, we find that with increasing window size, the return is reduced slightly. The graphic also suggests that this effect is more dominant for Bollinger Bands with higher distance factors. Indeed, the longer window size leads to a smaller variance of the mean-reverting signal and therefore, trades are less profitable despite using a higher distance factor.

3.2 FNN Signals

As our first application of neural networks, we implement a simple ANN to combine the four input asset prices to a mean-reverting signal. We try to achieve this by defining a custom loss function on the output signal, which will be only one dimensional. The architecture of the neural network is described in table 3.1.

Layer number	Layer kind	Input dim.	Output dim.	Activation function
1	Dense	4	2	id/Sigmoid
2	Dense	2	1	id

Table 3.1: Table of the hidden layers of the neural network used for the signal extraction.

We will run two approaches, where one will be only using linear activation functions (id) and the other one will be using the non-linear Sigmoid activation function. For the linear network it would also be possible to use Linear Regression, but using neural networks, we will still benefit from automation and the opportunity for wider extensions. Furthermore, the loss function is easier to customise and the linear activation functions guarantee that the produced signal will be replicable directly by the asset prices. Moreover, due to the similarity of the model the comparison to the non-linear model is easier. For this non-linear network, we will have to replicate the signal using a linear regression from the asset prices on the signal. The input of the model will be the prices introduced in section 1.4. The networks are fitted using consecutive observations to maintain the time series structure of the data. A sequence is chosen to be 5000 time-steps long and the batch size b is set to 128. Consequently, the networks produce 128 signals of length 5000 as their output

for one batch. We will use a learning rate of 0.0005 for training.
The special feature of these neural network is given by the loss function, which is calculated on the previously mentioned output. For the first loss function we will proceed as follows: Let $\overline{x}_i$, with $i = 1, ..., 128$ and σ_i, with $i = 1, ..., 128$ be the sample mean and the sample standard deviation of each signal in the batch, respectively. We then calculate the *mean-variance loss* as:

$$\mathcal{L}_{mv}(x) = \sqrt{\frac{1}{b-1}\sum_{i=1}^{b}\left(\overline{x}_i - \frac{1}{b}\sum_{j=1}^{b}\overline{x}_j\right)^2} + \left(\frac{1}{b}\sum_{i=1}^{b}\sigma_i\right)^{-1} \tag{3.2.1}$$

By controlling the extent to which the mean is allowed to vary while rewarding a larger variance in the signal, two main properties of a profitable mean-reverting signal are emphasised.
A further loss function, we will use to produce a mean-reverting signal, is based on the variance ratio approach by Lo and MacKinlay [26]. We calculate the loss function value as an estimation of the variance ratio function:

$$\mathcal{L}_{vr}(x) = \sum_{i=1}^{n}\frac{\mathbb{V}[x_n]}{\mathbb{V}[x_1]n} \tag{3.2.2}$$

where x is one output sequence and, with slight abuse of notation, with x_n we refer the output process up to and including the n-th period starting from the beginning. We will set the length of one period to 100, which results in 50 processes with increasing size for a output of length 5000. The loss of one whole batch is the arithmetic mean of the single losses for each output process. We will refer to this loss function as the *variance-ratio loss*. An example of the difference between the variance ratio loss of a mean-reverting process and an arithmetic Brownian motion with zero drift and volatility $\sigma = 10$ is shown in figure 3.2

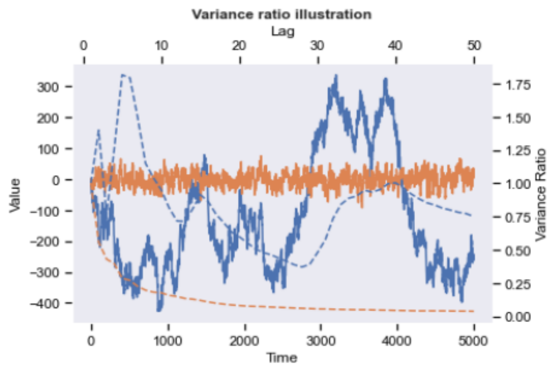

Figure 3.2: Figure showing an arithmetic Brownian motion (blue, solid line) and a mean-reverting OU process (orange, solid line) together with the variance ratio functions (blue, dashed for Arithmetic Brownian Motion and orange, dashed for OU process) obtained by calculating the variance ratio at different lags.

We can see that for increasing lag values, the variance ratio function of the mean-reverting process goes to 0 asymptotically. In contrast, the variance ratio loss for the arithmetic Brownian motion increases drastically towards the end of the process. Hence, it is reasonable to investigate the mean-reverting behaviour of a time series using the integral of the variance ratio function or any approximation of it.
For the signal extraction we will combine each network with each loss function and we will refer to the four resulting models as FNN VR NL (variance-ratio loss + non-linear network), FNN MV NL (mean-variance loss + non-linear network), FNN VR L (variance-ratio loss + linear network) and FNN MV L (mean-variance loss + linear network),

3.3 Autoencoder Signals

As mentioned before, we use an autoencoder as the second model to extract a mean-reverting signal along with the two given loss functions of equations (3.2.1) and (3.2.2). Table 3.2 provides a quick overview of the architecture of the autoencoder, which can be seen as an extension of the previous neural network, by just mirroring the first two layers to the back of the network. The original network now forms the encoder and the copied part builds the decoder. Using an autoencoder will ensure that the produced signal does not lose its relation to the overall market. This might happen for neural networks where the signal is not used to replicate the market. We will again use a linear and a non-linear approach with the same activation functions.

Layer number	Layer kind	Input dim.	Output dim.	Activation function
1	Dense	4	2	id/Sigmoid
2	Dense	2	1	id
3	Dense	1	2	id/Sigmoid
4	Dense	2	4	id

Table 3.2: Table of the autoencoder architecture used for signal extraction from asset prices.

All other fitting parameters as well as the overall fitting procedure will stay the same. The only difference will be in the loss calculation where, we will add the mean squared error of the network output and target to the two loss functions mentioned in the neural network trading section above. The custom loss functions are still calculated on the one-dimensional output, which we extract from the second layer of the autoencoder. We will again refer to the 4 different models using the convention from the previous section while replacing FNN with AE.

3.4 Neural Network Signal Performance

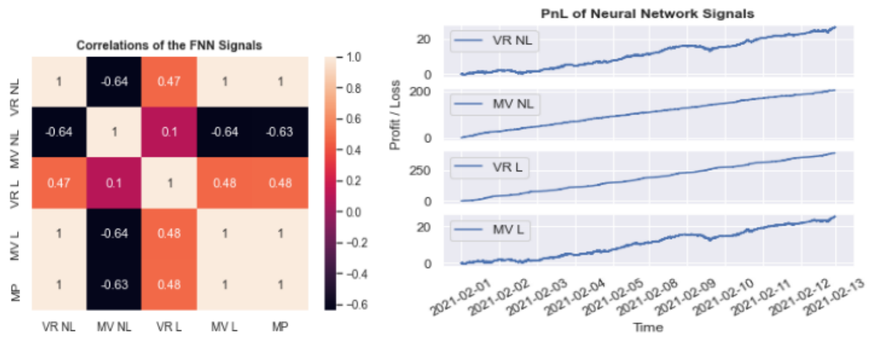

Figure 3.3: Left: Plot of the correlation matrix for the signals produced by the different FNN models including the mean price. Right: PnL development for the mentioned signals over the two-week period. PnL is in indicated USD.

We test all introduced neural networks on the data from section 1.4, beginning the analysis with the FNNs. All signals are evaluated using Bollinger Bands with a window size of 200 and a distance factor of 1.5. The left plot of figure 3.3 visualises the correlation between the different signals and the mean process of the asset prices. We include the mean price process to illustrate a drawback in the signals, which we found during the analysis of the signals. It becomes evident that two signals show a significantly high correlation with the mean prices. These are the signals produced by the FNN VR NL model and the FNN MV L model. We expect that for these signals no actual mean-reversion could be detected. This is underlined by the plot of the actual raw signals shown in figure B.7, which shows the similarity of the two signals and no mean-reverting behaviour. We scale all asset weights obtained by the signal such that the absolute basket value is 100USD. This is slightly advantageous for signals focusing on the cheaper assets, but is necessary to make the signals more comparable.

The PnL shown on the right side in figure 3.3 depicts some promising results with the middle plots showing an almost linear performance similar to the one seen for the PCA signal. These PnLs were generated by the FNN MV NL model and the FNN VR L model which both only having a correlation to the mean price of -0.63 and 0.48, respectively. As expected, the performance for the other two signals is a lot worse and unstable.
When conducting the analysis for the signals produced by the autoencoder, we again find signals that are highly correlated with the mean price. The left plot of Figure 3.4 shows that the correlation between the mean price and either of the two linearly produced signals is 1. For autoencoders this is expected to happen for the linear models since the mean price is the optimal solution to the mean squared error loss when there is only one regressor. We furthermore observe that the correlation between the signal of the AE VR NL model and the mean price process is also relatively high and by investigating B.7 we find that the signals appears to follow a slight trend.

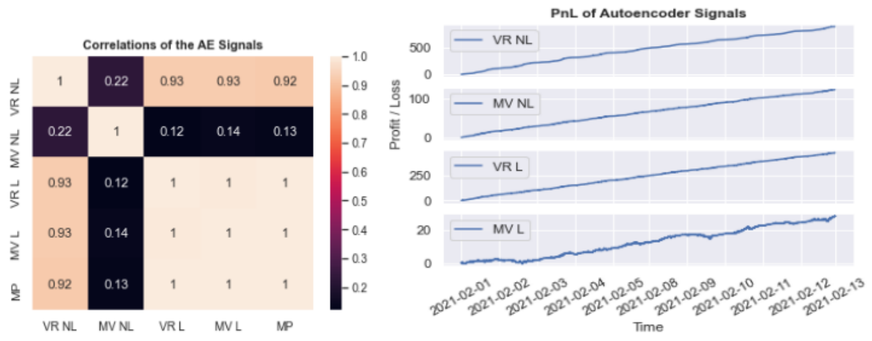

Figure 3.4: Left: Plot of the correlation matrix for the signals produced by the different AE models including the mean price. Right: PnL development for the mentioned signals over the two-week period. PnL is in indicated USD.

We again see that the PnL for one of the models (AE MV L) correlated to the mean price is relatively unstable. Surprisingly, the PnL of the other linear model signal shows a good performance and we will later investigate this further.

Model	**PnL**	**PnL/Net BV**	**Avg. Net BV**	**Asset Weights**
FNN VR NL*	26.77	0.28	94.51	1.29 — -0.94 — 1.13 — 0.52
FNN MV NL	204.22	80.72	2.53	5.98 — 3.90 — 0.40 — -0.85
FNN VR L	389.91	92.62	-4.21	12.76 — -15.18 — -0.16 — 0.20
FNN MV L*	25.47	0.26	96.84	1.30 — -0.39 — 1.18 — 0.47
AE VR NL	**883.85**	166.14	5.32	-2.14 — 2.77 — -0.72 — 0.79
AE MV NL	122.71	360.91	0.34	-1.14 — 1.10 — -0.81 — 0.82
AE VR L*	472.35	7.32	64.57	2.01 — 0.64 — -0.31 — 1.32
AE MV L*	28.17	0.3	93.62	0.75 — -1.05 — 1.09 — 0.57
PCA	472.2	**3577.24**	**0.13**	-0.66 — 0.72 — 0.85 — -0.85

Table 3.3: Summary table of the final signals. BV stands for basket value. *Models only producing a replication of the mean price

Table 3.3 summarises the results of the signal extraction models. We here focus on the relative performance of the models with respect to their net basket value (calculated as sum of average asset price times weight for the asset). The net basket value is used as a simple measure to judge on the mean-reverting properties of the signal. It makes evident that the signals replicating the mean price are indeed only following the market and hence should not be used for mean-reversion trading. Having Net BV values of over 90 for each of these signals indicates that the models did not pick up the idea of mean-reversion. The AE VR L value has only a value of 64.57 for the Net BV and might therefore have not completely picked up the mean price process which would explain the unexpected good performance. The relative performance as well as the Net BV values for the remaining models show promising results. Although none of the models outperforms the PCA signal on a relative basis, we find that the AE VR NL shows a stronger absolute performance than all other models including the PCA model.

A word of caution — Although, it is possible to find a profitable signal using the approach of replicating a signal produced by a non-linear procedure, the causal connection between the signal and the PnL needs to be investigated deeper and requires more back-testing than other strategies. Overall, the signals produced by the neural networks in combination with the custom loss funtions show very promising results and motivate further analysis in the automation of signal extraction. Although, none of the usable signals of the introduced models could outperform the classic PCA signal on a relative basis, the produced output shows that some of the models understand the problem and find a profitable mean-reverting signal.
For the remaining chapters of this thesis, we will assume that we directly trade a signal such as the PCA signal (basket of assets) and we will focus our attention on optimising the trading decisions with the use of reinforcement learning.

Chapter 4

Trading Agents and Simulation

NOTE

Please be advised that parts of this chapter include animations to visualise content. To obtain full access, it is therefore recommended to view this document with a PDF-Reader. The document has been tested on ADOBE ACROBAT READER DC

This chapter introduces the algorithms used to trade on the previously found mean-reverting signals. The Bollinger Bands will represent the benchmark trading. We aim to improve trading behaviour by using machine and reinforcement learning. In this, the focus lies on one pure supervised algorithm: A LSTM neural network and three different reinforcement learning algorithms: the REINFORCE, the Advantage Actor-Critic (A2C) and the Proximal Policy Optimization (PPO) algorithm.

4.1 LSTM Trading Agent

We decided to use LSTM neural networks as a trading agent since the properties mentioned in the introduction about LSTM networks provide advantages when dealing with time-series data. For example, thanks to the different gates in a LSTM Cell, relationships between data points of varying time gaps can be found and time-delayed effects of events can be investigated with more precision.

4.1.1 Algorithm and Setup

Our agent will be a vanilla LSTM neural network with two hidden *LSTMCell* (meaning that the layers do not return sequences) layers and one hidden dense layer as introduced in the second chapter of this thesis. The exact architecture is described by table 4.1

Layer number	Layer kind	Input dim.	Output dim.	Activation function
1	LSTM Cell	3	512	tanh
2	LSTM Cell	512	512	tanh
3	Dense	512	1	tanh

Table 4.1: Table of the neural network architecture of the LSTM trading agent.

All layers are fully connected and we allow for a bias in every node. The input vector will consist of the specific mean-reverting signal and two difference time series. The first one being the difference between the actual signal and an exponential weighted moving average (EWMA) of the signal with a half-life of 64. The second will be the difference of EWMAs with half-lives of 64 and 128, respectively. The tanh is used as the activation function for both LSTM Cell layers as it is the default choice for LSTM networks. To map the output of the last layer to an asset weight which can then be realised in the market we scale it to the interval $[-1, 1]$ using again the tanh activation function. This achieves two things: The traded volume does not blow up in the attempt of the agent to increase the PnL and the final results of the model are more comparable to the results of other models. The output of the network represents the new position in the asset basket represented by the signal. Furthermore, we will only focus on the sign of the output, meaning the possible positions will only be -1 and 1 and they are only changed if the output of the network changes its sign. This is done to compare the results with a reinforcement learning agent that acts within a discrete action space.

The training process and especially the loss function build the core part of the LSTM trading agent. In our case, those vary significantly from the typical supervised learning approach in which neural networks are usually improved by comparing the output of the network with a label given for each observation. Using the previously mentioned features and their points in time as an observation, we do not have such a label telling the network what the correct action would be. We build our loss function by focusing on the main goals of trading.
The trading agent has two objectives when acting in the environment. Firstly, it should be profitable and generate a reliable PnL over different market periods. Secondly, the maximum drawdown should be low, which can be controlled quantitatively by reducing the standard deviation of the returns. This leads us naturally to the problem of maximising the Sharpe Ratio during the learning process. We define the Sharpe Ratio formally as

$$SR = \frac{\mathbb{E}[R_{agent} - R_f]}{\sigma_{agent}} \tag{4.1.1}$$

with R_{agent} being the returns generate by the agent and R_f the risk free rate. We will calculate the loss of the neural network as the negative value of the Sharpe Ratio. The loss for one training batch is therefore calculated as outlined by algorithm 2.

Algorithm 2: Loss calculation for LSTM trading agent

```
1:  for observation in batch do
2:    position = 0
3:    cash = 0
4:    pnl = 0
5:    pnls = emptylist()
6:    for i in 1, ..., T do
7:      new_position = GetNewPositionFromModel($observation_i$)
8:      position_change = new_position - position
9:      cash = cash + position_change * $signal\_price_i$
10:     pnl = cash + new_position * $signal\_price_i$
11:     position = new_position
12:     pnls.append(pnl)
13:   end for
14: end for
15: pseudoSR = Mean(pnls) / Stddev(pnls)
16: Loss = -pseudoSR
```

where T is the length of one sampled time series and the function *GetNewPositionFromModel()* returns the new position by feeding the observation at the current time t into the LSTM trading agent. It is important to mention that we do not calculate the Sharpe Ratio as defined in (4.1.1). This is because we assume the risk free rate over short periods to be too small and therefore negligible.
The specific parameters for the training can be found in table 4.2. Having a sequence length of 300 (in this case, the training sequence) is equal to using $N = 300$ when generating the OU process (see 1. section 1.4.1). Together with the batch size of 128, this results in an input tensor with dimensions (128, 300, 3) on which the loss is calculated as shown in algorithm 2. We use the standard Adam optimizer to update the weights and biases. The following section summarizes the results of the agent performing in the trading simulation environment.

Parameter	Value
Train Sequence	300
Test Sequence	3000
Batch Size	128
Learning Rate	0.001

Table 4.2: Table containing the basic parameters for the training of the LSTM Trading Agent

4.1.2 Trading Simulation

To verify the algorithm's potential in a real market environment, we first test its learning capabilities with a simulated OU process as our signal. We will use the OU Base Setup, introduced in the first chapter, to generate the process and train the LSTM agent according to the previous section. Figure B.2 shows the learning process of the model, while figure 4.1 summarises the results of the training.

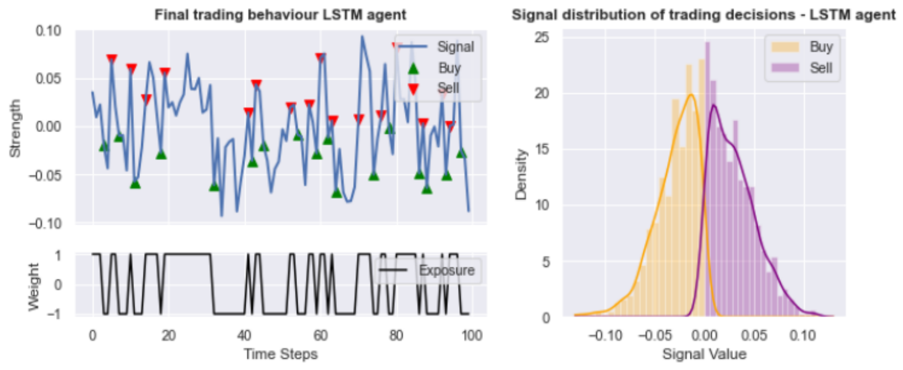

Figure 4.1: Left: The top plot shows the LSTM Agent's decisions on a sample signal where red downwards pointing triangles and green upwards pointing triangles show sell and buy actions, respectively. The bottom plot shows the agent's position over the trading period. Right: Distribution of the prices where buy and sell actions were taken. The left orange distribution represents buy actions and the right purple illustrates sell actions.

It can be seen that most of the actions the agent takes are profitable, although it misses some opportunities visible in the interval of $[20, 40]$ on the timeline. By repurchasing the asset earlier followed by an almost immediate sell it could have made another profitable trade, taking a slightly higher risk of a loss. Another important observation can be made by investigating the right plot of figure 4.1: The agent seems to only buy the asset when the signal is negative and only sell it when it is positive. Neither a buy nor a sell action is taken on the respectively other side. This results in no overlap of the two histograms and almost avoids trading for a loss. This showing that the network seems to 'understand' the problem. The trading amount of the model is reasonable given that it can pick up on many opportunities but still misses some potential trades.

4.2 REINFORCE Agent

4.2.1 Algorithm and Setup

The earlier sections provide us with the building blocks to introduce the particular algorithms. The simplest of the RL algorithms used in this thesis is the REINFORCE algorithm. The REINFORCE algorithm was first introduced by William in 1992 [18] and therefore is one of the oldest RL algorithms used here. It is only based on the policy gradient defined in equation (2.2.11) and the update rule of equation (2.2.10). We will use a deep neural network to model the policy π_θ, whereas before θ represents the current network weights. Table 4.3 summarizes the architecture of this network.

Layer number	**Layer kind**	**Input dim.**	**Output dim.**	**Activation function**
1	Dense	3	256	ReLU
2	Dense	256	256	ReLU
3	Dense	256	2	Softmax

Table 4.3: Table of the neural network architecture of the REINFORCE policy network.

The network consists of two hidden layers, each with 256 nodes and both using the ReLU activation function. The network is fully connected and the output of the last layer is activated with the Softmax activation function. Furthermore, all layers allow for biases. The last activation is done to normalize the output vector to a length of one, which will enable us to interpret the output as probabilities for the possible actions. This network together with the method of policy gradient builds the core of the REINFORCE algorithm as shown in algorithm 3.

Algorithm 3: REINFORCE algorithm

```
1: Initialize neural network weights $\theta$ for policy $\pi_\theta$
2: Set learning rate $\alpha$
3: Set discount rate $\gamma$
4: Set sequence length $T$
5: for episode = 1,..., $N\_EPISODES$ do
6:     Sample sequence $S_0, A_0, R_1, S_1, A_1, R_2, ..., R_T$
7:     Set $\nabla_\theta J(\pi_\theta) = 0$
8:     for $t = 0, ..., T-1$ do
9:         Calculate $G_t = \sum_{\bar{t}=t}^{T-1} \gamma^{\bar{t}-t} R_{\bar{t}+1}$
10:        $\nabla_\theta J(\pi_\theta) = \nabla_\theta J(\pi_\theta) + G_t \nabla_\theta \log \pi(a_t|s_t)$
11:    end for
12:    $\theta = \theta + \alpha \nabla_\theta J(\pi_\theta)$
13: end for
```

After the initialization of the policy network, sequences are sampled from the environment using the current policy. Based on this experience the gradient is calculated using the sum of discounted future rewards and the network is updated accordingly. The process starting with the environment sampling is repeated $N\_EPISODES$ times. The simplicity of the algorithm makes it rather easy to implement and allows for a lower run-time compared to more complex methods.

4.2.2 Trading Simulation

As for the LSTM Trading Agent covered before, we will test the REINFORCE Agent in a simulated trading environment using an OU process. We will again use the OU Base Setup to generate the processes and train the agent for a maximum number of 2000 episodes. The training parameters for the model are chosen as follows: $\alpha = 0.0001$, $\gamma = 0.9$ and $T = 300$ (=length of one episode). The left side of figure 4.2 shows the trading activity of the algorithm on the sample process, while the right side again shows the price distribution of the actions.

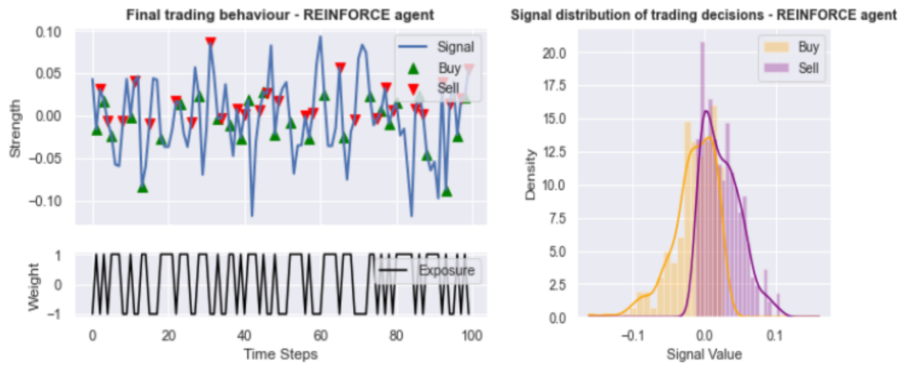

Figure 4.2: Left: The top plot shows the REINFORCE Agent's decisions on a sample signal where red downwards pointing triangles and green upwards pointing triangles show sell and buy actions, respectively. The bottom plot shows the agent's inventory over the trading period. Right: The plot shows the distribution of the signal values where a buy or sell action was taken. The yellow histogram belongs to the buy actions and the purple one to the sell actions.

The first observation one can make is that the average trading frequency is much higher for the REINFORCE agent compared to the LSTM. In detail, this simple reinforcement learning model trades on average 44.9 times in a window of 100 time steps. For the same time frame the LSTM agent only averages 23.6 trades. Furthermore, we find that the action are all taken quite close to the long-term mean of the process (0) and the agent rarely trades large signal jumps.
Moreover, we see that the distributions of the signal values for buy and sell actions now overlap. On one hand this allows the agent to increase the number of profitable trades but on the other hand can also lead to a quite fast trading close to the mean as we can see in the left plot. It would therefore better for the agent to wait for more profitable trades. We will furthermore expect this behaviour to change if we introduce 'trading costs' to the signal as we will do in chapter 5.

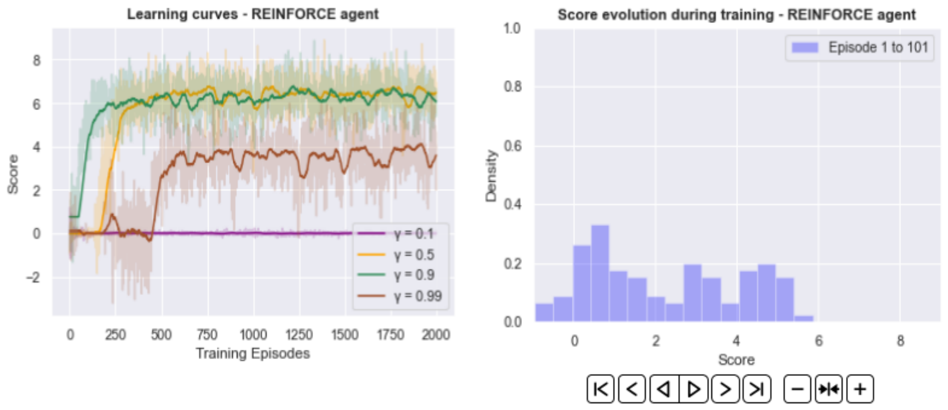

Figure 4.3: Left: The figure shows learning curves of the REINFORCE agent which are moving averages of the score received by the agent in every training episode with a window size of 20. The transparent lines are the actual scores for the respective parameter settings. We used the same settings as before only changing γ according to the legend. Right: The sequence shows the evolution of the score distribution over the first episodes for the agent with $\gamma = 0.9$

Figure 4.3 shows that the maximum average score the agent receives during trading is around 6.2. Still, the actual scores vary quite significantly taking values from as low as four to sometimes over eight. The settings with a discount factor of 0.5 and 0.9 seem to work best with the model with

$\gamma = 0.9$ learning slightly faster. A value of 0.99 for γ appears too high for the algorithm to pick up the full trading potential in the signal and a value of 0.1 does not allow the agent at all to make informed trading decisions. The connection established between consecutive actions is not strong enough to draw conclusions about the best long-term strategy. Analysing the evolution of the score distribution, we find that the score variance is reduced over the time and that after reaching the final distribution no more improvement is achieved. The standard deviation is reduced from 1.87 for the first 100 episodes to 0.78 for the last 100 episodes.

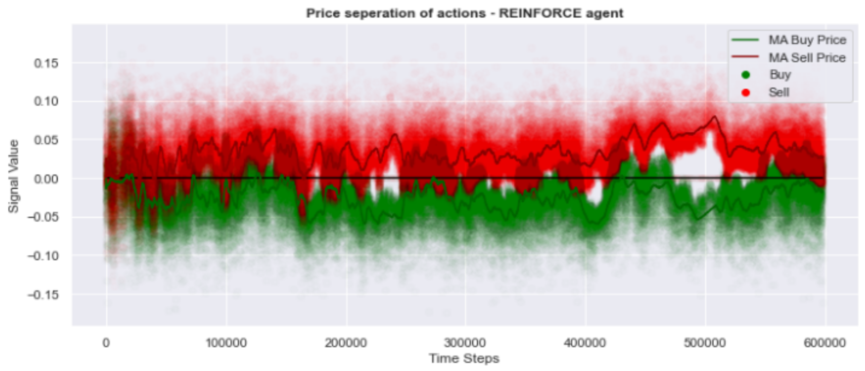

Figure 4.4: The figure shows a scatter plot of the signal values at which actions were taken by the REINFORCE agent over the duration of the training period. Red and green dots represent sell and buy actions respectively. The line plots indicate the moving average of the signal values again split by action using a window of 100 time steps. The used setting is as before including $\gamma = 0.9$.

The plot in figure 4.4 illustrates the separation of the actions over the whole training period. After the first 50,000 steps, a pattern is evolving that maps buy actions to lower and sell actions to higher prices. Moreover, after 200,000 simulation steps, the buy price average seems to stay below 0 and the sell price average stays above 0. Interestingly, depending on the recent learning experience, the average values of actions prices still vary quite a lot. The diversion get significantly stronger towards the final learning steps of the illustrated learning period. It is therefore reasonable to decrease the learning rate over time, to avoid short term deviations from a good strategy but catch long-term changes in the environment. In the next section, we will introduce the first Actor-Critic method and analyse if the more complex algorithm will help to improve the previously mentioned results.

4.3 Advantage Actor-Critic Agent

4.3.1 Algorithm and Setup

The Advantage Actor-Critic algorithm is the first of two Actor-Critic methods we will apply to the trading problem. As mentioned before, the algorithm is implemented using two deep neural networks. The actor-network θ_a modelling the policy and the critic-network θ_c modelling the state-value function. Regarding the architecture for the actor-network, we will use the same as shown in table 4.3. The only difference for the critic-network will be the last layer with an output dimension of only 1 using the identity as activation function. The remaining activation functions and the number of hidden layers, as well as nodes, are the same as for the actor-network.
The 'Advantage' in the name of the model refers to the *advantage function*, which is used by the algorithm to improve the results and learning compared to other policy gradient methods such as the REINFORCE algorithm. The intuition behind the advantage function is that we want to take action at certain states that provide a reward advantage compared to the other possible actions at the state. The function itself is defined as

$$A^{\pi}(s_t, a_t) = q_{\pi}(s_t, a_t) - v_{\pi}(s_t) \tag{4.3.1}$$

We consequently need to estimate the state- and action-value functions. Modelling the state-value function with the critic-network, we need to estimate q_π from v_π. One could also train a third neural network to learn q_π, but this would be too computationally expensive ([30], page 137). Finding an estimate of the action-value function depending on v_π we start with the definition of q_π found in the first line of equation (2.2.8).

$$\begin{aligned} q_{\pi}(s,a) &= \mathbb{E}_{\pi}[G_t \,|S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[R_t + \gamma R_{t+1} + \gamma^2 G_{t+1} \,|S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+N} \,|S_t = s, A_t = a] + \gamma^{t+N+1} v_{\pi}(s_{t+N+1}) \\ &\approx R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+N} + \gamma^{t+N+1} v_{\pi}(s_{t+N+1}) \end{aligned} \tag{4.3.2}$$

with $N \in \mathbb{N}$, we will refer to this estimation as the N-step estimation of q_π. This is also why we are going to refer to $A(s_t, a_t)^\pi$ as the *N-step advantage function* $A(s_t, a_t)^{\pi}_{Nstep}$. The value of $v_\pi(s_{t+N+1})$ is obtained using the critic-network while the rewards are samples from the current trajectory. The advantage function compares the chosen action to the other possible actions at the respective state. It assigns a positive value (advantage) to actions better (meaning higher action-value) than the average action at that state (state-value) and a negative value (disadvantage) otherwise. This advantage is then used to learn the best actions at each state through an adjusted policy gradient method. One improvement of using the advantage function over other methods is that it provides a score for the quality of the action in relative terms and does not judge the action with respect to the current state of the policy. This makes learning more focused on the state-action pair at hand and less influenced by the current quality of the policy.

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}\left[A^{\pi}_t \, \nabla_{\theta} \log \pi(a_t|s_t)\right] \tag{4.3.3}$$

This gradient (4.3.3) is as before used to update the network weights θ_a of the actor. The critic-network is improved by approximating a target state value and calculating the squared difference to the state value given by the critic-network.

$$\begin{aligned} v_{tar}(s_t) = q(s_t, a_t) &= R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... + \gamma^n R_{t+n} + \gamma^{n+1} \hat{v}_{\pi}(s_{t+n+1}) \\ &= A^{\pi}_t(s_t, a_t) + v_{\pi}(s_t) \end{aligned} \tag{4.3.4}$$

is used as an approximation and the loss for the critic-network will be mean squared error over all states in the trajectory. Finally, the whole A2C procedure is summarised by algorithm 4.
After initializing the two networks, setting the learning rates and the discount factor, we sample state-action-reward sequences with length T, $N\_EPISODES$ times while calculating the previously mentioned metrics for all of them and subsequently updating the networks during each iteration.

Algorithm 4: Advantage Actor-Critic algorithm (N-Step)

```
1:  Initialize weights $\theta_a$ and $\theta_c$ for actor and critic-network
2:  Set learning rate $\alpha_a$ and $\alpha_c$ for actor and critic-network
3:  Set discount rate $\gamma$
4:  for episode = 1, ..., N_EPISODES do
5:      Sample and store data tuples $(s_t, a_t, r_t, s_t')$ until $t = T$ using $\pi_{\theta_a}$
6:      for t = 0, ..., T-1 do
7:          Calculate $\hat{v}(s_t)$ using the critic-network
8:          Calculate $A_{Nstep}(s_t)$ using the critic-network and the stored rewards
9:          $v_{tar}(s_t) = A_{Nstep}(s_t) + \hat{v}(s_t)$
10:     end for
11:     $L_{critic}(\theta_c) = \frac{1}{T}\sum_{t=0}^{T}(v_{tar}(s_t) - \hat{v}(s_t))^2$
12:     $L_{actor}(\theta_a) = -\frac{1}{T}\sum_{t=0}^{T}(A_{Nstep}(s_t)\,\nabla_\theta \log \pi(a_t|s_t))$
13:     $\theta_c = \theta_c + \alpha_c\, L_{critic}(\theta_c)$
14:     $\theta_a = \theta_a + \alpha_a\, L_{actor}(\theta_a)$
15: end for
```

4.3.2 Trading Simulation

The trading simulation for the A2C agent is run with both learning rates α_a and α_c being equal to 0.0001, the sequence length T being 300 and the discount rate fixed at 0.9. Analysing the trading behaviour of the A2C agent by investigating the plots of figure 4.5 we see that the agent picks up the overall idea of the problem and in general, buys low and sells high. This is especially well illustrated by the right distribution plot, which shows only a slight overlap of the two histograms. Only around a signal value of 0 buy and sell actions seem to share some actions. This small overlap and also the fact that the means of both distributions are close to 0 indicates already what is shown by the left plot: Many of the trades are close to zero, which most of the time leads to a swift selling and buying of the signal. This results in a trading frequency of around 30.8 trades per 100 time steps. This behaviour is subject to changes as soon as trading costs are introduced since profits of single round trip trades will be too small to cover all execution costs.

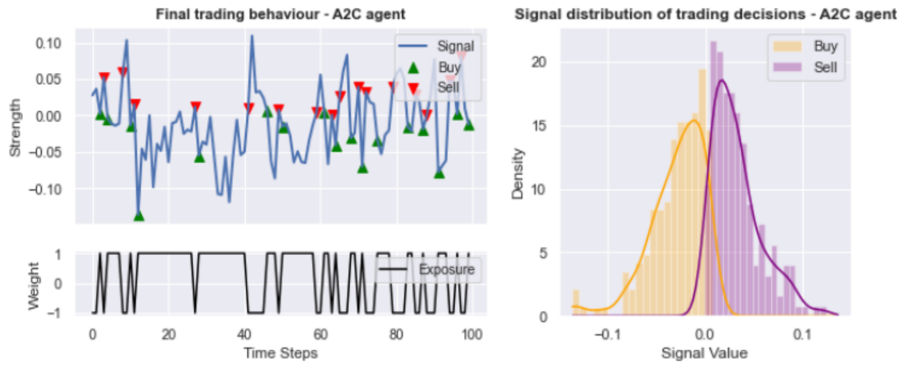

Figure 4.5: See figure 4.2 for further description. Agent parameters as described with $N = 40$.

The left plot of figure 4.3 illustrates that the A2C agent performs best with the number of steps being either 10 or 40. Both seem to deliver quite similar results with an average score of around 6, although the performance of the agent with 40 steps appears to be slightly more stable. The agents using either 5 or 20 steps to estimate the action-value are not able to achieve an average score higher than 4.1. For the better-performing agents, the score never comes back below 4 in the late stages of the training. Furthermore, the development of the score distribution shows a good and continuous learning performance. The standard deviation of the score distribution can be reduced to 0.69, which is smaller than for the REINFORCE Agent. So far it seems a score a little above 6 is close to the optimal solution for the problem.

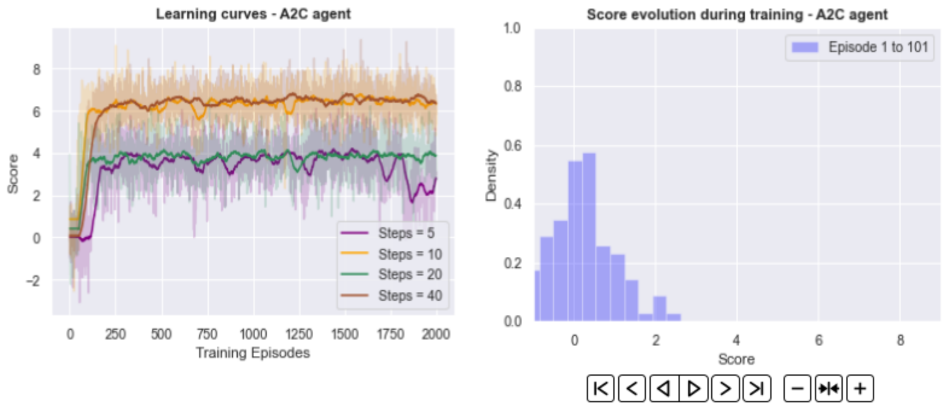

Figure 4.6: See figure 4.3 for further description. Steps are the advantage estimation steps N. Agent parameters as described with right plot using $N = 40$.

The price separation of the A2C agent shown in figure 4.7 shows slightly more stable learning compared to the REINFORCE agent. Although the moving average lines of both trading actions stay closer to 0, they seem to do so relatively stably in the later parts of the training process.

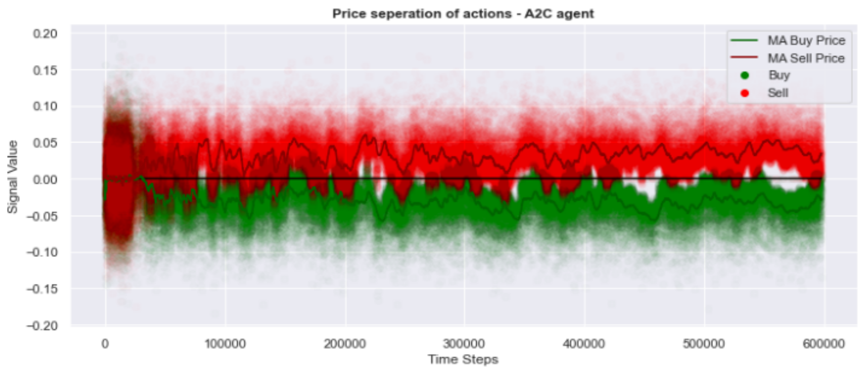

Figure 4.7: See figure 4.4 for further description. Agent parameters as described with $N = 40$

4.4 Proximal Policy Optimization Agent

4.4.1 Algorithm and Setup

First introduced by Schulman et al. [31] in 2017, the Proximal Policy Optimization Algorithm is now widely used in modern reinforcement learning applications [32, 33]. It was designed to find a trade-off between simplicity and sample efficiency and is part of the policy gradient model family. Being an Actor-Critic model the PPO algorithm also uses two neural networks to model the policy and the state–value function. The novelty of the model compared to the previous methods is the learning procedure and the calculation of the network losses.
Trying to make more efficient use of the already sampled data, the PPO algorithm does not optimise the policy once per sample as the REINFORCE or the A2C method would but instead performs several optimization runs on one sample trajectory. This makes the learning procedure quite different to other policy gradient models but improves the sample efficiency. Moreover, the gradient is obtained through a slightly different approach. We will outline its derivation and reasoning in the following paragraphs.
One main problem, that one faces when using, for example, the gradient in equation (4.3.3) for more than one learning step on the same sample, is that the gradient can be quite large and hence learning would be temporarily driven in a non-optimal direction. This gradient explosion arises from the fact that the gradient used in the A2C method is calculated on the policy space Π. This space includes all possible policies with respect to the environment. Furthermore, the only space we can directly interact with is our parameterisation of the policy space: the network weights and biases $\theta_1, \theta_2, \theta_3, ...$ in the space Θ. Consequently, the distance between two policies π_1 and π_2 is not necessarily the same as the distance between the two parameterizations θ_1 and θ_2 ([30], pages 166-167). Hence learning steps can become unnecessarily large or far too small when using (4.3.3) as the gradient for more learning iterations. This problem was already partly solved by Schulman et al. [34] and implemented in the *Trust Region Policy Optimization* algorithm by using a *surrogate objective function*. The idea behind this objective function is a monotonic improvement of the policy based on a relative measure. This relative measure is the probability ratio $r(\theta)$ defined as

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{4.4.1}$$

where we easily find that $r(\theta_{old}) = 1$. This function is then used to the define the surrogate objective function $J^{CPI}(\theta)$ as follows

$$J^{CPI}(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t^{\pi_{old}}\right] = \mathbb{E}[r(\theta)\, A_t] \tag{4.4.2}$$

where CPI stands for conservative policy iteration [35] and A_t is the advantage function as defined in equation (4.3.1). We omitted the π_{old} on the right-hand side of the equation since the calculation of A_t necessarily always happens on the old policy. It can be shown that the gradient of $J^{CPI}(\theta)$ with respect to the old policy is equal to the gradient of the objective function $\nabla J_\theta(\pi_\theta)$ as in equation (2.2.11) and that $J^{CPI}(\theta)$ satisfies the *monotonic improvement theory* which guarantees at least non-negative jumps in the objective function meaning $J(\pi') \geq J(\pi)$ where π' is the improved policy [36]. For the latter to hold, we need to guarantee one additional constraint which restricts the deviation of the new objective value in terms of distance from the old objective. This will be realised by limiting the expected Kullback-Leibler (KL) divergence of the policy probabilities:

$$\mathbb{E}_t[KL(\pi'(a_t|s_t)\,||\,\pi(a_t|s_t))] \leq \delta \tag{4.4.3}$$

This then results in the *trust region policy optimization problem*:

$$\max_\theta E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A_t\right] \text{while satisfying}$$
$$\mathbb{E}_t[KL(\pi'(a_t|s_t)\,||\,\pi(a_t|s_t))] \leq \delta$$

while there is a PPO algorithm version that directly implements an objective function based on the problem stated above. We will focus on a slightly different approach that does not require us to choose a threshold value δ but keeps the main idea of the KL divergence limit. This method is called PPO *with clipped surrogate objective* or simply *with clipping* and uses the following objective function:

$$J^{CLIP}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)A_t,\, clip(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)A_t\Big)\Big] \tag{4.4.4}$$

where the choice of ϵ allows us to control the difference between the old and the new policy more easily than in (4.4.3). Furthermore, this control also guarantees that we will have no significant drop in the quality of the current policy since we can always choose $\pi' = \pi$ and therefore have $J(\pi') = J(\pi)$. We will later investigate what influence the choice of ϵ in equation 4.4.4 has on the learning process.
The complete PPO algorithm with clipping is illustrated in pseudo code in algorithm 5.

Algorithm 5: Proximal Policy Optimization algorithm (N-Step)

```
1: Initialize weights $\theta_a$ and $\theta_c$ for actor and critic-network
2: Set learning rate $\alpha_a > 0$ for actor-network
3: Set learning rate $\alpha_c > 0$ for critic-network
4: Set discount rate $\gamma$
5: Set $N$, the number of episodes
6: Set $T$, sample sequence length
7: Set $B < T$, the minibatch size
8: for $i = 1, 2, ...$ do
9:     Set $\theta'_a = \theta_a$
10:    Sample and store data tuples $(s_t, a_t, r_t, s'_t)$ until $t = T$ using $\pi_{\theta'_a}$
11:    Calculate $\hat{v}(s_t)$ using the critic-network
12:    Calculate $A_{Nstep}(s_t)$ using the critic-network and the stored rewards
13:    $v_{tar}(s_t) = A_{Nstep}(s_t) + \hat{v}(s_t)$
14:    Let batch be the tuple $((s_t, a_t, r_t, s'_t), A_{Nstep}(s_t), v_{tar}(s_t), \hat{v}(s_t))$
15:    for episode $= 1, ..., N$ do
16:        for minibatch $m$ in batch do
17:            for observation $o$ in $m$ do
18:                Calculate $r_o(\theta_a)$
19:                Calculate $J_o^{CLIP}$ using the advantages $A_{Nstep}$ from o and the $r_o(\theta_a)$
20:            end for
21:            $J^{CLIP}(\theta_a) = \frac{1}{B}\sum_{o \in m} J_o^{CLIP}(\theta_a)$
22:            Set actor loss equal to clip value when using no entropy:
23:            $L_{actor}(\theta_a) = J^{CLIP}(\theta_a)$
24:            Calculate the critic loss using the value approximation and the value target from the
               observation:
25:            $L_{critic}(\theta_c) = \frac{1}{B}\sum_{o \in m}(v_{tar}(s_o) - \hat{v}(s_o))^2$
26:
27:            $\theta_c = \theta_c + \alpha_c L_{critic}(\theta_c)$
28:            $\theta_a = \theta_a + \alpha_a L_{actor}(\theta_a)$
29:        end for
30:    end for
31: end for
```

The loop starting at line 8 is mainly used for sampling the state-action sequences. After a trajectory of length T is sampled, minibatches of size B are randomly selected from the data and the objective function J^{CLIP} is calculated for all observations in the minibatch. Subsequently, the actor-network is updated with the average of the J^{CLIP} values and the critic-network is updated with the mean squared error of the difference between the approximated target state-value v_{tar} and the critic-network output $\hat{v}$.
According to the results from Schulman et al. [31] this algorithm provides better overall performance in a number of problems. It, furthermore, proves to be simpler and therefore also more straightforward to implement than methods with comparable good performance. As for the other models, we will run the algorithm on the OU process base setting and analyse its results.

4.4.2 Trading Simulation

Beginning again with the analysis of the resulting trading behaviour, we observe that the algorithm picks up many of the local minimum and maximum spikes for buying and selling actions, respectively. Moreover, we can not see a buy over a signal value of 0 and a sell under 0 for this frame of the learning process which also underlines the good learning of the algorithm. The right plot of figure 4.8 furthermore points out that the two distributions of the buy and sell signal values are separated and we can even detect a small gap around 0. The average number of trades executed by the agent in a 100 time step frame is 42 which is slightly higher compared to the agents before.

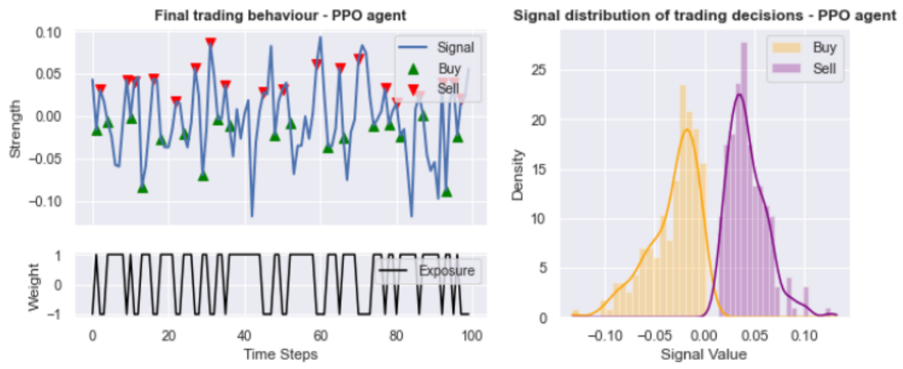

Figure 4.8: See figure 4.2 for further description. We used the following parameters for the training: $\alpha_a = \alpha_c = 0.001$, $\gamma = 0.95$, $N = 2000$, $T = 300$, $B = 5$, $\epsilon = 0.2$

Surprisingly the learning curve for the PPO agent using $\epsilon = 0.1$ in the left plot of figure 4.9 shows a drop in learning. Given that the PPO aims to satisfy the monotonic improvement theory, this is unexpected, but can happen through a sampling many unseen processes at once. However, despite this large drop the agent seems to recover quickly and finishes training with an average score of around 6.2. This is also the case for the agent using $\epsilon = 0.2$. The agents with ϵ values of 0.3 and 0.4 are performing worse with the advantage at the latter since the PPO agent with a clipping value of 0.3 does not seem to solve the problem. The plot on the right-hand side of the same figure shows us that the learning also seems to reduce the standard deviation of the score over the different episodes. Having a score standard deviation of 1.69 over the first 100 episodes, this value decreases to 0.76 for the last 100 episodes.

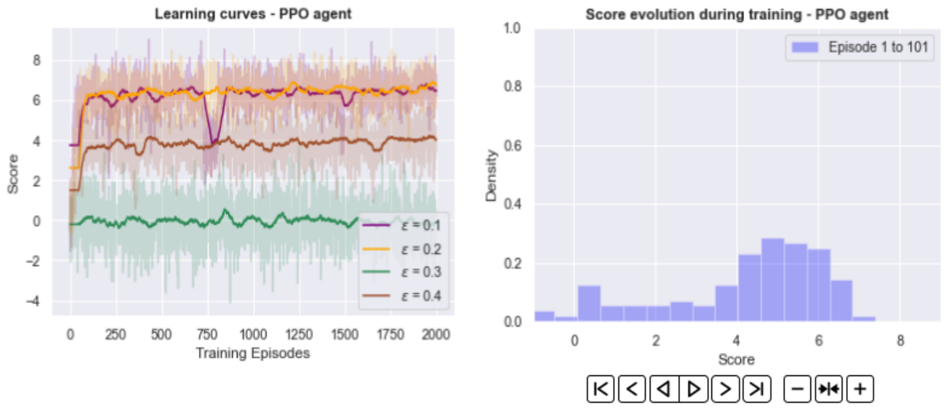

Figure 4.9: See figure 4.3 for further description. Agent parameters as figure before. The right plot uses $\epsilon = 0.2$

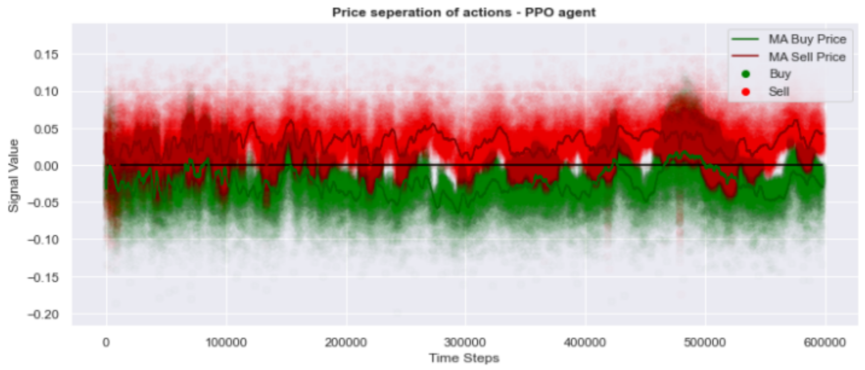

Figure 4.10: See figure 4.4 for further description. Agent parameters as figure before.

Interestingly figure 4.4 illustrates a possible disadvantage when using a PPO algorithm. Although the agent seems to solve the problem during the learning process, the average buy and sell price varies quite drastically. The agent appears to buy at a price as high as 0.1 and sell at a price as low as -0.1 with those trades being far from profitable. So far, the A2C algorithm seems to deliver the most stable results among the trading algorithms in the trading simulation environment.

4.5 Direct Comparison of the Agents

To better understand the performance of the different models in relation to each other, we test all models on the same signals and compare their theoretical profit and loss and their trading behaviour. We, therefore, assume that the signal represents a basket of assets that can be traded in the market. The left plot of figure 4.11 shows bespoke theoretical PnL. For the analysis, all models are run on their best configuration based on the results from the previous section. We find that the REINFORCE and the PPO algorithm show the best performance with a final score of around four. Up to almost half the trading time, no particular difference is observable between the two agents, only after that the PPO agent gains a slight advantage. The Advantage Actor-Critic model appears to outperform the LSTM as well but only reaches a final score of around 2.7 in this particular case. All machine-learning-based models outperform the Bollinger Bands with window size 20, distance factor 1.5 and a final score of 0.61.

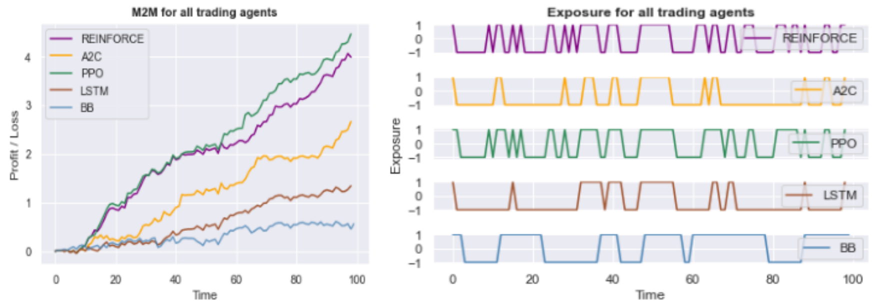

Figure 4.11: Left: Theoretical performance of all trading agents and Bollinger bands on the same OU process. The best configuration for each model is chosen for the analysis. Right: Position changes for all models for the given PnL.

While we can see some periods in the right plot of figure 4.11 where all agents seem to agree on the trades to make, we also observe some significant differences regarding trading frequency and timing. With 38 trades, the REINFORCE agent has highest number of executed buy and sell actions, closely followed by the PPO agent with 34 trades. This and the similarity of the timing of the actions explains the almost equal performance of the two agents on the signal. The fewest trades are done by the Bollinger Bands with only 10. The A2C trading agent and the LSTM model both have on average between 15 to 20 trades in a 100 time step period resulting in a mid-level performance compared to the other agents. Figure 4.12 elucidates the advantage of the A2C agent over the LSTM one and categorises its average performance short of the one from the REINFORCE and PPO agents. These two models deliver the best results regarding the average score of 4.30 and 4.05 for REINFORCE and PPO, respectively. Despite the lower score for the LSTM agent, it is worth mentioning that the score is achieved with the lowest standard deviation of all models. In comparison the three reinforcement learning agents result in a score standard deviation between 0.78 and 0.80, while the LSTM model delivers only 0.41. Analysing the right plot of 4.12 we find that at most extreme points, the agents seem to agree on the trade but also that the REINFORCE, A2C and PPO are able to pick less obvious trading opportunities in the signal.

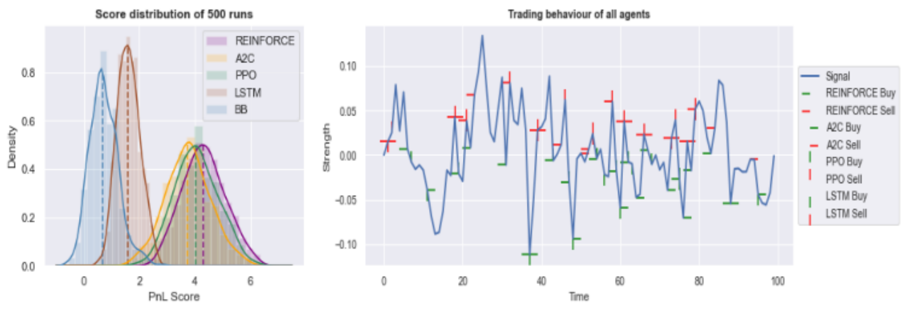

Figure 4.12: Left: Score distribution over 500 runs of the models on an OU Process with the dashed line indicating the median for each distribution. Left: Trading behaviour of all agents on the same signal, Left pointing: REINFORCE, Right: A2C, Upwards: PPO and Downwards: LSTM.

Table 4.4 summarises our findings. As seen in the 4.12 the REINFORCE agent delivers the best performance with a theoretical Sharpe Ratio of 5.4. Closely behind that are the two other reinforcement learning models, with 5.09 for the PPO agent and 4.74 for the A2C agent. Fisher's information criterion gives an estimate of how strongly buy and sell prices are separated by the agents' decisions. Figures 4.5 and 4.8 already suggest what is verified by the metrics. PPO separates the actions stronger than every other model, leading to a good performance with fewer trades. Furthermore, we find that the Fisher's information value is also relatively high for the Bollinger Bands trading. This can be traced back to the fact that the BBs do not try to make profitable trades close to the long term mean of the signal but rather execute their trades only on extreme signal values. Hence also the trading rate decreases, resulting in a poor performance. Closely related to the FIC, we find that the average buy and sell prices are most extreme for the PPO (sell) and the A2C (buy) agents. With 37.9 trades on average the REINFORCE agent executes by far the most trades among all agents. The PPO agent acts only 27.9 times on average, which makes it the next closest, followed by the A2C agent with 25.2 trades. The other extreme is found with the BB strategy with only 8.4 trades on average.
Overall we conclude that all machine learning algorithms provide a significant improvement over the Bollinger Bands trading behaviour. Furthermore, we find that all reinforcement learning agents namely REINFORCE, Advantage Actor-Critic and Proximal Policy Optimization outperform the pure machine learning LSTM model. The performance difference between the RL models themselves is only narrow, with the simple REINFORCE algorithm showing a slight advantage over the other two algorithms. PPO seems to provide insignificant better performance than A2C, making A2C the 'worst' of the RL trading agents.

Metric	**REINFORCE**	**A2C**	**PPO**	**LSTM**	**BB**
Avg. Score	**4.28**	3.78	4.05	1.64	0.78
Std. Dev.	0.79	0.80	0.80	**0.42**	0.49
Sharpe Ratio	**5.40**	4.74	5.09	3.94	1.57
FIC	0.034	0.071	**0.095**	0.052	0.086
Avg. Buy/Sell Price	-0.017 / 0.026	**-0.031** / 0.028	-0.026 / **0.040**	-0.030 / 0.028	-0.030 / 0.023
Avg. Turnover	**37.9**	25.2	27.9	20.6	8.4

Table 4.4: Result summary of the trading agents run on 500 OU processes. Fisher's information criterion (FIC) is calculated on the distributions of buy price and sell price.

The following chapter will take the results and models and test them in a historical market environment to investigate the usefulness of these algorithms in actual trading applications. Moreover, we will put our focus on the reinforcement learning agents and refrain from further analysis of the LSTM model.

Chapter 5

Market Data Trading Analysis

This section concludes the analysis of the trading agents with their application on the previously introduced PCA signal. Some of the models are pre-trained on a simulated signal in order to boost performance on the historical intraday data. The simulated signal is generated to replicate the PCA signal as closely as possible.

5.1 Methodology and Setup

As a last investigation step, we will test the trading agents on the historical intraday data introduced in section 1.4 of this work. The PCA signal, being based on historical data, is slightly noisier than the simulated signals. This makes training for the agents a bit more challenging. Hence we will start by first fitting them to a simulated signal which resembles the PCA one, before finally training the agent on the actual signal. To make the transition from simulation to historical data as smooth as possible we will fit an OU process on the PCA signal using the method provided by Avellanda and Lee ([10], pages 21-22 (781-782)). We will refrain from outlining the detailed mathematics but rather provide an intuition for the main idea. As mentioned before, a signal such as the PCA signal can be seen as a discrete version of an OU process. We, therefore, can estimate the value of the process at time $t+1$ with respect to the value at time t using the linear regression:

$$x_{t+1} = \alpha + \beta x_t + \epsilon_{t+1} \tag{5.1.1}$$

The results from the fitting of this linear regression can then be used to match the parameters α and β to the parameters of the solution of the stochastic differential equation provided in equation (1.2.2). We should then obtain an OU process with similar characteristics as the original signal. Running the fitting process as outlined, we obtain a parameter estimation for θ, m and σ (see equation (1.2.1)) at every point in time. Plotting the evolution of mean-reversion speed θ and the instantaneous volatility σ, we find that the process shows some periodicity. Figure 5.1 shows aforementioned evolution of the parameters.
Both parameters show an increase in value at two specific times during the day. Disregarding the spikes at market close, which are caused by slight irregularities in the data, at official market opening and market closing they seem to reach their daily maximums. We, therefore, conclude that the signal depends on the trading volume since these are the times when intraday trading volume is usually the highest. Hence, it is reasonable to include indicators of market volume in both the search for a trading signal and the actual trading process. We will, for now, refrain from such inclusions and focus on the information provided by the signal itself. The analysis shows that the value for θ varies from 0 to 10 with the main chunk of the values staying in the interval of [0, 5]. Hence we decide to pre-train the reinforcement learning trading agents on an OU process with $\theta = 1.94$, which is the mean of the values over time. We will proceed in a similar manner for σ, which has an average value of 0.0195. The results for the mean showed quite stable values with an average mean of 0.12. Nevertheless we will sample the OU process with a mean of 0 since every mean-reverting signal can just be shifted to have a zero mean.

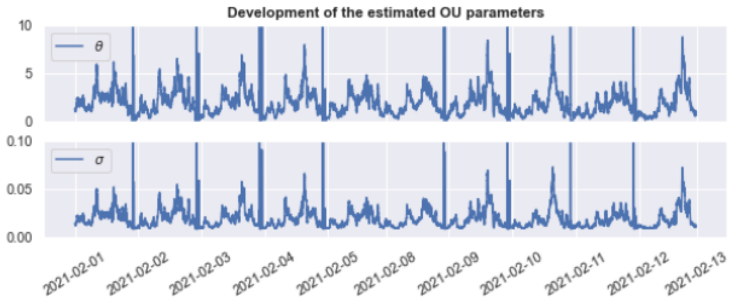

Figure 5.1: Plot showing the development of the two parameters θ and σ over the trading period on the data introduced earlier. The top plot is showing θ and bottom one σ.

The idea of pre-training the model is that the neural networks mainly responsible for the learning of the agent are already accustomed to the state space, that they will face when using the historical data. Furthermore, some of the actions will already be relatively close to the desired ones and the time spent on the learning process on the more noisy historical data can be reduced. We, therefore, expect to see an increase in learning speed and model performance.
Since the comparison on the simulated mean-reverting process in chapter 4 already showed the advantages of the reinforcement learning agents will we focus on these three agents in this analysis. The exact calibration of the used models can be extracted from table 5.1 with the architecture of the neural networks staying the same as introduced in chapter 4.

Parameter	**Description**	**REINFORCE**	**A2C**	**PPO**
T	Sequence length	1000	1000	1000
$\alpha_a = \alpha_c$	Learning rate	0.00025	0.00025	0.00025
γ	Discount rate	0.9	0.9	0.9
N_{step}	Adv. esti. steps	-	20	30
N_{eps}	Mini episodes	-	-	5
B	Minibatch size	-	-	5
ϵ	Clipping threshold	-	-	0.2

Table 5.1: Table showing the model calibration parameters for all reinforcement learning agents (REINFORCE, A2C, PPO) fitted on the PCA signal.

Before training the models on the simulated data, we make a few adjustments to the reward function. Firstly, we split the reward for the different actions such that for the action where the agents rest, a constant reward of -0.0001 will be assigned to the action. Secondly, the reward for an active action (flipping the exposure) will be calculated as before using the mark-to-market values only now we are additionally subtracting a 'trading cost' value of 0.001. This is on purpose chosen above (in absolute terms) the non-trading penalty of -0.0001 to avoid that the agent cheats its way around this penalty in times where the PCA signal holds a constant value for a longer period (see figure B.1). Cheating in this case meaning that the agent could, for example, open and close trades constantly and since being active, not experiencing the non-trading penalty while also not making any losses (since the signal was assumed to be constant). Adding the trading costs makes this more costly than doing nothing and the agent will instead take the non-trading penalty.
We run all models for a maximum of 10000 episodes on the simulated OU process, terminating the learning process if no improvement is made within 1000 episodes after the initial learning jump. This gives the models a maximum of 10^7 different states to learn from in the pre-training phase. Subsequently, we will train the models on parts of the PCA signal using the same procedure. Finally, we will compare the training and performance results with the agents immediately trained on the PCA signal.

5.2 Training and Comparison

We will train and test three models for each agent category on the simulated and, respectively, historical data. We will refer to the first agent group as the PRE agents, being the ones only fitted on the simulated data and consequently, we will call the agents using the pre-fitted model and being trained on the actual signal the POST agents. Finally, the agents directly fitted on the PCA signal will be referred to as RAW agents. This will result in nine agents three for each of REINFORCE, A2C and PPO method.
Figures 5.2 and 5.3 show the results of the learning process for the REINFORCE PRE and POST agents. We observe that after around 6000 training episodes, the score seems to reach an upper boundary on both the pure score development as well as the score divided by turnover line for the PRE agent. Both plots seem to support a good learning performance and the model also stabilizes its trading frequency with an average turnover of around 440 trades in one trading interval. In an interval of only 1000 time steps, this seems high, but the behaviour can be explained by the characteristics of the simulated signal, which offers quite a lot of trading opportunities due to its large and frequent jumps. Hence, we expect the turnover value to drop by a significant amount when fitting the model on the actual signal, which does not offer as many opportunities. As shown by the bottom row of plots, we find that the pre-fitting served as a good model preparation for the actual data. From almost the start of the fitting process, we see a relatively high and almost constant mean score. Only towards the end of the 5000 episodes, we see some slight but still insignificant improvements. As expected, the turnover value drops to almost half its value to around 250. With the overall score balancing in at slightly above six, we see some enhancements in the ratio of score and turnover. We conclude that the pre-fitting of the model prepared the neural network very well for the actual signal and only little training is need to adjust the trading algorithm to the new environment.

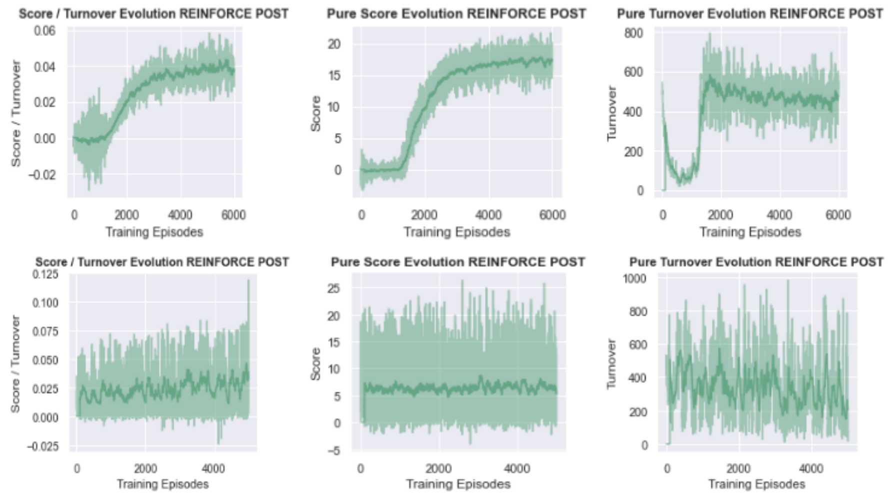

Figure 5.2: All top plots for the REINFORCE PRE trading agent and all bottom plots for the REINFORCEMENT POST agent. The solid line shows a 100 step moving average whereas the more transparent line shows the actual values. Left: Development of the return score divided by the number of trades in the respective trading period. Middle: Stand alone score evolution of the agent. Right: Stand alone turnover development over the training period.

When comparing the range of the actual values from the top row to the bottom row, we find that the value range for the POST model is a lot larger than for the PRE models. This observation can be traced back to the quite irregular PCA signal, which can show long periods of few trading opportunities but also periods of very profitable behaviour. This range of the values also depends on the sequence length T, resulting in a more constant score development for higher T. Moreover, the trading behaviour of the REINFORCE RAW agent as shown by the third row of Figure B.8

and also the final PnL (figure 5.4 illustrates the inability of the model to learn from the data from scratch.
We conduct further analysis of what the agent actually learned but investigating the action probabilities of the policy modelled by the actor of the REINFORCE POST agent. Reminding ourselves that the state consists of 3 inputs namely the current signal value, the current exposure and the signal value of the last trade (price) we are going to fix the exposure and analyse $\pi_\theta(a = 1|s = (s_1, s_2 \in \{-1, 1\}, s_3))$ with $s_1, s_3 \in [-0.25, 0.25]$.

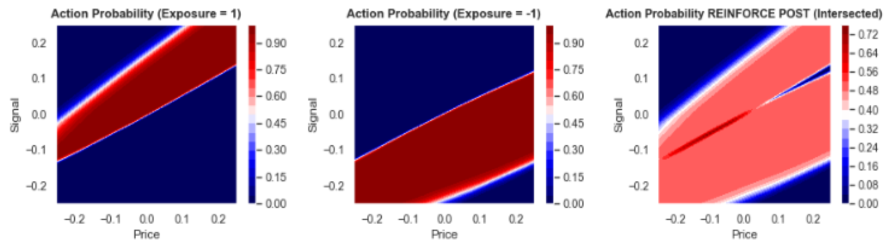

Figure 5.3: Left: Plot showing the action probability for flipping the exposure for the REINFORCE POST agent. The x-axis shows the current signal value (first state dimension) and the y-axis shows the price at which the agent last acted (third state dimension). The current exposure (second state dimension) is assumed to be 1. Middle: Same as left for exposure assumed to be -1. Right: Sum of the to surfaces to the left divided by 2, building the average probability of a trade action independent of the exposure.

The left plot of figure 5.3 shows the action probabilities for the exposure fixed at 1. Hence the x-axis can be seen as the price (or signal value) at which the agent bought the last time. Looking at the red strip, which indicates high selling probabilities, we can see the agent learns where to best sell the asset again and to avoid selling for a loss. The lower boundary of the red strip shows a clear cut and almost aligns with the diagonal (signal = price). Only in the top right corner this boundary crosses the diagonal, indicating that more training and maybe more exploration would be necessary to learn these parts of the state space. The upper edge of the red strip does not show such a clear cut but rather a transition space. Nevertheless, it is surprising that the agent does produce the blue area in the top left corner since those are the most profitable states with the lowest buying and highest selling price. We observe the same behaviour for the probabilities of states with a fixed exposure of -1 that are illustrated by the middle plot of figure 5.3. This time on the lower right corner, we again find a blue triangle indicating lower action probabilities where we would expect the agent to buy. Additionally, the change towards this triangle from the red strip is not clear cut and we again conclude that the agent still is unsure what to do best.
A further important observation is that the red area shown by the middle plot is unmistakably more extensive than the one in the left plot. This shows a clear learning step in the right direction and once more indicates that more exploration or more data is needed to fully fit the agent. Although it is possible that more exploration would be sufficient to learn the action probabilities for the mentioned area, this is potentially not improving the agent as elaborated in the following. Given the mean-reverting PCA signal, extreme values such as numbers above 0.15 and below -0.15 are less probable and hence such trade opportunities rarely arise. This makes waiting for the rare, highly profitable trades less rewarding than trading for a smaller profit but more often. The REINFORCE POST agent picks up this behaviour, resulting in the execution of trades as soon as they cover the trading costs and provide acceptable profit.
The final plot of figure 5.3 shows the average probabilities combining the two exposure values. Noticeable is the irregular colour in the intersection of the two red areas, which again underlines the asymmetry of the two previous plots. Here one would probably expect a more explicit separation of the two intersected red areas, but as mentioned before, the agent attempts to pick up smaller profits and therefore sells at a relatively lower and buys at relatively higher value. This results in the intersection observed in the right plot.
Table 5.2 shows that the REINFORCE POST agent outperforms its PRE and RAW counterparts. This is also because after trying a range of hyperparameters for the RAW agent, the model does not learn the right actions on the PCA signal. This underlines the usefulness of pre-fitting the

model on a simulated signal where as much data as necessary is available and the data is less noisy. Moreover, the A2C agents as well stress the advantages of the pre-fitting procedure. Although the A2C RAW agent picks up on the learning quite well, it is still outperformed by the A2C POST agent, which also trades less as shown by the turnover. Despite the only slight PnL difference of 48.79USD the A2C POST agents shows a more stable trading behaviour with a PnL over turnover value of 0.0007 and a profitable trade ratio of 0.53. The best performing agents in terms of PnL are the PPO RAW and the REINFORCE POST with the former showing an almost perfect positive trade ratio of 0.99. Given that the PnL is also slightly better than the one of REINFORCE POST agent, this is a very promising result and leads to a PnL-Turnover ratio that is by far better than for any other agent.

Model	**REINFORCE**			**A2C**		
	PRE	**POST**	**RAW**	**PRE**	**POST**	**RAW**
PnL	937.43	1'148.37	-1.16	356.88	670.81	621.02
Sharpe	3.59	3.56	-1.21	3.25	3.55	3.55
Turnover	504'337	360315'	**1'355'148**	262'874	899'559	1'149'431
P. Trades	0.64	0.72	0.48	0.61	0.53	0.51
PnL/SDR	78.48	123.23	-0.42	54.36	75.91	83.77
PnL/T.	0.0019	0.0032	0.0	0.0014	0.0007	0.0005

Model	**PPO**		
	PRE	**POST**	**RAW**
PnL	586.87	949.30	**1'178.25**
Sharpe	3.58	**3.95**	3.54
Turnover	1'118'817	267'601	147'138
P. Trades	0.52	0.74	**0.99**
PnL/SDR	66.10	124.07	**135.58**
PnL/T.	0.0005	0.0035	**0.0080**

Table 5.2: Summary of the main results of the application of all algorithms on the whole PCA signal. PnL is given in USD, Turnover represents the number of trades made during the process, and positive trades are the ratio of profitable trades to all trades.

The PnL shown in figure 5.4 illustrates the trading differences of the models and we can see that the performance of the two best agents is indeed quite similar in terms of the PnL development. It is conspicuous, though that the PPO POST agent seems to stop trading during the middle sector of the two-week period since it appears to keep up with the best-performing agents up to this point. Further investigation showed that the PCA signal seems to show a shifted value range during this period. Therefore, we expect that more exploration in this area would improve the agent. Reminding ourselves of the performance given by the different Bollinger Bands (see figure 4.11), we find that almost all agents outperform the best Bollinger Band result of 428.39. Only the REINFORCE RAW and the A2C PRE agent do not show a PnL above this value.
We have shown that using reinforcement learning, it is possible to outperform the standard approach of Bollinger Bands trading. With the PnL increasing by a factor of almost three the best performing agents show very satisfying results and also point out that Bollinger Bands miss many trading opportunities. They furthermore managed to outperform the more traditional machine learning approach using a LSTM neural network to trade. Moreover, it proves helpful when training the agents to trade a mean-reverting signal to start the fitting process on a simulated signal. Subsequently, the pre-trained model can then be introduced to the actual signal based on historical data. In the cases of the REINFORCE and the A2C agent, this process improved the performance compared to fitting the model directly on the actual data. It also resulted in a more robust trading behaviour. In general, with the exception of the REINFORCE RAW agent, PPO models, as well as REINFORCE models, seem to be more suitable for the problem of mean-reversion trading than A2C methods.

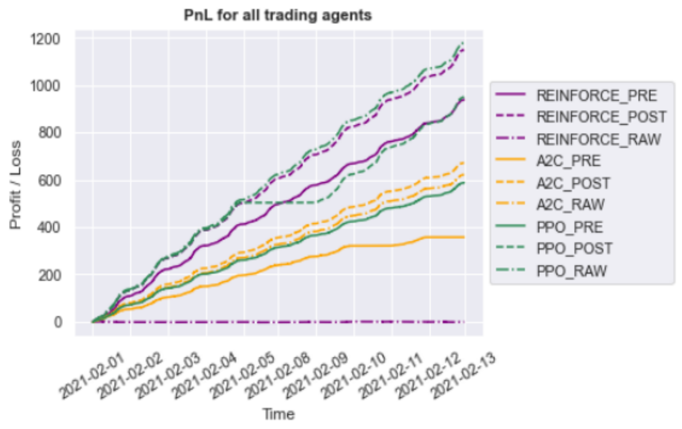

Figure 5.4: PnL in of all nine reinforcement learning trading agents on the PCA signal with the following agent separation: Purple: REINFORCE, Yellow: A2C, Green: PPO and Solid line: PRE, Dashed only: POST, Dashed/Dotted: RAW. PnL is in indicated USD.

Conclusion

We introduced the potential of machine learning algorithms with a short example of trading signal extraction. Using neural networks and customised loss functions, we showed that it is possible to extract signals with mean-reverting properties from historical futures data. Although simple methods like including the variance ratio in the loss function could not outperform the classical methods such as the ones introduced by Avellaneda and Lee [10] on a relative basis, they showed usefulness in the extraction process.

With the latest advancements and an increased overall use, reinforcement learning is on the brink of establishing a fixed place among models in the financial industry. In the main part of this work we demonstrate its huge potential and motivate further investigation into the area of decision making in trading. We focused on the application of Policy-based methods particularly the REINFORCE algorithm and two Actor-Critic models namely Advantage Actor-Critic and Proximal Policy Optimization. Linking the investigation to the previous discussion of mean-reverting signals we started with training the models on a simulated Ornstein-Uhlenbeck process. The state space for all models included the current signal value, the current exposure and the signal value at the last buy or sell action. Using a loss function to maximise the PnL over a given time horizon, the agents show a relatively good performance. All three trading algorithms can outperform the classical Bollinger Bands and a more traditional LSTM network trained on the same problem. Moreover, we find that the REINFORCE and the PPO models have slight advantage over the A2C model on the OU process.

As the final step we attempted to improve a trading strategy motivated by Avellaneda and Lee [10], which is based on principal components. We started by fitting the parameters of an Ornstein-Uhlenbeck process to the historical data signal and training the agents on a simulated signal using these parameters. As it turns out, for the two trading agents REINFORCE and A2C this procedure increased the performance when later fitting the agents on the actual signal. We found that all models but the REINFORCE RAW are outperforming the trading suggested by classical Bollinger Bands. The two best performing agents, REINFORCE POST and PPO RAW, can increase the PnL by almost 800USD during the two-week testing period which comes down to a 200% increase over the best Bollinger Bands.

The positive results provided by the reinforcement learning agents motivate further analysis of their application possibilities. As a further step one could skip the signal generation part and train the agents directly on the raw asset returns. Combined with a suitable loss function those agents could then be trained to generate a risk adjusted positive return and it will be interesting to see if they replicate one of the know trading strategies such as momentum or mean-reversion trading.

Appendix A

Explanatory Calculations and Definitions

Hadamard product: Given two matrices A and B of the same dimension $(m \times n)$ the Hadamard product is a matrix of the same dimensions obtained by element-wise multiplication of the two matrices:

$$A \odot B = (a_{ij}b_{ij}) = \begin{pmatrix} a_{11}b_{11} & \cdots & a_{1n}b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & \cdots & a_{mn}b_{mn} \end{pmatrix} \in R^{m \times n} \tag{A.0.1}$$

Derivation of $\nabla_\theta J(\pi_\theta(x))$ ([30], page 28):

$$\begin{aligned} \nabla_\theta J(\pi_\theta(x)) &= \nabla_\theta \mathbb{E}_{\pi_\theta(x)}[G_T^{\pi_\theta}] \\ &= \nabla_\theta \int dx\, G_T^{\pi_\theta} \pi_\theta(x) \\ &= \int dx\, \nabla_\theta (\pi_\theta(x) G_T^{\pi_\theta}) \\ &= \int dx\, (G_T^{\pi_\theta} \nabla_\theta \pi_\theta(x) + \pi_\theta(x) \nabla_\theta G_T^{\pi_\theta}) \\ &= \int dx\, G_T^{\pi_\theta} \nabla_\theta \pi_\theta(x) \\ &= \int dx\, G_T^{\pi_\theta} \pi_\theta(x) \frac{\nabla_\theta \pi_\theta(x)}{\pi_\theta(x)} \\ &= \int dx\, G_T^{\pi_\theta} \pi_\theta(x) \nabla_\theta \log \pi_\theta(x) \\ &= \mathbb{E}[G_T^{\pi_\theta} \nabla_\theta \log \pi_\theta(x)] \end{aligned} \tag{A.0.2}$$

Appendix B

Complementary Figures

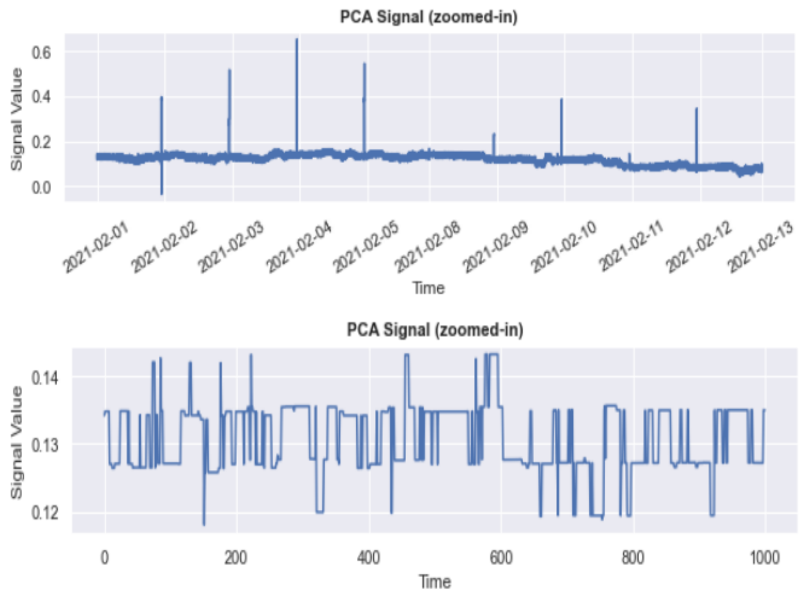

Figure B.1: Top: Plot of the raw PCA signal as mentioned in section 3.1. (Spikes in signal are due to day changes and are excluded in the performance analysis. Bottom: Zoomed-in version of the first 1000 time steps.

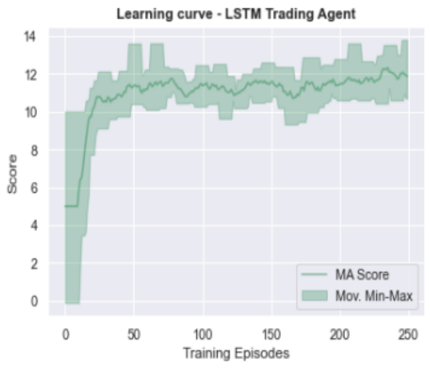

Figure B.2: Learning curve of the LSTM Trading agent. Score is the defined Sharpe ratio.

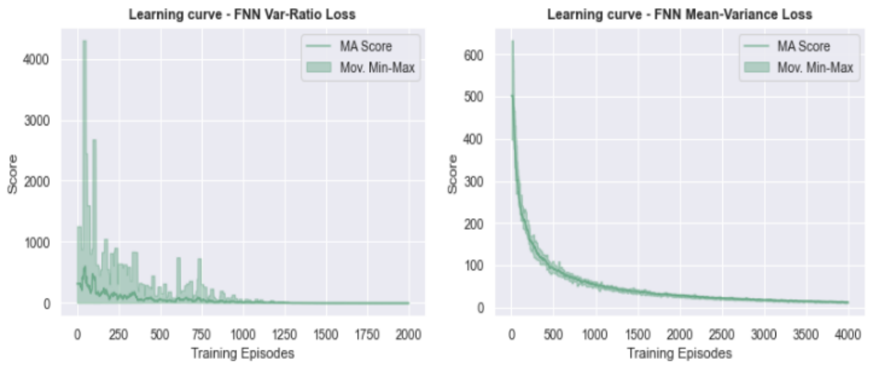

Figure B.3: Left: Learning curve of the linear FNN with variance-ratio loss. Right: Same model with mean-variance loss

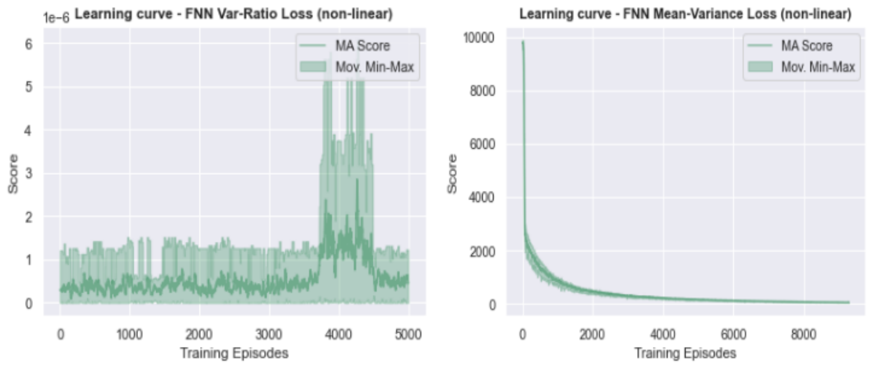

Figure B.4: Left: Learning curve of the non-linear FNN with variance-ratio loss. Right: Same model with mean-variance loss

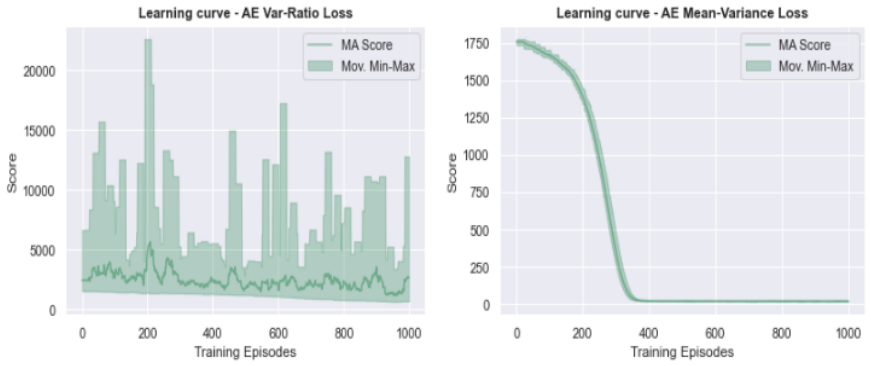

Figure B.5: Left: Learning curve of the linear AE with variance-ratio loss. Right: Same model with mean-variance loss

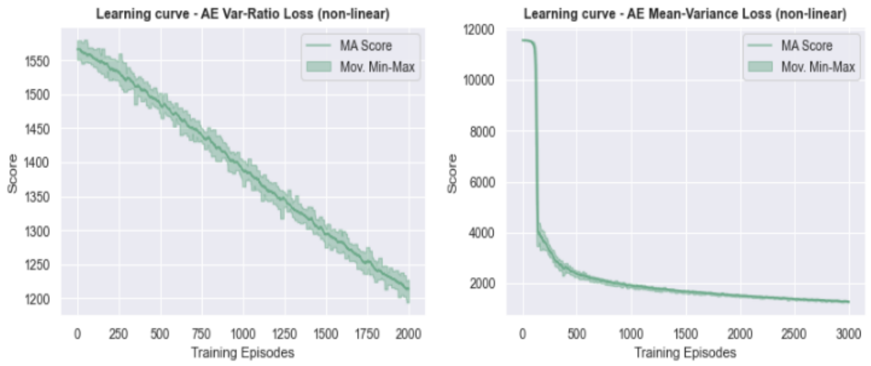

Figure B.6: Left: Learning curve of the non-linear AE with variance-ratio loss. Right: Same model with mean-variance loss

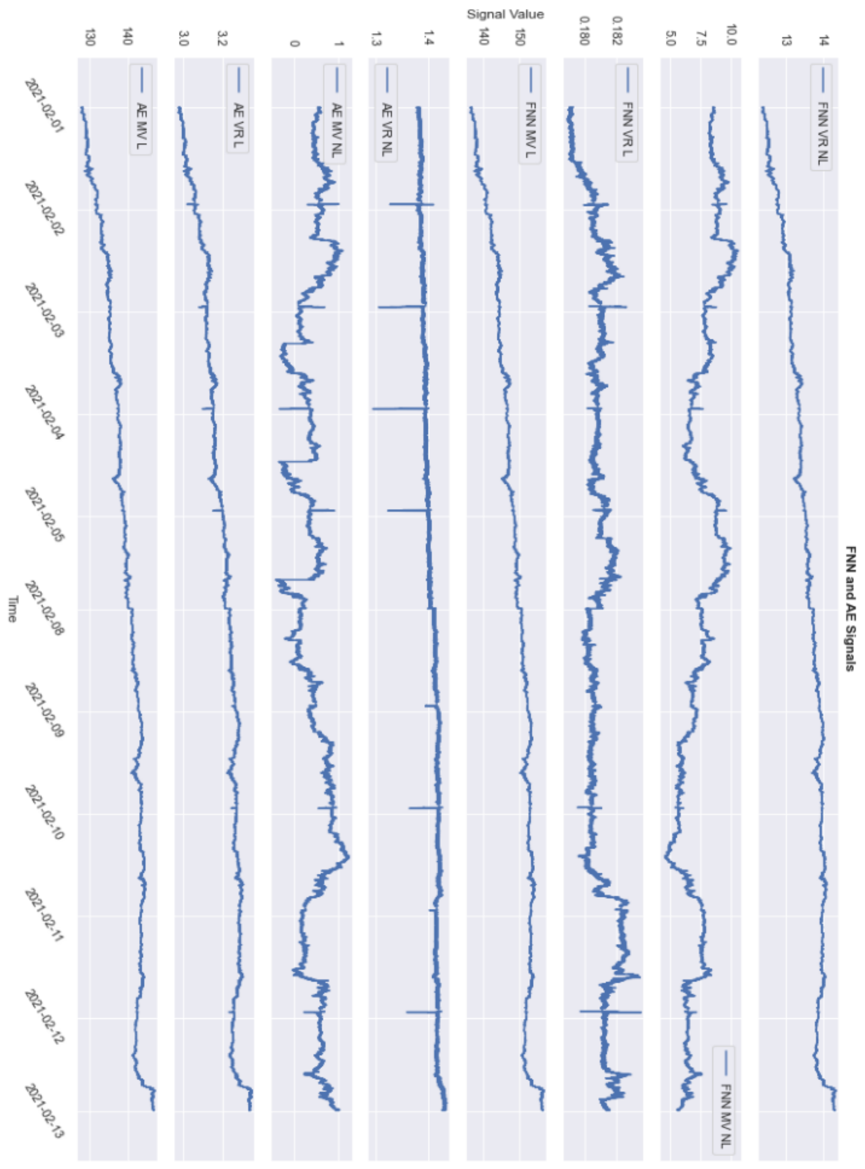

Figure B.7: All raw signals generated with the FNN and the AE.

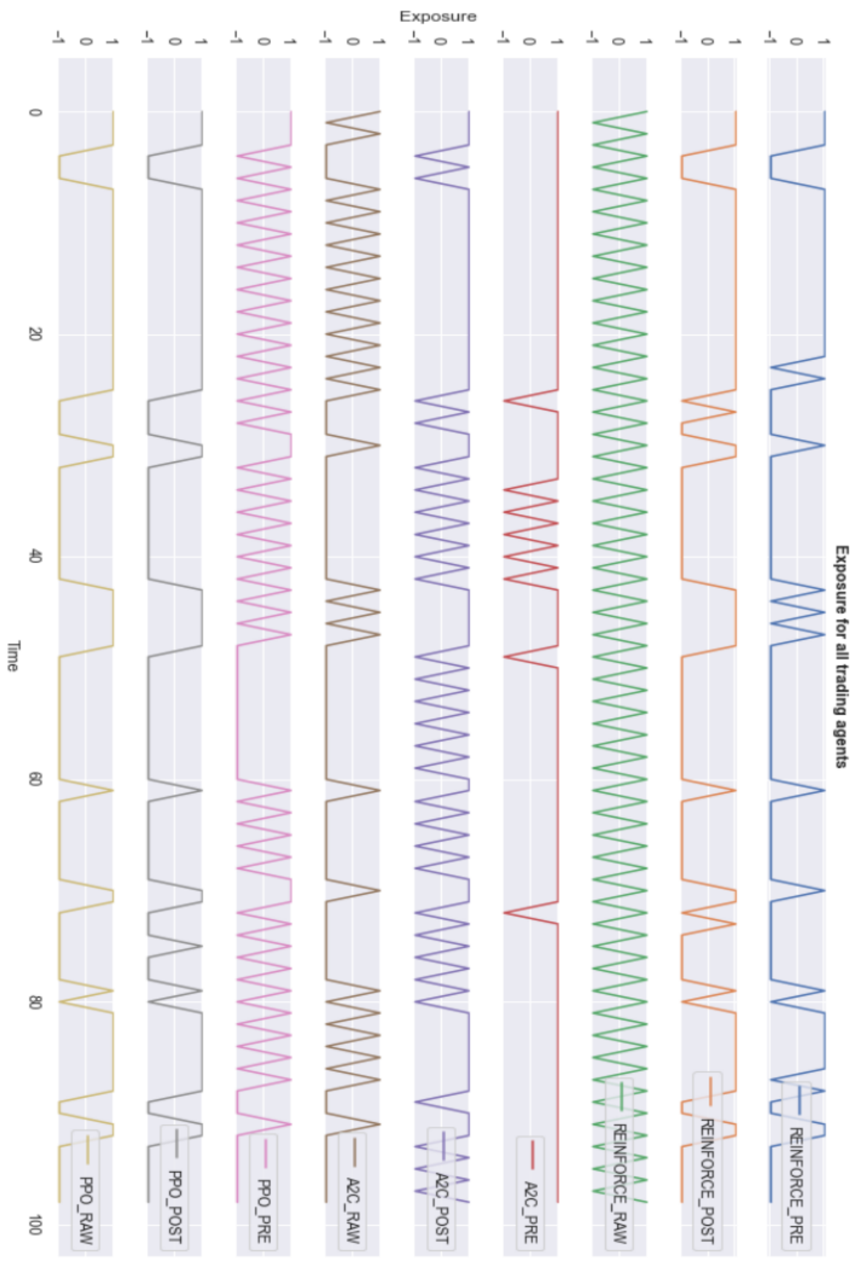

Figure B.8: Trading exposure for all RL agents on a sample from the PCA signal

Bibliography

[1] Sandro Ephrem Tarek Nassar. Mean reversion: A new approach. *SSRN: https://ssrn.com/abstract=3682487 or http://dx.doi.org/10.2139/ssrn.3682487*, 2020.

[2] Joao R. de M. Palotti Wagner Meira Jr. Leonardo C. Martinez, Diego N. da Hora and Gisele L. Pappa. From an artificial neural network to a stock market day-trading system: A case study on the bmf bovespa. 2009.

[3] Sample architecture of a neural network. *Link: https://towardsdatascience.com/step-by-step-guide-to-building-your-own-neural-network-from-scratch-df64b1c5ab6e*, Last access: 05.09.2021.

[4] Unfolding of an rnn cell. *Link: https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0*, Last access: 05.09.2021.

[5] Detailed view of an lstm cell. *Link: https://towardsdatascience.com/grus-and-lstm-s-741709a9b9b1*, Last access: 05.09.2021.

[6] General autoencoder architecture with 7 layers. *Link: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798*, Last access: 05.09.2021.

[7] Chulwoo Han, Zhaodong He, and Alenson Jun Wei Toh. Pairs trading via unsupervised learning. *https://ssrn.com/abstract=3835692 or http://dx.doi.org/10.2139/ssrn.3835692*, 2021.

[8] John van der Hoek Robert J. Elliott and William P. Malcolm. Pairs trading. *Quantitative Finance*, 5, 3:271–276, 2005.

[9] Christopher Krauss. Statistical arbitrage pairs trading strategies: Review and outlook, iwqw discussion papers. *ECONSTOR*, 9, 2015.

[10] M. Avellaneda and J.-H. Lee. Statistical arbitrage in the us equities market. *Quantitative Finance*, 10:7:761–782.

[11] Simão Moraes Sarmento and Nuno Horta. Enhancing a pairs trading strategy with the application of machine learning. *ScienceDirekt, 2020 Elsevier Ltd.*, 158, 2020.

[12] Qianwen Xu Victor Chang, Xiaowen Man and Ching-Hsien Hsu. Pairs trading on different portfolios based on machine learning. *Expert Systems, Wiley*, 2020.

[13] Karen Simonyan Ioannis Antonoglou Aja Huang Arthur Guez Thomas Hubert Lucas Baker Matthew Lai Adrian Bolton Yutian Chen Timothy Lillicrap Fan Hui Laurent Sifre George van den Driessche Thore Graepel Demis Hassabis David Silver, Julian Schrittwieser. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[14] Brooke Chan Vicki Cheung Przemysław "Psyho" Debiak Christy Dennison David Farhi Quirin Fischer Shariq Hashme Chris Hesse Rafal Józefowicz Scott Gray Catherine Olsson Jakub Pachocki Michael Petrov Henrique Pondé de Oliveira Pinto Jonathan Raiman Tim Salimans Jeremy Schlatter Jonas Schneider Szymon Sidor Ilya Sutskever Jie Tang Filip Wolski Susan Zhang Christopher Berner, Greg Brockman. Dota 2 with large scale deep reinforcement learning. *OpenAI Publication.*

[15] Evans R. Pritzel A. et al. Jumper, J. Highly accurate protein structure prediction with alphafold. *Nature, https://doi.org/10.1038/s41586-021-03819-2*, 2021.

[16] Daniele Tantari Alessio Brini. Trading mean-reversion with reinforcement learning. *Scuola Normale Superiore*, 2020.

[17] Andrew Brim. Deep reinforcement learning pairs trading. *Utah State University, https://digitalcommons.usu.edu/gradreports/1425*, 2019.

[18] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[19] Lili Tang. An actor-critic-based portfolio investment method inspired by benefit-risk optimization. *Journal of Algorithms Computational Technology*, 12(4):351–360, 2018.

[20] Siyu Lin and Peter A. Beling. An end-to-end optimal trade execution framework based on proximal policy optimization. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*, 2020.

[21] Kais Hamza Binh Do, Robert Faff. A new approach to modeling and estimation for pairs trading. *Monash University*, 2006.

[22] Juan Evangelista Trinidad-Segovia José Pedro Ramos-Requena and Miguel Ángel Sánchez-Granero. Some notes on the formation of a pair in pairs trading. *Sigma mathematics, MDPI*, 2020.

[23] J. Bollinger. *https://www.bollingerbands.com/*. Last access: 11.07.2021.

[24] Warren McCulloch und Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[25] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization. *Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego*, 2015.

[26] A. Lo and A. MacKinlay. Stock market prices do not follow random walks: Evidence from a simple specification test. *The Review of Financial Studies*, 1:41–66, 1988.

[27] Joongyeub Yeo George Papanicolaou. Risk control of mean-reversion time in statistical arbitrage. *Standford Mathematics*, 2017.

[28] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[29] D. A. Dickey and W. A. Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74:427–431.

[30] L. Graesser and W. L. Keng. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison Wesley Data Analytics Series, 2020.

[31] J. Schulman et al. Proximal policy optimization algorithms. 2017.

[32] Joren Gijsbrechts Nathalie Vanvuchelen and Robert Boute. Use of proximal policy optimization for the joint replenishment problem. *Computers in Industry*, 119, 2020.

[33] Prafulla Dhariwal Alec Radford John Schulman, Filip Wolski and Oleg Klimov. Proximal policy optimization algorithms. *OpenAI*, 2017.

[34] Philipp Moritz Michael I. Jordan John Schulman, Sergey Levine and Pieter Abbeel. Trust region policy optimization. 2017.

[35] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. 2002.

[36] Joshua Achiam. Advanced policy gradient methods. *UC Berkeley, OpenAI*, 2017.

ZIMMERMANN_HENDRIK_01950260

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57

PAGE 58

PAGE 59

PAGE 60

PAGE 61

PAGE 62

PAGE 63