Deep Reinforcement Learning for Dynamic Asset Allocation

An analysis of the use of deep reinforcement learning for solving the optimal portfolio policy problem

Frederik Spedtsberg (S127454), Oliver Bondrop (S128139) & Oliver Hellum (S128095)

Supervisor: **Anders Rønn-Nielsen**

Number of pages: 63

Number of characters: 110.738

Department of Statistics

Copenhagen Business School

Study Programme: Bachelor in Business Administration and Mathematical Business Economics

Date: 25/05/2021

Acknowledgements

We want to express our gratitude for our supervisor Anders Rønn-Nielsen for providing great input, guidance and academic support during this project.

Abstract

Estimating the expected returns and covariance matrix used to determine the optimal portfolio policy with Markowitz Modern Portfolio Theory is problematic in practice. This project investigates how to overcome this hurdle by directly estimating the optimal portfolio policy. This is achieved with deep reinforcement learning that relies on different deep neural network architectures. Using a simulation study we test the efficacy of the proposed deep reinforcement learning models to learn the underlying structure of the data and hence learn polices to determine asset allocations. The results show that all of the proposed models are capable of outperforming the equal weighted portfolio on the test data. However, further tuning seems needed for the models to perform on other data series. Finally, we discuss that further research focusing on other reinforcement learning methods, the use of other inputs, and more computational power could lead to an increase in performance.

Abstract - Dansk

Estimationen af det forventetede afkast og kovariansmatricen, der bliver benyttet til at bestemme den optimale portfølje policy vha. Markowitz Moderne Porteføljeteori er problematisk i praksis. Dette projekt undersøger, hvordan man kan overkomme denne problematik ved direkte at estimere den optimale portefølje policy. Dette opnås gennem dyb reinforcement learning som benytter sig af forskellige dybe neurale netværksarkitekture. Ved brug af et simulationsstudie tester vi effektiviteten af den forslåede dybe reinforcement learning model til at lære den underliggende struktur af data og dermed lære policies til at bestemme allokationer af aktiver. Resultaterne viser, at alle de forslåede modeller er i stand til at outperforme ligevægtsporteføljen på test data. Dog virker videre tuning nødvendig, hvis modellerne skal kunne performe på andre dataserier. Endeligt diskuterer vi, at mere research med fokus på andre reinforcement learning metoder, brugen af andre inputs og mere computerkraft kan føre til bedre performance.

Contents

1 Introduction

In 1952, Harry Markowitz introduced his Modern Portfolio Theory (MPT) that gave a theoretical policy for determining an optimal portfolio given an investor's risk profile. The framework assumes knowledge of the expected returns of the assets as well as the covariance matrix of those returns. This is a problem in practice. Estimating expected returns is not straight forward. Green et al. (2013) gives a supra-view of 300 different papers that all concern themselves with estimating expected returns. The covariance matrix of returns is also not simple to estimate using classical methods due to the curse of dimensionality. In finance, the number of assets is often in the same order of magnitude as the number of data points leading to severe numerical instability.

Instead of having a policy that dictates an optimal portfolio given expected returns and covariance matrix of returns estimated from data, another possibility is to directly optimize the policy given the data. We try this approach using reinforcement learning (RL), a branch of machine learning (ML). In RL, the model attempts to learn a policy that maximize a reward thus guiding the models behaviour. The reward can be whatever is deemed appropriate. This method would therefore in principle allow an investor to set an individual reward that fits their personal investment preferences. One investor might weigh ESG highly while another investor might only care about total returns, irregardless of associated risks. This method allows each of them to determine a policy constructing portfolios fitting their investment preferences.

Various ML models can be used in the framework of RL. In this project, we will be exploring the use of three different deep neural networks (DNN): convolutional neural networks (CNN), recurrent neural networks (RNN), and long short-term memory (LSTM). Thus, the methods used in this project are deep reinforcement learning (DRL) methods. The different DNNs have achieved remarkable results in a great variety of domains such as image and speech recognition, language translation, playing games etc. It will be interesting to test their efficacy in solving the optimal portfolio policy problem using RL.

This project will be a simulation study. This is done so that we can test the efficacy of the models given an actual structure in the data. It will allow us to more readily determine whether or not a policy

is performing well simply due to luck or if it has figured out an optimal portfolio given an impression of the underlying structure of the data. Furthermore, this also allows us to test similar network topologies on different data sets acting as out-of-sample data.

We investigate the following questions:

- How can DRL be used to solve the optimal portfolio policy problem?
- How do CNNs, RNNs, and LSTMs differ in their choice of optimal portfolio policy?
- To what extent can DRL be used for real world allocation problems?

This project is inspired by Jiang et al. (2017) that used DRL with the before-mentioned DNNs for allocating between cryptocurrencies. We want to investigate whether or not their proposed methods work on different data and if the allocations from the policies make sense knowing the model that generated the data.

1.1 Outline

Section 2: Markowitz Modern Portfolio Theory
In this section, we explain Markowitz Modern Portfolio Theory and its problems in practice.

Section 3: Data Simulation
Here, we explain the method used to simulate price data for an arbitrary number of assets from a given seed. We briefly explain how a stationary vector autoregression model is used and how the elements of the model can be generated from a seed.

Section 4: Machine Learning
We briefly explain machine learning in general and its core concepts. We then go through the process of how to train, validate, and test machine learning models. Lastly, we discuss the use of machine learning in finance.

Section 5: Neural Networks

In section 5, we first introduce the basic neural network, regularization methods and momentum before diving into deep neural networks. In particular, we explain convolutional neural networks, recurrent neural networks, and long short-term memory recurrent neural network.

Section 6: Reinforcement Learning

Here, we explain the general framework of reinforcement learning before explaining how we use it for solving the optimal portfolio problem. Next, we introduce the policy gradient method which is the reinforcement learning method used in this project. Lastly, we explain the implementation of the policy gradient using deep neural networks.

Section 7: Implementation

We explain our choices regarding number of assets and lags for our price data, our use of a benchmark, choice of seeds. Then we go through how we train and tune models, how we choose network topologies, as well as whether to train on price data or return data.

Section 8: Results

We go through the performance of the various models, first comparing them to each other. We investigate the learned policies and look at telling differences between them. We then compare the best models to the chosen benchmark. Lastly, we test on other seeds as well.

Section 9: Discussion

In this last section, we discuss problems, possible solutions and ideas for further work. This includes using different rewards, other reinforcement learning methods, and alternative inputs.

2 Portfolio Theory

The scope of this section is to introduce the theory behind investors decision making and define investment strategies, as well as the motivation for the use of reinforcement learning techniques in portfolio optimization. This section is written taking inspiration from Ledoit and Wolf (2017) and Hillier et al. (2012).

2.1 Markowitz Modern Portfolio Theory

Modern Portfolio Theory was developed by Harry Markowitz in 1952. The core concept of MPT is to construct a portfolio that minimizes risk given expected return. MPT assumes that investors are mean-variance optimizers, meaning that investors seeks a portfolio with the lowest possible return variance given a desired expected return. In this theoretical framework, the risk is defined as the variance of an investment return. MPT furthermore assumes that the financial markets are frictionless. Frictionless markets means that all investments are tradeable at any price and in any quantity. Furthermore, there is no short sale restrictions, and no transaction costs, regulations or tax consequences of asset purchases or sales. The investor is trying to determine an optimal portfolio. Mathematically, a portfolio is defined to be a weight vector, $\boldsymbol{a}$, where the elements indicate the proportion of the investor's wealth invested in the assets. The investor is trying to solve the minimization problem

$$\min_{\boldsymbol{a}} \frac{1}{2} \cdot \boldsymbol{a}^T \boldsymbol{\Sigma} \boldsymbol{a} \qquad subj. \quad \boldsymbol{a}^T \boldsymbol{\mu} = \mu_{pf}, \qquad \boldsymbol{a}^T \mathbf{1} = 1 \tag{1}$$

where $\boldsymbol{a}$ is the portfolio-weight vector of the n assets, $\boldsymbol{\Sigma}$ is the $(n \times n)$-covariance matrix of returns, $\boldsymbol{\mu}$ is the expected return vector for the n assets and μ_{pf} is the desired expected return of the portfolio. Solving this problem gives the following optimal portfolio-weight vector and portfolio return variance:

$$\boldsymbol{a}_{pf} = \boldsymbol{\Sigma}^{-1} \begin{bmatrix} \boldsymbol{\mu} & \mathbf{1} \end{bmatrix} \boldsymbol{A}^{-1} \begin{bmatrix} \mu_{pf} \\ 1 \end{bmatrix} \qquad \sigma_{pf}^2 = \frac{A_{22}\mu_{pf} - 2A_{12}\mu_{pf} + A_{11}}{A_{11}A_{22} - A_{12}^2} \tag{2}$$

where

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} & \boldsymbol{\mu}^T \boldsymbol{\Sigma}^{-1} \mathbf{1} \\ \mathbf{1}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu} & \mathbf{1}^T \boldsymbol{\Sigma}^{-1} \mathbf{1} \end{pmatrix}$$

It is possible for an investor to infinitely leverage their portfolio by shorting other assets in the portfolio. It is worth noting that μ_{pf} is chosen by the individual investor. This means that in this framework, no "optimal" portfolio can exist; it depends on the preferences of the individual investor. However, if two different portfolios have the same expected return, one can be judged better looking at the standard deviation of returns of each portfolio. The worse portfolio is called a dominated portfolio. Therefore, it is possible to determine a set of portfolios which are not dominated by any other portfolio. This set of portfolio is called the efficient frontier, see figure 1.

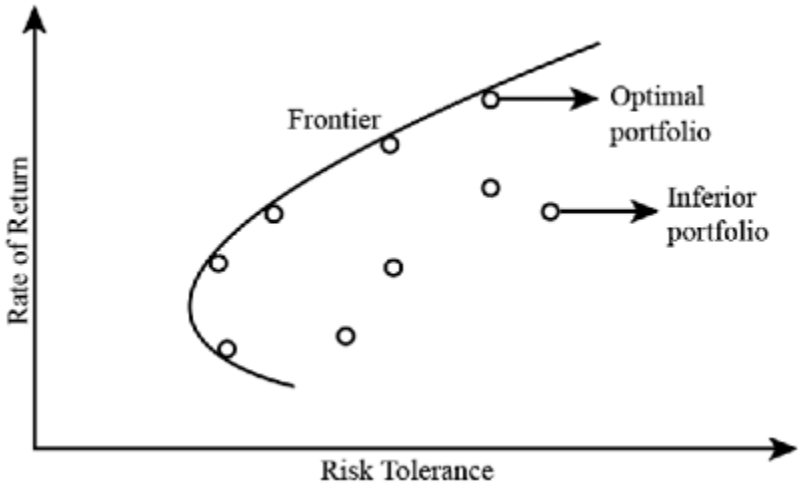


Figure 1: Efficient frontier

2.2 Practical Problems of Markowitz Modern Portfolio Theory

Essentially, MPT gives us an optimal policy for constructing a portfolio if we know the expected return, and covariance matrix of returns. A natural problem with this approach is the fact that we in practice do not know those things. Thus, we need approaches to estimate expected return and the covariance matrix of returns of assets. A problem with estimating the covariance matrix is the fact that in practice, the number of assets n is in the same order of magnitude as the sample size, making it unstable. Furthermore, we have to use the inverse of the covariance matrix, making it even more unstable. Research has been done to improve the estimation of expected returns, Green et al. (2013), and the covariance matrix of returns, Ledoit and Wolf (2017). Green et al. (2013) presents a supra-view of papers concerning the estimation

of expected returns, while Ledoit and Wolf (2017) presents a method for shrinking the covariance matrix to alleviate the problems of numerical stability.

Another problem could be the non-dynamic nature of the method. If an investor has a preference for an expected return over a given time period, i.e. a year, then MPT does not give an optimal policy for daily reallocation. Another approach is to directly approximate the optimal policy for constructing a portfolio instead of first estimating expected return, and the covariance matrix. This is what we are trying to investigate in this project using reinforcement learning on a simulated universe of assets. Though, it should be noted that the way we use reinforcement learning in this project results in long-only policies. It is not possible for them to leverage their portfolio by shorting assets.

3 Data Simulation

This section will introduce the Stationary Vector Autoregression Model, which is used for simulating the data in this project. The stationary vector autoregression model (VAR) is a common tool for multivariate time series analysis, primarily used for forecasting. In this project, the VAR model will be used as a Monte-Carlo method in order to simulate price series for an arbitrary number of risky assets. This part is inspired by Zivot and Wang (2002).

3.1 Motivation for Simulation Study

In this project, we want to research the efficacy of using reinforcement learning for determining an optimal policy for allocation between assets. However, we wish to test on data with different underlying structures. In this case, it is more feasible to test the method on many different simulations of data. This allows us to determine whether or not there are particular structures of data where the method works and structures where it falls flat. To do this on real data, we would have to procure and clean data for different assets, sectors, etc., which is unfeasible for this project. To avoid this, we have instead chosen to do a simulation study. It follows that we are testing the theoretical efficacy of a reinforcement learning approach to solving the optimal allocation problem. Furthermore, we also force an actual underlying

process and structure on the data which the model should be able to learn, in theory. On real data, we cannot be sure of the underlying process and we would be unable to draw more general conclusions from results on that data. Naturally, the simulated data should still not be too unrealistic. Finally, we can also look at the allocations determined by the approximated policies and see if they make sense given the model that generated the data.

3.2 Stationary Vector Autoregression Model

Let $\boldsymbol{Y}_t = (Y_{1t}, Y_{2t}, ..., Y_{nt})^T$ denote a stochastic vector of length n of returns on the risky assets at time t. The basic p-lag vector autoregressive, VAR(p), model takes the form

$$\boldsymbol{Y}_t = \boldsymbol{c} + \boldsymbol{\Pi}_1 \boldsymbol{Y}_{t-1} + \boldsymbol{\Pi}_2 \boldsymbol{Y}_{t-2} + ... + \boldsymbol{\Pi}_p \boldsymbol{Y}_{t-p} + \boldsymbol{\varepsilon}_t, \quad t = 1, ..., m. \tag{3}$$

Here, $\boldsymbol{\Pi}_i$ are the $(n \times n)$ coefficient matrices, $\boldsymbol{c}$ and $\boldsymbol{\varepsilon}_t$ are vectors of size n, where $\boldsymbol{\varepsilon}_t$ is an unobservable and normally distributed vector white noise with zero mean and a time invariant covariance matrix $\boldsymbol{\Sigma}$. Thus, $\boldsymbol{Y}_t | \boldsymbol{Y}_{t-1}, ..., \boldsymbol{Y}_{t-p}$ is multivariate normally distributed with mean $\boldsymbol{c} + \boldsymbol{\Pi}_1 \boldsymbol{Y}_{t-1} + ... + \boldsymbol{\Pi}_p \boldsymbol{Y}_{t-p}$ and covariance $\boldsymbol{\Sigma}$. Therefore, returns approach a multivariate normal distribution, and since

$$Y_{it} \approx \log(1 + Y_{it}) = \log\left(\frac{X_{i,t}}{X_{i,t-1}}\right) = \log(X_{i,t}) - \log(X_{i,t-1}), \quad \forall i = 1, ..., |\boldsymbol{Y_t}|,$$

then prices given prices at last time step are approximately log-normally distributed for each asset

$$\log(\boldsymbol{X}_t) | \boldsymbol{X}_{t-1} = \log(\boldsymbol{X}_{t-1}) + \boldsymbol{Y}_t \sim N(\log(\boldsymbol{X}_{t-1}) + \mathbb{E}[\boldsymbol{Y}_t], \boldsymbol{\Sigma}),$$

where $\boldsymbol{X}_t$ is the price vector of size n at time t. Note that when any function with real valued input is applied to a vector, it is understood that the function is applied element-wise. Having returns and prices being normally and approximately log-normally distributed respectively are convenient properties given the nature of returns and prices. Returns can be both positive and negative while prices can only take positive values.

If the companion matrix, $\boldsymbol{F}$, defined by

$$\boldsymbol{F} = \begin{bmatrix} \boldsymbol{\Pi}_1 & \boldsymbol{\Pi}_2 & \dots & \boldsymbol{\Pi}_n \\ \boldsymbol{I} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} & \vdots \\ \mathbf{0} & \mathbf{0} & \boldsymbol{I}_n & \mathbf{0} \end{bmatrix},$$

has modulus of eigenvalues less than one then the VAR(p) process converges to a stationary distribution. In this project, a VAR(1) model is used for n risky assets

$$\begin{pmatrix} Y_{1t} \\ \vdots \\ Y_{nt} \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} + \begin{pmatrix} \pi_{11}^1 & \cdots & \pi_{1n}^1 \\ \vdots & \ddots & \vdots \\ \pi_{n1}^1 & \cdots & \pi_{nn}^1 \end{pmatrix} \begin{pmatrix} Y_{1t-1} \\ \vdots \\ Y_{nt-1} \end{pmatrix} + \begin{pmatrix} \varepsilon_{1t} \\ \vdots \\ \varepsilon_{nt} \end{pmatrix}. \tag{4}$$

Thus, $\boldsymbol{F} = \boldsymbol{\Pi}_1$. Setting $\boldsymbol{\Pi}_1$ to have eigenvalues with modulus less than one makes $\boldsymbol{Y}_t$ approach a multivariate normal distribution with unconditional mean given by

$$\boldsymbol{\mu} = (\boldsymbol{I}_n - \boldsymbol{\Pi}_1)^{-1}\boldsymbol{c} \tag{5}$$

and covariance matrix that can be found by solving the difference equation

$$V[\boldsymbol{Y}_t] = V[\boldsymbol{c} + \boldsymbol{\Pi}_1 \boldsymbol{Y}_{t-1} + \boldsymbol{\varepsilon}_t].$$

Note that $\boldsymbol{\Pi}_1 \boldsymbol{Y}_{t-1}$ and $\boldsymbol{\varepsilon}_t$ are independent since all $\boldsymbol{\varepsilon}_i$ are independent of each other and of $\boldsymbol{Y_0}$. Then the difference equation becomes

$$\begin{aligned} V[\boldsymbol{Y}_t] &= V[\boldsymbol{\Pi}_1 \boldsymbol{Y}_{t-1}] + V[\boldsymbol{\varepsilon}_t] \\ &= \boldsymbol{\Pi}_1 V[\boldsymbol{Y}_{t-1}] \boldsymbol{\Pi}_1^T + \boldsymbol{\Sigma} \end{aligned}$$

When the process is stationary, then

$$V[\boldsymbol{Y}_t] = V[\boldsymbol{Y}_{t-1}] = \boldsymbol{\Sigma}_u$$

$$\boldsymbol{\Sigma}_u = \boldsymbol{\Pi}_1 \boldsymbol{\Sigma}_u \boldsymbol{\Pi}_1^T + \boldsymbol{\Sigma}.$$

Thus, the unconditional covariance matrix $\boldsymbol{\Sigma}_u$ is given by

$$\boldsymbol{\Sigma}_u = (\boldsymbol{I}_n - \boldsymbol{\Pi}_1)^{-1} \boldsymbol{\Sigma} (\boldsymbol{I}_n - \boldsymbol{\Pi}_1^T)^{-1} \tag{6}$$

These elements of the VAR(1)-model will later be used to analyze the decision making of the deep reinforcement learning models.

3.3 Simulation

In order to be able to generate data structures from different seeds, core elements from the VAR(1)-model have been implemented with a stochastic aspect. Different seeds will result in vastly different data. This feature will be important when analyzing whether the proposed models in the projects can learn the price structures across different seeds.

Starting with $\mathbf{\Pi}_1$, its eigenvalues are chosen to be real-valued and numerically smaller than 1, making the time series stationary. The eigenvalues are drawn from a uniform distribution with minimum and maximum values numerically smaller than 1. In particular, we draw from $\mathcal{U}(-0.3, 0.3)$ and the eigenvalues are placed in a diagonal matrix, $\mathbf{\Lambda}$. Then, a random $(n \times n)$-matrix, $\boldsymbol{M}$, is created where values are drawn from $\mathcal{U}(0, 1)$. A SVD-factorization is then performed on $\boldsymbol{M}$ so that $\boldsymbol{M} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$. It is known that $\boldsymbol{R} = \boldsymbol{U}\boldsymbol{V}^T$ is a unitary matrix. Thus, $\mathbf{\Pi}_1$ can be constructed as

$$\mathbf{\Pi}_1 = \boldsymbol{R}\mathbf{\Lambda}\boldsymbol{R}^T,$$

ensuring eigenvalues have modulus less than 1.

The conditional covariance matrix $\mathbf{\Sigma}$ is created using scipy. With scipy, it is possible to generate a random correlation matrix $\boldsymbol{C}$ given an array of eigenvalues that sum to the size of $\boldsymbol{C}$. The eigenvalues are drawn from the Dirichlet distribution Dir$(\mathbf{1})$ and then multiplied by the size of $\boldsymbol{C}$. Standard deviations of returns for the assets are then each drawn from $\mathcal{U}(0.008, 0.02)$ and placed into a diagonal matrix $\boldsymbol{D}$. Then, the covariance matrix is

$$\mathbf{\Sigma} = \boldsymbol{D}\boldsymbol{C}\boldsymbol{D}.$$

Lastly, a $\boldsymbol{c}$ is chosen so that the elements of $\boldsymbol{\mu}$ are generally larger than 0. If the conditional distribution of returns is symmetric around 0 then prices will tend towards 0. As an example, if in one time period the return is 0.1 and -0.1 the next, then the total return is $(1+0.1)\cdot(1+(-0.1))-1 = -0.01$. So for prices not to simply tend towards 0, the unconditional mean of return should be larger than 0. This is done by drawing the elements of $\boldsymbol{\mu}$ from an appropriate uniform distribution and then calculating a $\boldsymbol{c}$ from

$$\boldsymbol{c} = (\boldsymbol{I}_n - \mathbf{\Pi}_1)\boldsymbol{\mu}.$$

Now, the only thing missing is generating the starting prices and returns, $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$. The starting prices $\boldsymbol{y}_0$ are drawn from $\mathcal{U}(100, 200)$, and $\boldsymbol{x}_0$ is drawn from $\mathcal{U}(-0.005, 0.02)$ The VAR(1)-model is therefore generated using only the number of trading days to simulate, m, and a seed. The Monte-Carlo algorithm for generating the times series of returns and prices can be formulated as in algorithm 1. Figure 2 shows a simulated price series of 10 assets using algorithm 1. This is the price series, we will later be working on.

Algorithm 1 Monte-Carlo Algorithm

1: **Inputs:**

Seed, m = Number of trading days to simulate

2: **Initialize:**

$\boldsymbol{\mu}$, $\boldsymbol{c}$, $\boldsymbol{\Pi}_1$,$\boldsymbol{\Sigma}$,$\boldsymbol{y}_0$ & $\boldsymbol{x}_0$

3: **for** t = 1 to T **do**

4: $\boldsymbol{y}_t = \boldsymbol{c} + \boldsymbol{\Pi}_1 \boldsymbol{y}_{t-1} + \boldsymbol{\varepsilon}_t$

5: $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} \cdot (1 + \boldsymbol{y}_t)$

6: **end for**

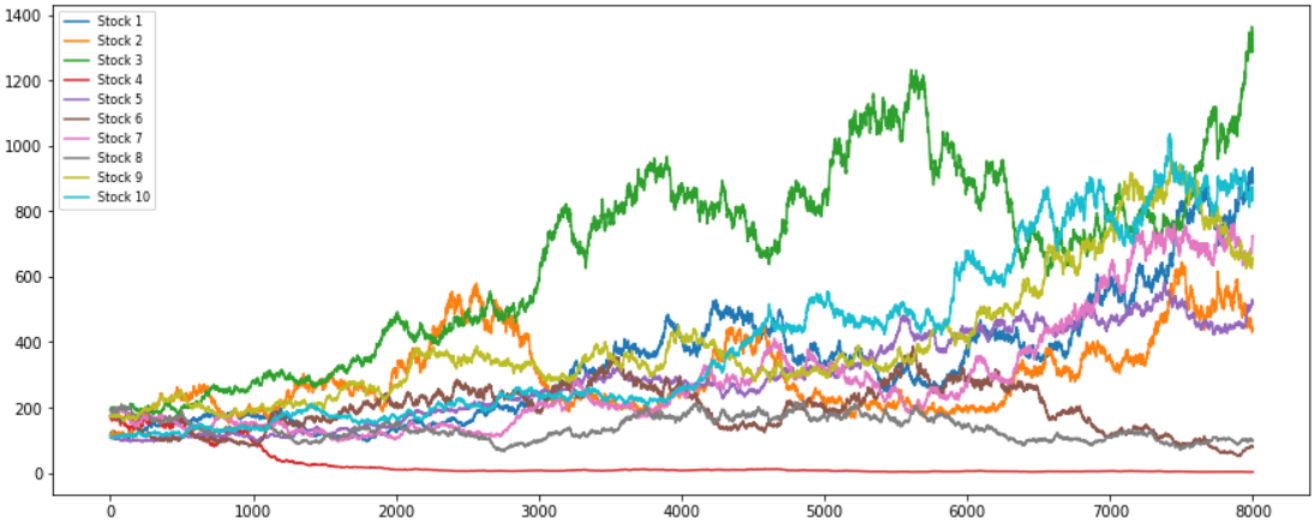


Figure 2: Price series generated from seed 1

For the rest of this project, we will in effect pretend to not know the underlying structure of the model so as not to make choices we know would make more sense given the data model.

4 Machine Learning

This section will introduce the basics of machine learning before elaborating on the concepts of machine learning, neural networks and reinforcement learning, used in this paper to select the optimal portfolio policy. The knowledge to write this section has been obtained from Sutton and Barto (2018), Dixon et al. (2020) and Israel et al. (2020).

4.1 Introduction

Machine learning was introduced in 1959 by Arthur Samuel. In recent years there has been an exponential surge in the application of ML meaning it is now deeply embedded in our every day lives; from translation to detecting diseases. ML is a branch of computer science and applied statistics. Concretely, ML algorithms build models based on a data set to make predictions or decisions. This leads us to how ML models can be classified according to type - supervised, unsupervised or reinforcement learning - by the level of supervision they have.

In supervised ML, we use the ground truth, that is we know the desired result/output. Here, a model is trained to learn a function that maps input to output based on a structure of input-output pairs. From these pairs, the objective is to predict new output values based on new inputs. Inputs has to be labeled often making a need for human pre-processing which is a major well known problem for supervised learning. Supervised ML is used for classification problems; problems that concern mapping input to the correct label e.g. predicting the label, Iris, based on the inputs petal length and width. Furthermore, supervised learning is also used for regression problems; problems that concern mapping input to a continuous output e.g. predicting salary based on inputs such as age, years spent studying, etc.

Unsupervised learning on the other hand is where you have no ground truth. Unsupervised learning algorithms take a set of data containing only inputs, and find structure in the data, like clustering of data points or reducing the dimensionality of the data set. The algorithms learn from data that has not been labeled, classified or categorized. Instead of responding to feedback, unsupervised learning algorithms identify commonalities in the data and uses the presence or absence of those commonalities when looking

at new pieces of data. Some of the most common algorithms in the field of unsupervised learning are Principal Component Analysis (PCA) for dimensionality reduction and K-Means Clustering.

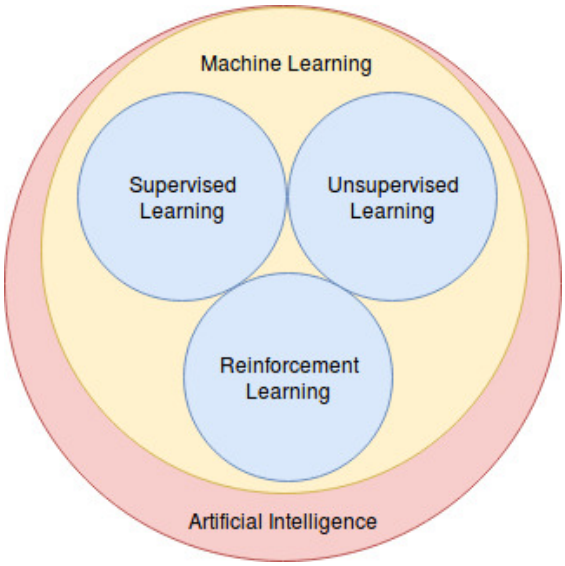


Figure 3: The AI Circle

4.2 A General Framework for Supervised and Reinforcement Learning

In supervised learning, a hypothesis function $h : \mathcal{X} \mapsto \mathcal{Y}$ takes an input and maps it into a prediction. An example could be the linear hypothesis function

$$h(\boldsymbol{X}) = \boldsymbol{W}\boldsymbol{X} + \boldsymbol{b} = \hat{\boldsymbol{y}},$$

where $\boldsymbol{W}$ is a weight parameter and $\boldsymbol{b}$ is a bias parameter. The weight and bias parameters are fitted based on data to determine the "best" weight and bias. The best weight and bias are determined using the notion of a loss function, $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}$. A loss function is a measure the ML model uses to get a sense of how good its predictions or classifications are. The greater the value of the output of the loss function the worse the prediction. Next we define the cost function, $J(\boldsymbol{W}, \boldsymbol{b})$, which measures how far off the predictions, $\hat{\boldsymbol{y}}$, of the model on average was compared to the true values. The general cost function is of the form

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y_i, \hat{y}_i),$$

where m is the number of data points. The cost function is simply a measure of how the ML model's predictions on average was compared to the true labels. The Mean Squared Error (MSE)

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2,$$

is a very common cost function when dealing with a regression problem. For a binary classification task, a common cost function is the binary cross entropy loss

$$J(\boldsymbol{W}, \boldsymbol{b}) = -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \right),$$

where $\hat{\boldsymbol{y}}$ is the predicted probability from the ML model e.g. the probability of being a man if the classification task was to distinguish men from women.

A reinforcement learning model outputs an action rather than a prediction. The only feedback the model gets is a numerical score to give an indication for how good that action was. Instead of having a loss function, the feedback is instead a reward by convention. The parameters are then fitted in order to maximize the rewards. Reinforcement learning will be explained more in-depth in section 6.

4.3 Bias/Variance Trade-off

The primary goal of every ML problem is to find a model specification that achieves a great performance on both the data that the model is trained on and the data that the model is tested on i.e. the training data and the test data. In other words, the objective is to find a model that has a sufficient level of complexity that simultaneously generalizes well to unseen data. To successfully achieve this goal, the designer of a ML model has to balance the *bias* and *variance* of the model. Bias refers to when the model fails to fit to the signal while variance refers to when the model fits to the noise. The objective is obviously to both have low bias and low variance but lowering the bias by introducing greater complexity to the model will often cause a greater variance and vice versa. This dilemma is referred to as the bias/variance trade off and is illustrated in figure 4. From the figure, the inverse relationship between the bias and variance is illustrated together with overall generalization error. The objective is to somehow find the point that minimizes the overall error by tuning the bias and variance of the model. As mentioned, the bias can often

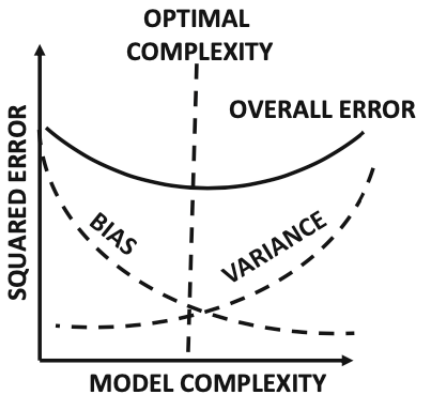


Figure 4: The Bias Variance Trade off

be reduced by creating a more complex model e.g. fitting a linear regression with a polynomial term while the variance of a model could be reduced by removing a polynomial term in a linear regression.

4.4 Training, Validating and Testing a Machine Learning Model

Every ML model needs a preprocessed data set. Assuming the data have been cleaned, a common first step is to split the data set into a training, a validation, and a test set. The training data is a sample used to fit the model's parameters by minimizing the cost. The training is guided using hyperparameters controlling the rate and the way the model is trained. The validation data is used to find the set of hyperparameters that maximizes the performance on some fixed evaluation metric. The most common metrics to maximize for in classification tasks are accuracy, precision, recall and the F-score while MSE is typically used for regression problems.

After having found a model with parameters that minimizes the cost and a set of hyperparameters that maximizes performance on the validation data, the model is evaluated on the test data to asses the model's performance on unseen data hence its ability to generalize.

Often, the ML model trains iteratively for a certain number of epochs (training iterations). The validation can then be used to create loss curves. The training and validation loss curves show loss of the model

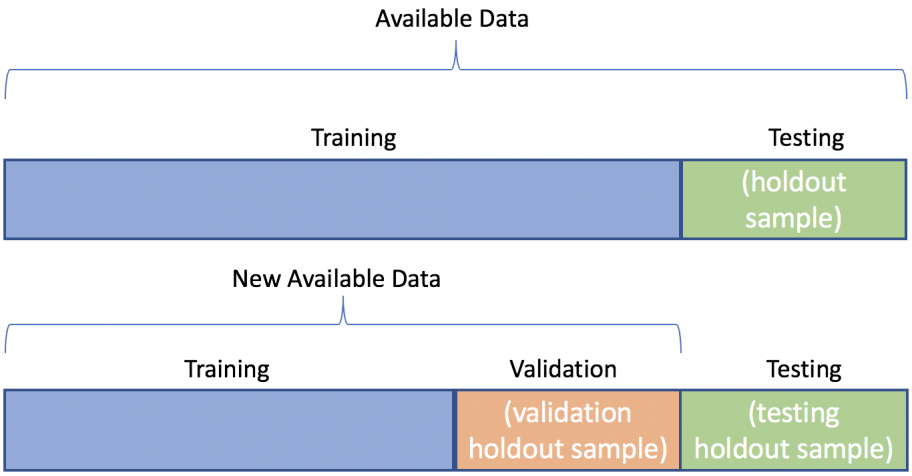


Figure 5: Data split

on training and validation data respectively dependent on the number of epochs the model has trained for. The loss curves give a visual method to get a sense for whether the ML model has been underfitted, overfitted, or fitted just right. If the model is underfitted, it means that it has failed to fit to the signal. Therefore, it would fail to achieve a low overall error on both the training and validation data. This would result in the loss curves for both the training data and validating data not dropping as the epochs rises as illustrated in figure 6. To reduce the bias, the designer of the ML model would simply try to add some more complexity to the model. Another sign of underfit is a result of the model not having trained enough, when it is clear from the loss curve that it could improve by increasing the model as illustrated in figure 7.

To detect if the model has overfitted, the loss of the validation data will often diverge from the training loss. This is a sign that the model have a high variance as it fits to the noise within the training data, resulting in a poor performance on the unseen validation data. This is illustrated in figure 8. One way to accomodate this problem would be to reduce the complexity of the model by regularizing the model. Another method could be to use early stopping, which is simply to stop the training of the model at the point where the validation loss starts to diverge from the training loss. A sign that the model has balanced the bias/variance trade-off well is if both the training and validation curves flattens as the number of epochs rises and the validation loss converges to the training loss as illustrated in figure 9.

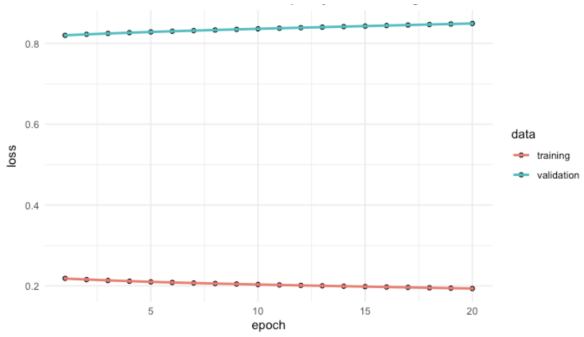


Figure 6: Complexity Needed

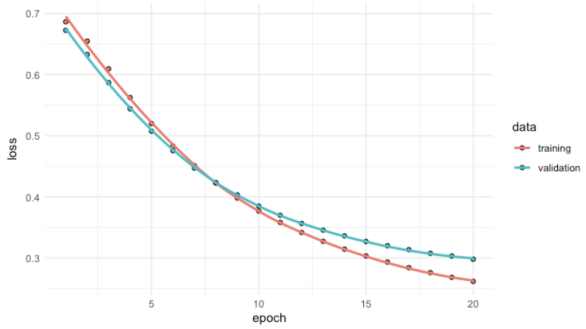


Figure 7: More training needed

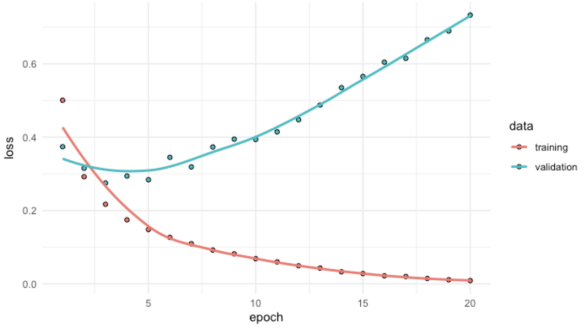


Figure 8: Needs regularization

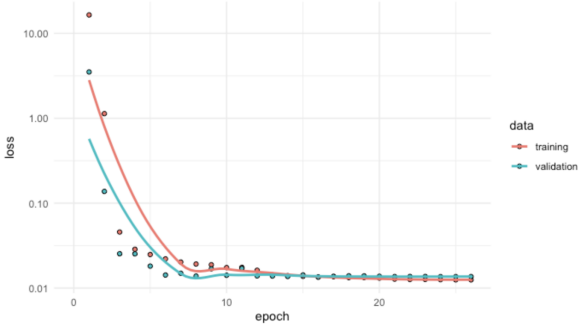


Figure 9: Good fit

4.5 Machine Learning in Finance

Machine Learning as a discipline in finance range in perception from hype to skepticism as people are still developing a basic understanding. Some of the hype comes from the great results of ML such as recognising images and speech, driving cars, and beating grandmasters at complex games of strategy. It should however be made clear that problems in finance differs in many ways compared to other problems. First of all, finance problems often have what will be considered a smaller data set at least in our asset allocation problem using daily data. Secondly, finance data tends to have a low *signal-to-noise ratio* - a ratio describing how much predictability exists within the system. Where identifying a cat in an image can be done with high accuracy - hence a high signal-to-noise ratio - financial data such as stock prices has a high degree of randomness embedded. Additionally finance data comes from an evolving market. This means a model that worked well yesterday may not work well today. A final point is the need for interpretability. The ability to understand the inner workings of one's model is a basic requirement in asset management. This plays into the risk-return trade-off mentioned in Markowitz. While any asset

manager prefers a more predictive model, all else equal, they can be averse to using black box ML models that have proved historically reliable but cannot be interpreted. In general in finance, there is a need to be able to reason ones decision making in order to take proper responsibility.

Despite these obstacles there is ways and arguments for ML to thrive in finance. The practice of ML in finance has grown somewhat commensurately with both theoretical and computational developments in ML. Some of the first to implement ML in a financial field was quantative hedge funds such as Bridgewater Associates, WorldQuant, Two Sigma and more. Some of its fields today are algorithmic trading, high-frequency trade execution and mortgage modeling. With ML already finding cases in finance were it is of use there is reason to further explore how new theory and stronger computational power can be utilized in finance. The incorporation of ML in finance is well put by Israel et al. (2020) as "an evolution, not a revolution". One of the new phenomenons in this evolution is RL hence the motivation for this project.

5 Deep Neural Networks

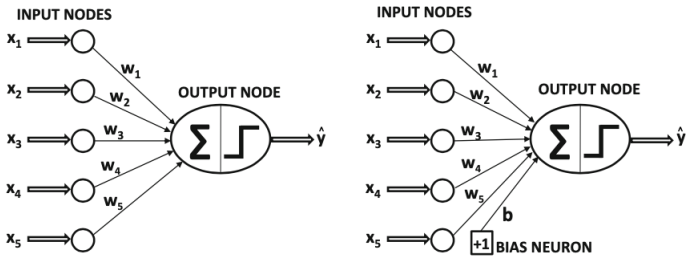
The scope of this section is to introduce and explain the core concepts of deep neural networks (DNN) and explain three of its variations which will be implemented later in our reinforcement learning models. This section has taken inspiration from Aggarwal (2018), Dixon et al. (2020) and Understanding LSTM Networks (2015).

The figures relating to the regular neural networks are from Aggarwal (2018).

5.1 The basic architecture of neural networks

5.1.1 The Perceptron

The simplest neural network (NN) is the perceptron which consists of an input layer and a single output node. The perceptron is depicted in figure 10. First, n features, $\boldsymbol{x} = [x_1, x_2, ..., x_n]$, are fed into the input layer which are then passed on to the output node. In our project, these features will be the prices at



(a) Perceptron without bias (b) Perceptron with bias

Figure 10: The basic architecture of the perceptron

time t for the n assets in our simulated universe. Each input has its own corresponding weight which can be interpreted as the signal strength of that exact input feature. The weights corresponding to the inputs are kept within the vector $\boldsymbol{w} = [w_1, w_2, ..., w_n]$. The linear function $\boldsymbol{w}^T\boldsymbol{x}$ is computed at the output note and then passed through an activation function, Φ, in order to map the linear function to a prediction, $\hat{\boldsymbol{y}}$. Optionally a bias node can be added to the computation of the linear function as an invariant part of the prediction. This is achieved by adding a node that always transmits a value of 1 to the output node with a weight connecting the bias node to the output node. In total the prediction is modeled as

$$\hat{\boldsymbol{y}} = \Phi(\boldsymbol{w}^T\boldsymbol{x} + b) = \Phi\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right). \tag{7}$$

5.1.2 The Multi-Layer Perceptron

The perceptron contains one computational layer, which is the output layer. The multi-layer perceptron (MLP) is a neural network which consist of more than one computational layer. The additional intermediate layers between the input and output layers are referred to as *hidden layers* because the computations performed are not visible to the user. As depicted in figure 11, the MLP can have one or more hidden layers, each containing one or more nodes. In the figure, the neural network consist of 2 hidden layers each containing 3 nodes.

The weights of the connections between the input layer and the first hidden layer are contained in a matrix, $\boldsymbol{W_1}$, with size $n \times p_1$ where n is the dimension of the input vector and p_1 is the number of nodes within the first hidden layers, $\boldsymbol{h}_1$. The weights between the r^{th} hidden layer and the $(r+1)^{\text{th}}$ hidden layer

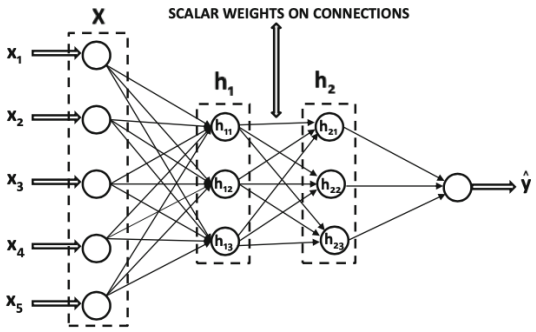


Figure 11: The basic architecture of the multi-layer perceptron

are denoted by the $(p_r \times p_{(r+1)})$-matrix W_{r+1} where p_r is the number of nodes within the r^{th} hidden layer. The computations performed within each hidden layer are thus

$$\boldsymbol{h}_{r+1} = \Phi(\boldsymbol{W}_{r+1}\boldsymbol{h}_r + \boldsymbol{b}_{r+1}), \tag{8}$$

where we will denote the pre-activation value of a nodes in a layer $\boldsymbol{a}$

$$\boldsymbol{a}_{r+1} = \boldsymbol{W}_{r+1}\boldsymbol{h}_r + \boldsymbol{b}_{r+1}. \tag{9}$$

It is worth noting that the activation function, Φ, can vary across layers. The activation functions are primarily used to introduce non-linearity and thus complexity to the network but also to shape the output of the network.

5.1.3 Training a Neural Network

The NN is trained using the cost function introduced earlier. The optimization of the cost function does not have any closed form solution. The neural network is therefore optimized with an iterative algorithm called the gradient descent algorithm, which takes the form

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}), \tag{10}$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}). \tag{11}$$

The gradient corresponding to the loss function in a multi layered neural network is a complicated composition function of the weights in earlier layers. The gradient of a composition function is computed

using the backpropagation algorithm. The backpropagation algorithm leverages the chain rule of differential calculus which computes the error gradients in terms of summations of local-gradient products over various paths from a node to a output. In this way, the neural network can efficiently figure out which parts within the neural network contributed the most to the loss and hence needs the most adjustment.

The backpropagation algorithm has two phases; the forward phase and the backward phase. In the forward phase, the inputs for all the training instances are fed into the neural network. This is simply a forward pass through all the layers in the network with its current weights and bias parameters which then results in predictions, $\hat{\boldsymbol{y}}$. These predicted values can then be used to compute a cost. In the second phase, the backward phase, the gradient of the cost function is computed with respect to the current weights and bias parameters. For an arbitrary layer, the computations are as follows

$$\frac{\partial J}{\partial \boldsymbol{W}_r} = \frac{\partial J}{\partial \boldsymbol{h}_r} \cdot \frac{\partial \boldsymbol{h}_r}{\partial \boldsymbol{a}_r} \cdot \frac{\partial \boldsymbol{a}_r}{\partial \boldsymbol{W}_r}, \tag{12}$$

$$\frac{\partial J}{\partial \boldsymbol{b}_r} = \frac{\partial J}{\partial \boldsymbol{h}_r} \cdot \frac{\partial \boldsymbol{h}_r}{\partial \boldsymbol{a}_r} \cdot \frac{\partial \boldsymbol{a}_r}{\partial \boldsymbol{b}_r}. \tag{13}$$

Since these gradients are learned in the backward direction, starting from the output node, this learning process is referred to as the backward phase.

5.2 Regularization

NN's have achieved remarkable accomplishments in the past. They have succeeded in learning complex functions in a great number of domains and are hence said to be powerful learners. However, NN's are also prone to overfitting the training data if the designer does not carefully take this into consideration when building and designing the NN. The designer of an NN can tackle the problem of overfitting to the training data by creating simpler models i.e. lower the number of hidden layers and hidden units. Another option for the designer is to *regularize* the NN. Regularizations are techniques to reduce overfitting. The use of regularization techniques allows the designer to deliberately overparameterize the model. In the following, the concepts of L2 regularization and dropout will be covered as these are the techniques used to regularize the networks in this project.

5.2.1 L2-regularization

The most common way to reduce overfitting of a model is to use penalty-based regularization. The main intuition with L2-regularization is that it tries to shrink the weights that where highly activated when a high loss occured. For an NN, L2-regularization will take the following form

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}_i, y_i) + \frac{\lambda}{m} \sum_{r=0}^{R} \sum_{i=1}^{p_r} ||\boldsymbol{W}_{r+1}^{i}||^2, \tag{14}$$

where R is the number of hidden layers in the NN, $\boldsymbol{W}_{r+1}^{i}$ is the vector of weights connecting the i^{th} node in the r^{th} layer to the $(r+1)^{\text{th}}$ layer, and λ is another hyperparameter like the learning rate α discussed above. Tuning the value of λ controls the softness of the penalty associated with the weight parameter. This way, the weights associated with unnecessary hidden nodes will get exceedingly small, depending on λ. This will be more or less equivalent to choosing a lower number of weight parameters.

5.2.2 Dropout

Dropout is a technique to reduce overfitting by constantly turning different input and hidden nodes on and off. Note that in this project, only hidden nodes will be switched on and off as there is no point in leaving out some assets in the investment universe. If the NN contains U nodes, then it is possible to sample 2^U different NN's. In order for dropout to work, it uses weight-sharing meaning that the weights will be shared across the sampled NN's during training. The training process with dropout can be summarized in the following three steps:

1. Sample a NN from the base NN where the hidden nodes are sampled independently with probability ψ. Note that when a node is removed from the sampled NN, all its incident edges are removed as well.

2. Sample a training batch with the sampled NN

3. Update the weights of the active nodes using backpropagation on the training batch

Dropout samples thousands of NN's from the base NN forcing the base NN to constantly learn different activations and not memorize activations from the past. Intuitively, dropout can be understood in the

following way: Assume that every employee at a small company with probability ψ has to show up to work meaning that everyday some employees might not be present. Perhaps the employee responsible for accounting might not be present and therefore another employee must step in to handle the accounting tasks. This forces the employees to learn different *activations* and thus have a diverse skillset as there is probability ψ that a colleague (node) in another department might not show up. Intuitively, this means that multiple nodes will try to do same thing in the NN. These redundancies again allow the designer of the network to overparameterize the model knowing parts will end up doing the same thing.

5.3 Momentum

Momentum-based techniques recognizes that a constant learning rate is not desirable. If the learning rate is too big, the gradient descent algorithm might take too long of a step in the direction of the optimal solution and actually end up longer away from the optimal solution. If the learning rate is too small, the algorithm might end up in a local minimum while also increasing the time of training significantly. A momentum-based technique tries to move in an "averaged" direction of the past steps in order to achieve a smoother and faster path towards the optimal solution. This intuition is visible from the general momentum-based gradient update

$$\boldsymbol{\nu} \leftarrow \beta\boldsymbol{\nu} + \alpha\nabla_{\boldsymbol{W}} J(\boldsymbol{W}),$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu},$$

where $\beta \in (0, 1)$ is a smoothing parameter. For large values of β, the gradient descent algorithm picks up a consistent velocity $\boldsymbol{\nu}$ in the direction of an optimal solution. Note that for $\beta = 0$, the above update rule reduces to the usual gradient descent update.

Momentum will often force the gradient updates to overshoot in the direction where velocity is high. This is intuitively the same as when a marble rolls down a bowl in which it gains great momentum and overshoot the bottom after which it will roll up and down each edge with a decaying velocity. However, with a carefully chosen β, the overshooting is often far superior to a gradient update without momentum as the quicker arrival to the solution offsets the overshooting. Figure 12 illustrates the challenges that the gradient descent algorithm faces and how added momentum helps to overcome these pitfalls.

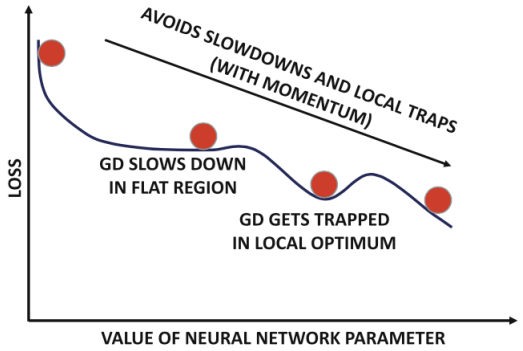


Figure 12: Momentum

5.3.1 Nesterov Momentum

The above proposed momentum-based technique helps the network to quicker converge towards the optimal solution by increasing velocity taking previous gradient updates into consideration. However, the traditional momentum-based technique suffers from overshooting in the direction where velocity is high. The Nesterov momentum is a modification of the above proposed momentum technique which uses a corrected gradient that takes the current momentum step into consideration. The motivation is that this corrected gradient has more information available as it has a better understanding of how the gradients will change due to the momentum part. The Nesterov momentum is performed with the following gradient update

$$\boldsymbol{\nu} \leftarrow \beta\boldsymbol{\nu} + \alpha\nabla_{\boldsymbol{W}} J(\boldsymbol{W} - \beta\boldsymbol{\nu}),$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boldsymbol{\nu}.$$

The only difference between the momentum-technique proposed earlier and Nesterov is the point at which the gradients are computed. Using the value a little further along the previous update can lead to a faster convergence. Using the intuition of a marble as before, Nesterov momentum incorporates a set of brakes on the gradient update when the marble is close to the bottom of the bowl as the look ahead will act as a warning sign, informing the marble about a reversal of the direction in the gradient.

5.4 Recurrent Neural Network

In a time-series data set, the values on succesive time-stamps are closely related to one another. In a regular NN, these time-stamps are fed as independent features into the network, which causes key information from the relationship between the values to be lost. The time-series can have autoregressive properties, meaning the data point at time t can be dependent on its former values. Recurrent Neural Networks (RNN) generalizes classical linear autoregression models.

Figure 13 is from Dixon et al. (2020), though altered to better fit the notation used in this paper.

5.4.1 The Basic Architecture of Recurrent Neural Networks

The simulated data in this project is autocorrelated observations, X and Y, at times $t = 1, ..., m$, thus the prediction problem can be expressed as a sequence prediction problem

$$\hat{\boldsymbol{y}}_{t+1} = f(\boldsymbol{X}_t) \quad where \quad \boldsymbol{X}_t := seq_{T,t}(\boldsymbol{X}) = (\boldsymbol{x}_{t-T+1}, ..., \boldsymbol{x}_t), \tag{15}$$

where t denotes the current timestamp, T denotes the number of lags and $\boldsymbol{x}_{t-j}$ is the j^{th} lagged observation of $\boldsymbol{x}_t$.

RNN's apply an autoregressive function $f^{(1)}_{\boldsymbol{W}_x, \boldsymbol{W}_h, \boldsymbol{b}_h}(\boldsymbol{X}_t)$ to each input sequence, $\boldsymbol{X}_t$. As illustrated in figure 13, a simple RNN is an unfolding of a single hidden layer neural network over all time steps in the sequence, $j = 0, .., T-1$. For each timestep, j, the network generates a hidden state $\boldsymbol{h}_{t-j}$ by applying the autoregressive function on the current $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-j-1}$:

$$\boldsymbol{h}_{t-j} = f^{(1)}_{\boldsymbol{W}_x, \boldsymbol{W}_h, \boldsymbol{b}_h}(\boldsymbol{X}_{t,j}) = \Phi^{(1)}(\boldsymbol{W}_h \boldsymbol{h}_{t-j-1} + \boldsymbol{W}_x \boldsymbol{x}_{t-j} + \boldsymbol{b}_h) \quad \text{where} \quad \boldsymbol{X}_{t,j} = seq_{T,t-j}(\boldsymbol{X}) \subset \boldsymbol{X}_t.$$

Here $\Phi^{(1)}$ is an activation function such as $\tanh$. The connections between the inputs $\boldsymbol{x}_t$ and the hidden nodes in the hidden layer are weighted by the time invariant matrix $\boldsymbol{W}_x \in \mathbb{R}^{p \times n}$ where p is the number of hidden nodes in the hidden layer. The recurrent connections between the hidden nodes are weighted by the time invariant matrix $\boldsymbol{W}_h \in \mathbb{R}^{p \times p}$ and $\boldsymbol{b}_h$ denotes the bias vector. Finally, the predicted value at time t of the n assets is calculated as follows

$$\hat{\boldsymbol{y}}_t = f^{(2)}_{\boldsymbol{W}_y, \boldsymbol{b}_y}(\boldsymbol{h}_t) = \Phi^{(2)}(\boldsymbol{W}_y \boldsymbol{h}_t + \boldsymbol{b}_y),$$

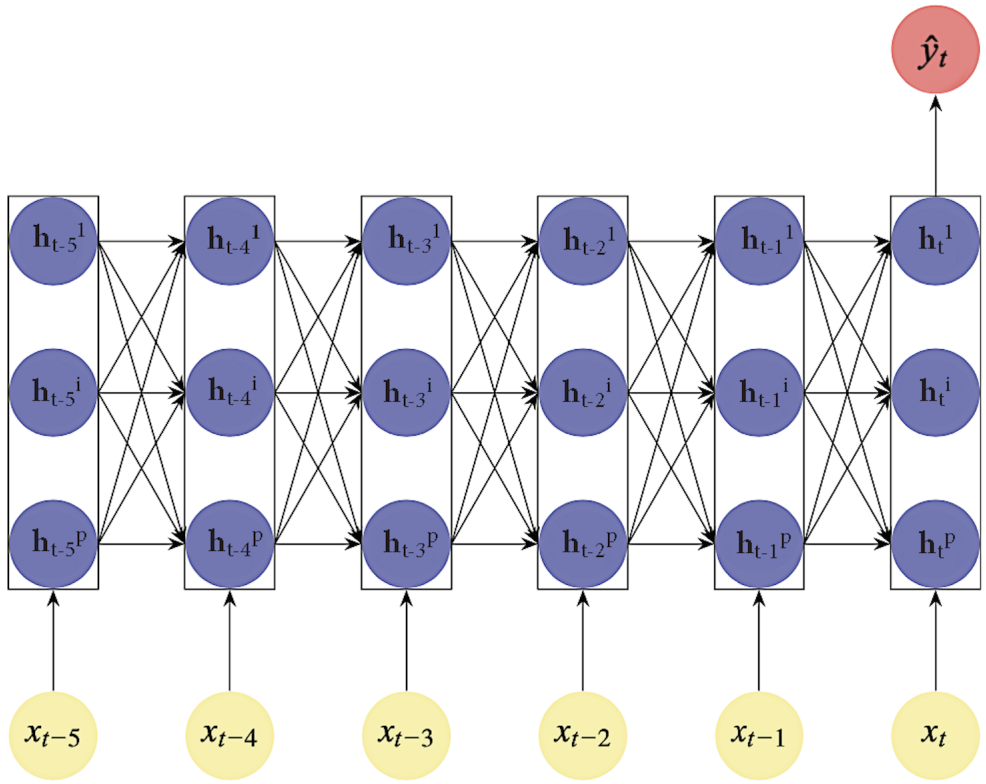


Figure 13: RNN with $T = 6$

where $\Phi^{(2)}$ denotes the softmax function, $\boldsymbol{W}_y \in \mathbb{R}^{p \times n}$ denotes the weights tied to the output of the p hidden units in the hidden layer and $\boldsymbol{b}_y$ is a bias vector.

For RNNs with multiple RNN-layers, the input for the next RNN-layer is simply the hidden states of the former RNN-layer.

5.4.2 Training a Recurrent Neural Network

Backpropagation Through Time (BPTT) is the training algorithm used to update the weights and bias parameters in a RNN. The main problem with training a RNN is the parameter-sharing across different temporal layers which will have an effect on the update process. The solution to the shared parameters is to assume that the parameters in the temporal layers are distinct. For this purpose we introduce the temporal variables $\boldsymbol{W}_x^{(t)}, \boldsymbol{W}_h^{(t)}, \boldsymbol{W}_y^{(t)}, \boldsymbol{b}_h^{(t)}$ and $\boldsymbol{b}_y^{(t)}$ at time t. The trick is now to apply the conventional

backpropagation algorithm explained in section 5.1.3 on the introduced temporal variables. Next, the gradients at each time step of each parameter are summed in order to create a unified update for each parameter. The BPTT algorithm can be summarized in the following 3 steps:

1. Feed forward an input sequence and compute the loss with the associated prediction of the network at time t

2. Compute the gradients of the weight and bias parameters at each time step t. That is, compute:

$$\frac{\partial J}{\partial \boldsymbol{W}_i^{(t)}}, \quad i = x, h, y,$$

$$\frac{\partial J}{\partial \boldsymbol{b}_k^{(t)}}, \quad k = h, y$$

3. Sum the gradients at each time step to one unified gradient:

$$\frac{\partial J}{\partial \boldsymbol{W}_i} = \sum_{t=0}^{T-1} \frac{\partial J}{\partial \boldsymbol{W}_i^{(t)}}, \qquad i = x, h, y$$

$$\frac{\partial J}{\partial \boldsymbol{b}_k} = \sum_{t=0}^{T-1} \frac{\partial J}{\partial \boldsymbol{b}_k^{(t)}}, \qquad k = h, y$$

4. Update the weight matrices and bias vectors according to the following update rule:

$$\boldsymbol{W}_i \leftarrow \boldsymbol{W}_i - \alpha \nabla_{\boldsymbol{W}} J(\boldsymbol{W}, \boldsymbol{b}), \qquad i = x, h, y$$

$$\boldsymbol{b}_k \leftarrow \boldsymbol{b}_k - \alpha \nabla_{\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b}), \qquad k = h, y$$

5.4.3 Vanishing & Exploding Gradient Problems

Vanishing and exploding gradients is a problem associated with DNN and hence also in RNN. It might be a bigger cause for concern in RNN because it often has long sequences of temporal layers which causes the signals to either converge exponentially towards 0 or diverge exponentially towards ∞. Considering figure 13, the gradient of the loss function associated with the hidden unit at time $t - 5$ is

$$\frac{\partial J}{\partial \boldsymbol{h}_{t-5}} = \frac{\partial \boldsymbol{h}_{t-4}}{\partial \boldsymbol{h}_{t-5}} \cdot \frac{\partial \boldsymbol{h}_{t-3}}{\partial \boldsymbol{h}_{t-4}} \cdots \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \cdot \frac{\partial J}{\partial \boldsymbol{h}_t}. \tag{16}$$

When the gradients are small, in particular less than 1, the gradient signal gets smaller and smaller as the gradients are backpropagated. This is referred to as the vanishing gradient problem. On the contrary, if the gradients are greater than 1, the gradient signal will grow bigger and bigger through time and in the end explode, the exploding gradient problem. The implication of this is that either, the weights far back will almost not get updated or they might get too powerful of an update.

5.5 Long Short-Term Memory Neural Network

It was briefly discussed in the previous section that DNN and hence RNN often suffer from the vanishing or exploding gradient problem. Intuitively, the problem is that the networks relies on many multiplicative updates and thus the network is good only at learning over short sequences. This means that the network might have a good short-term memory but its long-term memory is prone to be insufficient. This is the motivation for using the Long Short-Term Memory (LSTM) neural network architecture which tries to strengthen the long-term memory within the network.

Figures used in this subsection is from Understanding LSTM Networks (2015).

5.5.1 The Basic Architecture of LSTM

From the previous section, we saw that the RNN was simply an unfolding of a single hidden layer neural network over all time steps in the sequence. The general architecture in a LSTM consists of 4 unfolded hidden layer neural networks which is used to achieve long-term memory within the network We intro-

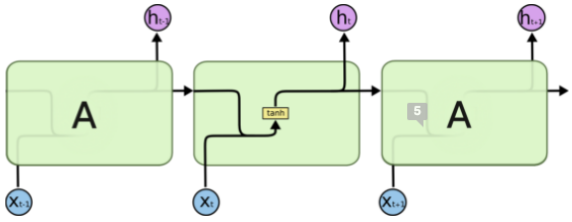


Figure 14: RNN

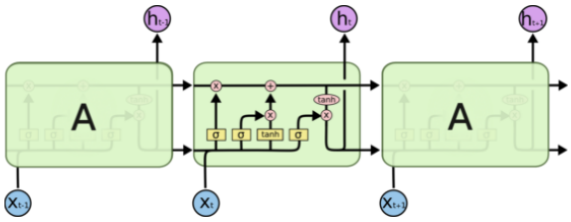


Figure 15: LSTM

duce the cell state vector, $\boldsymbol{c}_t$. Furthermore, we introduce three distinct gates: *forget gate* $\boldsymbol{f}_t$, *input gate* $\boldsymbol{i}_t$ and *output gate* $\boldsymbol{o}_t$. These gates have three distinct purposes, which will be explained in the following

run-through of a forward pass within an LSTM-layer.

First the hidden state at time $t-1$, $\boldsymbol{h}_{t-1}$, and the input at time t, $\boldsymbol{x}_t$ are fed into a neuron with a sigmoid activation function. In particular, the forget gate takes the form

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_f \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \boldsymbol{b}_f\right), \tag{17}$$

where $\boldsymbol{W}_f \in \mathbb{R}^{p\times(n+p)}$ is the weight matrix and $\boldsymbol{b}_f \in \mathbb{R}^p$ is the bias vector associated with the forget gate. The purpose of the forget is to filter out information that is no longer considered important. It should be noted that the sigmoid function is continuous. Therefore, the output is more akin to an estimated probability of whether or not information should be remembered. Next, the input gate is defined similarly to the forget gate, but with two new parameters

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_i \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \boldsymbol{b}_i\right), \tag{18}$$

where $\boldsymbol{W}_i \in \mathbb{R}^{p\times(n+p)}$ is the weight matrix and $\boldsymbol{b}_i \in \mathbb{R}^p$ is the bias vector associated with the input gate. The purpose of the input gate is to spot information that can be useful for the network at the current time step and also future time steps. The input gate is filtering information from the temporarily cell state which takes the same form as the forget and input gate but instead of applying the sigmoid function it uses the tanh activation function

$$\tilde{\boldsymbol{c}}_t = \tanh\left(\boldsymbol{W}_{\tilde{c}} \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \boldsymbol{b}_{\tilde{c}}\right). \tag{19}$$

Here $\boldsymbol{W}_{\tilde{c}} \in \mathbb{R}^{p\times(n+p)}$ is the weight matrix and $\boldsymbol{b}_{\tilde{c}} \in \mathbb{R}^p$ is the bias vector associated with the temporarily cell-state. We now have everything that we need in order to construct the cell-state at time t

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}, \tag{20}$$

where $\odot$ is the Hadamard product. It can be seen that the first term is about whether or not to reset or forget information from the former cell-state while the second term is about what to increment the

cell-state by. This way, a selective long-term memory capable of being updated by new information is achieved. Having computed the cell state for time step t, the next step is to calculate the output gate

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_o \begin{bmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \end{bmatrix} + \boldsymbol{b}_o\right), \tag{21}$$

where $\boldsymbol{W}_o \in \mathbb{R}^{p\times(n+p)}$ is the weight matrix and $\boldsymbol{b}_o \in \mathbb{R}^p$ is the bias vector associated with the output gate. The purpose of the output gate is to decide if the current cell state should be leaked into the hidden state which is clear from the computation of the hidden state at time step t

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh \boldsymbol{c}_t. \tag{22}$$

Putting the cell-state $\boldsymbol{c}_t$ through a $\tanh$-function ensures the result from the Hadamard product is bounded, particularly in the interval $[-1; 1]$.

5.5.2 Why LSTM Alleviates Vanishing/Exploding Gradient Problem

To understand how the LSTM succesfully reduces the exposure to the vanishing and exploding gradient problem, it is convenient to look at a simple LSTM with a single layer and $n = 1$. In this case, the cell state update simply is

$$c_t = c_{t-1} \cdot f + i \cdot c.$$

From this it is obvious that $\frac{\partial c_t}{\partial c_{t-1}} = f$ meaning that the gradient flows for c_t simply will be multiplied with the forget gate, *f*. Note that because the update is performed with the Hadamard product when $n > 1$, this result generalizes to higher dimensions. The forget gate might vary from time to time which can reduce the magnitude of the vanishing gradient problem. Furthermore, it is possible to express the hidden states in terms of the cell states as $h_t = o_t \cdot \tanh(c_t)$. From this it is possible to compute the partial derivative with respect to h_t with the use of a single $\tanh$ derivative.

5.6 Convolutional Neural Network

Convolutional neural networks have achieved great success in the field of computer vision tasks such as object detection and facial recognition. CNN's achieve their state of the art performance by learning

features from the input data. When training a CNN on a large amount of images with the object to classify according to a set of specified classes, the CNN learns high and low level features to distinguish the classes. This motivates the use of training a CNN on stock data, as one could imagine that the CNN can learn features pertaining to how the assets' correlate. This section is taking inspiration from Aggarwal

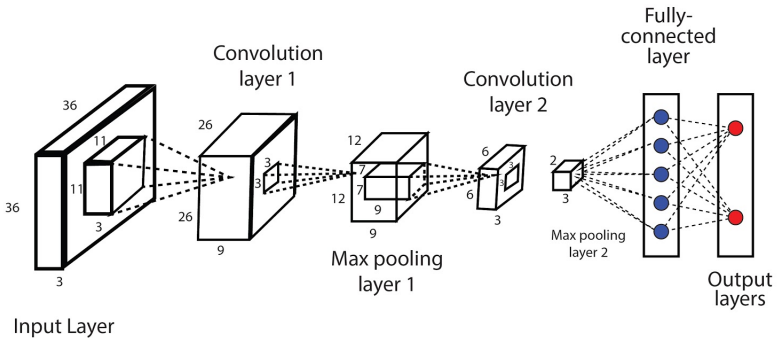


Figure 16: Example of a classic Convolutional Neural Network

(2018) and various figures is from this book as well except from figure 16 which stems from T. Balodi (2019).

5.6.1 The Basic Architecture of Convolutional Neural Networks

CNNs have three main types of layers; convolutional layers, pooling layers, and fully-connected layers. The convolutional layer is the first layer of the network, which is then often followed by a pooling or another convolutional layer, while the fully-connected layer comes last as seen in figure 16. Layer by layer the CNN becomes more complex and thus more capable of identifying more complex traits in the data.

The convolutional layer consists of input data, a filter and a feature map. A filter with size $F_q \times F_q \times d_q$ slides through each possible position taking the dot product between the filter and the portion of the input

covered by the filter. Note that the filter is 3-dimensional which means that the filter contains d_q different 2-dimensional filters that are applied to the input data. The convolutional operations are illustrated in figure 17. The filters are trained to detect different patterns i.e. feature maps. The complexity of the feature maps rises the deeper we get into the network. If a CNN was trained to detect human images, the first filters the network learns are often features like horizontal and vertical edges which evolves into more complex features such as a human eyebrow or nose. After all outputs have been calculated to the feature map a ReLU activation function is applied to introduce non-linearity. Following the convolutional layer, a pooling layer often follows. The pooling layer performs dimensionality reduction by reducing the spatial size of the previous layer's output. This is achieved by sweeping a filter across the previous layers output which applies an aggregation function to the cells within the regions. More on pooling in section 5.6.4. After the input data has been through all the convolutional and pooling layers, the data structure is flattened to a vector which is then fed into the fully-connected layer which is often a regular NN but can potentially be a RNN, LSTM or something third.

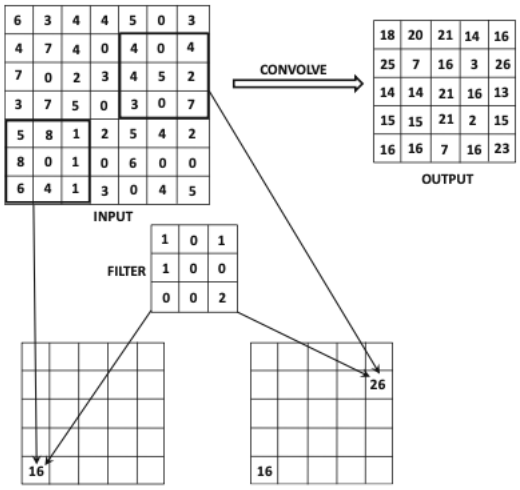


Figure 17: Convolutions

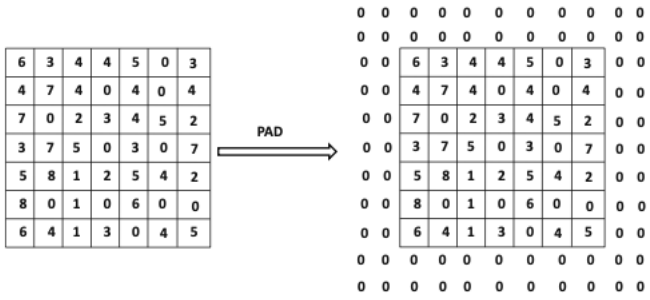


Figure 18: Padding

5.6.2 Padding

Applying convolutions reduce the size of the $(i+1)^{\text{th}}$ layer when compared to the i^{th} layer. This can be a problem due to potentially important information being lost along the borders of the input. Padding can

solve this problem by adding extra "pixels" around the borders so that the output from the convolution has the same dimensions as the input. The extra pixels are distributed evenly around the borders. If the convolution uses a $(F_q \times F_q)$-filter, the padding adds $(F_q - 1)/2$ extra rows/columns of pixels around the borders. The padding is 0's, so that it has no influence on the dot products. This type of padding is called *half-padding* (or same padding as in TensorFlow). No padding is then called *valid padding*. In our case where the "pixels" are prices of assets at different time steps, half-padding is absolutely necessary to avoid losing some information on "border assets", meaning the first and last asset. Without half-padding, we would also be throwing away some information from the first and - more importantly - the last prices.

5.6.3 Strides

When moving a filter over the data, it is not necessary to only move one position at a time and perform a convolution. Instead, it is possible to choose a *stride* $S_q > 1$, meaning the filter moves more than a single column/row at a time. Higher strides will reduce the spatial dimensions of the feature maps. Reducing the spatial dimensions also means fewer parameters in the next layer. Thus, a higher stride can reduce the risk of overfitting. However, it is also possible that some information will get more obscured. There is therefore a tradeoff between higher and lower strides. It is most common to simple use $S_q = 1$ or sometimes $S_q = 2$. While strides can be tuned in convolutional layers, they are also tuneable in pooling layers.

5.6.4 Pooling

The *max-pooling* operation uses small grid regions of size $P_q \times P_q$ on each d_q feature map of the layer. The maximum entry of the grid region is then returned to a new feature map. As this is done for each feature map, the depth stays the same. Therefore, the resulting layer has the dimensions $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$. However, a stride larger than 1 is often used in pooling, resulting in a layer with dimensions $((L_q - P_q)/S_q + 1) \times ((B_q - P_q)/S_q + 1) \times d_q$. Typically, $P_q = 2$ and $S_q = 2$ is used, meaning no overlap between the regions being pooled. The dimensionality is thus reduced from the feature maps whilst hopefully still retaining the relevant information. This way, potential noise in the data is reduced. Furthermore, this gives a form of translation invariance. A general problem with

regular neural networks is that they are heavily dependant on the particular position of a group of pixels. Translation invariance means that the CNN is capable of learning groupings and structures of pixels irrespective of their particular position. Usually, convolution and pooling layers are interleaved, meaning after each convolution layer there is a pooling layer.

5.6.5 Training a Convolutional Neural Network

Again, backpropagation is used to train a CNN. The last part of the CNN is the same as a traditional neural network. Therefore, backpropagation for training the weights and biases in that part of the CNN is the same. For the convolutional part of the network, we have two types of layers: convolutions and pooling layers. In a convolution, the value of a cell is a linear transformation of the elements of the grid covered by the filter in the former layer. In the case of overlap, a cell can contribute to multiple cells in the next layer. Let S_c denote the "forward set" of c, meaning the set of cells in the next layer being influenced by c. Then let w_r denote the weight of the filter element used feeding from c to $r \in S_c$. Then

$$\frac{\partial J}{\partial c} = \sum_{r \in S_c} \frac{\partial J}{\partial r} \frac{\partial r}{\partial c} = \sum_{r \in S_c} \frac{\partial J}{\partial r} \cdot w_r.$$

In the case of max-pooling, the partial derivative of the loss wrt. a max-pooled cell flow back to the cell with maximum entry in the grid being max-pooled. Meaning, the weight w_r is 1 if the cell is the maximum entry of the grid and 0 if it is not. The weight w_r can therefore be expressed using the indicator function $1_{\{c=r\}}(r)$ which gives 1 if $c = r$, meaning h is the maximum entry of the grid. And the function returns 0 if c is not the maximum entry. If c is a cell in the ith layer, then the loss gradients wrt. the weights between the $(i-1)$th and ith layer can be computed. Let A and C denote the set of cells connected by a weight w in the $(i-1)$th and ith layer respectively. Let $\forall j \in 1, ..., |A| : a_j$ be uniquely connected through w to c_j. Then

$$\frac{\partial J}{\partial w} = \sum_{j=1}^{|A|} \frac{\partial J}{\partial c_j} \frac{\partial c_j}{\partial w} = \sum_{j=1}^{|A|} \frac{\partial J}{\partial c_j} \cdot a_j.$$

This means that we pretend a weight of a filter element is different depending on which cells they connect even though it doesn't differ. The gradients are then summed to get a unified gradient for the weight. This is reminiscent of BPTT where we pretended the weights were distinct for different temporal layers.

6 Reinforcement Learning

This section will help give a basic understanding of the concept of reinforcement learning and then further expand on policy gradient (PG) a model-free policy optimization method which will be used in this project. Along with a motivation for the choice of this method the section will also elaborate on how to incorporate PG. The section takes inspiration from Sutton and Barto (2018), chapters 1 and 13, Dixon et al. (2020), chapter 9, Silver (2015) and Jiang et al. (2017).

6.1 Introduction

Reinforcement Learning aims to teach an agent how to act in different situations in order to maximize a numerical reward. The *agent* is not told what *actions* to take and has to learn what actions lead to what *rewards* given the *state* of the *environment*, giving the agent a clue to the effectiveness of the action in the situation. The actions are determined using a *policy*. As illustrated in figure 19, the RL agent enters

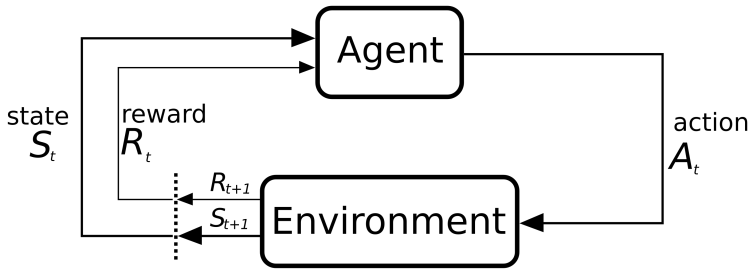


Figure 19: Reinforcement Learning Environment

into a finite or infinitely loop where it observes some state from the environment, evaluates the state and takes some action following some policy, meaning a way to determine what actions to take from a given state. The environment then gives the agent a reward conditioned on the action it took and the state it ends up in. Examples of environments in which this loop is finite are, the multi-armed bandit problem where the game ends if the agent goes bust or some kind of video game where the loop ends if the agent dies or wins. An example of a loop that never ends is the problem that are dealt with in this project, the asset allocation problem within the financial markets that naturally never ends.

RL has been used for a wide variety of tasks with great success in various different domains. RL has been used to create the strongest Go-computer, AlphaGo. Here, the state is the board position, the *action-space* is the set of possible moves for the computer to play. In this case, a reward is given at the end of the game based on whether or not the agent lost or won. The agent still needs rewards after a state is drawn from the *state-space* by the environment from the agent's action. With AlphaGo, this was done by assigning values to states by a *state-value function*. This way, it is possible for the agent to judge the value of an action, determined by an *action-value function*. In a similar way, RL is used to develop self-driving cars, even though the problem has both discrete and continuous state-spaces and action-spaces while they were discrete for AlphaGo. RL has been shown to work on a variety of domains with vastly different types of environments, state-spaces, and action-spaces, discrete, continuous or a mix of both. It makes sense to use methods from RL for determining an optimal portfolio policy to see if the success of the implementation in other domains translate to problems in finance.

Many quantitative investment strategies involve predicting future returns to guide allocation efforts. The prediction of future returns is a middle step and the models are not fitted with the resulting allocation in mind. Using RL instead, the model is fitting the portfolio policy directly, making predicting returns or estimating correlations unnecessary.

6.2 Reinforcement Learning Framework for Optimal Portfolio Policy

The environment is the series of the assets' prices, $\boldsymbol{x}_t$, and returns, $\boldsymbol{r}_t$. The state at time t, $\boldsymbol{S}_t = \boldsymbol{s} \in \mathcal{S}$, is a price matrix consisting of the last T price vectors; T being a tuneable hyperparameter. This state is used as input for our deep reinforcement learning agent which, through learning, determines a deterministic policy, $\boldsymbol{\pi_\theta}(\boldsymbol{s}) : \mathcal{S} \to \mathcal{A}$, mapping states to an action, $\boldsymbol{A}_t = \boldsymbol{a} \in \mathcal{A}(\boldsymbol{s})$, where $\mathcal{A}(\boldsymbol{s})$ is a continuous action-space of all the possible non-leveraged, long-only allocations between the assets. This is due to the limitation of the softmax function that can only output a vector of elements with values between 0 and 1, summing to 1. However, the softmax function is still the most suitable out-of-box activation function for our problem. The action at time t is therefore an n-dimensional vector of the allocation between the

assets, e.g. 0.2 (20%) in asset 1, 0.3 in asset 2, etc. The reward, R_t, from the environment triggered by the agent's actions is what controls the goal of the DRL agent. In this project, the reward will be given by

$$R_t(\boldsymbol{a}_t, \boldsymbol{r}_t) = \log(1 + \boldsymbol{a}_t^T \boldsymbol{r}_t) \tag{23}$$

In principle, we want the cumulated reward to be the total return over b periods. In reinforcement learning, we usually set the cumulated reward to be the sum of rewards in each time step in b periods. So the reward is set to be $\log(1 + \boldsymbol{a}_t^T \boldsymbol{r}_t)$ instead of the simple return. The cumulated reward of the next b periods from time t, G_t, is then

$$G_t(\boldsymbol{s}_t, \boldsymbol{a}_t, ..., \boldsymbol{s}_{t+b}, \boldsymbol{a}_{t+b}) = \sum_{i=t}^{t+b} R_t(\boldsymbol{a}_i, \boldsymbol{r}_i) \tag{24}$$

This is equivalent to log of the total return over the next b periods. Since the DRL agent aims to maximize this cumulated reward, and the cumulated reward is a monotone transformation of the total return, the DRL agent learns to maximize the total return over the next b periods. Usually, the cumulated reward would be the sum of discounted future rewards. However, since our time steps are supposed to be daily, a discounting factor doesn't seem suitable and has therefore been set to 1. In our setup, b is the size of our time window in which the DRL agent aims to maximize G_t. Therefore, the DRL agent learns to map price matrices to asset allocations by a policy in order to maximize the cumulated log-returns over b days. In this project, the optimal policy is approximated using a DNN, fitted using gradient ascent. This makes it a policy gradient method.

6.3 Policy Gradient

In PG, gradient ascent is used to update parameters $\boldsymbol{\theta}$ of our deterministic policy function $\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}_t)$ to maximize cumulated reward, using the gradient of the cumulated reward function. Some RL methods deal with approximating the action-value function. These methods can be cleverly used when there is no immediate external reward for an action. In our case the action-value function is simply the return from one time step to the next. Thus, the RL agent would be trying to estimate the expected return at every time step given the time series which is a problem already extensively researched as mentioned the section about Markowitz. An advantage for using PG might simply be that it is easier to approximate a policy function rather than a action-value function.

In our setup, PG is an off-policy method, meaning the states are not generated using a policy it needs to fit. With on-policy models, data is generated from a policy and the on-policy model then tries to approximate this policy. Off-policy methods usually also have data generated from a policy but they do not try to fit this policy. Instead, they try to determine an optimal policy given the data generated by the other policy. Since the actions of agents have no influence on states in this setup, it does not make sense to talk about having a policy generating the data. Without any such policy to approximate, the method cannot be called an on-policy method and is thus off-policy.

Usually, PG is used to determine a stochastic policy function where the output is a vector of probabilities of performing certain actions. An action would then be sampled using those probabilities. This is the case when the action-space is discrete. If it is continuous, you would usually approximate optimal parameters for a chosen distribution to choose actions from. However, in our case, the output is a vector of allocations rather than probabilities, making the method a deterministic PG. This also means there is no *exploration-exploitation trade-off*.

Referring back to the multi-armed bandit problem, exploration would refer to trying new machines in case one of the ones not yet tried turns out to be superior. Exploitation is then taking advantage of having found an optimal - or at least a good - machine. This can be done by what is called an ε-greedy policy; a policy that explores something different with probability ε and exploits otherwise. Normally, a stochastic PG would have this trade-off internalized, meaning an explicit ε is not needed. The agent balances exploration and exploitation on its own. Using a deterministic PG loses this feature. Luckily, the exploration-exploitation trade-off is only a consideration in problems where the agent has an influence on the environment and resulting states. In our setup, the agent has no such influence. Thus, a deterministic PG makes sense for our problem and the exploration-exploitation trade-off is not taken into consideration.

Normally, ensuring improvement of a policy by changing its parameters using PG would seem problematic due to the parameters affecting both actions and distribution of states where the state distribution is often unknown. This is actually not a problem due to the *Policy Gradient Theorem* showing the gra-

dient of performance wrt. the policy parameter does not involve the state distribution. In our case, the policy gradient theorem is not needed since actions do not affect state transitions.

6.3.1 Training using Deterministic Policy Gradient

Gradient ascent using the cumulated reward is equivalent to doing gradient descent using the negative cumulated reward. A trick that can be used here is to set the loss at time t to be the negative reward at time t

$$\mathcal{L}_t(\boldsymbol{\theta}) = -R_t(\boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}_t), \boldsymbol{r}_t). \tag{25}$$

Using SGD and setting the batch size to b, we end up with a total cost function

$$J_{t,b}(\boldsymbol{\theta}) = \frac{1}{b}\sum_{i=1}^{t+b} \mathcal{L}_i(\boldsymbol{\theta}) = -\frac{1}{b} G_t(\boldsymbol{s}_t, \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}_t), ..., \boldsymbol{s}_{t+b}, \boldsymbol{\pi}_{\boldsymbol{\theta}}(\boldsymbol{s}_{t+b})). \tag{26}$$

The parameters are then updated by

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J_{t,b}(\boldsymbol{\theta}), \tag{27}$$

where α is the learning rate. This is done due to TensorFlow using gradient descent as opposed to gradient ascent.

6.4 Implementation using Deep Neural Networks

Usually, SGD is simply used to improve training efficiency, i.e. choosing a batch size smaller than our training data, resulting in more updates per epoch. However, setting $J_t(\boldsymbol{\theta})$ this way results in the batch size b being a hyperparameter, controlling the size of the window over which the DRL agent attempts to maximize total return. The optimal deterministic policy function in PG is approximated using the DNNs RNN, LSTM, and CNN where $\boldsymbol{\theta}$ is the vector of all the weights and biases of the network. The loss function is set to be equation (25) so that when using SGD, the resulting cost function to be used for the update is equation (26). This way, the DNN approximates an optimal deterministic policy function for maximizing total return over a chosen period. That period being given by the batch size b.

The weights of the networks are initialized using a standard Glorot initialization scheme while biases

are initialized to 0. The starting policy is thus random. Usually with classification, this results in each classification being more or less equally likely. It would therefore be expected that the starting policy more or less chooses an equal weighting between the assets every time. After the first batch, it will update the parameters using equation (27). The policy from the new parameters are then used for the next batch and is then again updated in the same manner.

7 Implementation

7.1 Number of Assets and Lags

For this project, we have chosen the number of assets to be 10. As explained in section 4.5, a usual problem with using ML in finance is the dimensionality of the feature space relative to the number of data points. Although using RL makes it unnecessary to estimate expected returns and covariance matrix of returns, it does not fix the problem of having too large a feature space. We wanted to have a sample size in the thousands, corresponding to years of daily data. Thus, we chose a feature space with dimensionality orders of magnitude smaller than the sample space dimensionality. For real data, this could be achieved by having the assets be portfolios each based on different factors, sectors, or something along those lines.

Another consideration is the number of lags to use. Every model used in this project uses a number of lags. This is explicit in RNN and LSTM. In CNN, the dimensionality of the ”image” is determined from the number of assets - in this case 10 - and the number of lags. We have chosen the number of lags in this project to be 10. We wanted a number large enough to in principle capture daily and weekly trends whilst being small enough to not try and fit for trends on a larger scale. This would also increase the time needed for training considerably. For the particular number of lags, we simply chose it to be the same number as assets to have a square ”image” for the CNNs. It is a typical setting to have the image be square for CNNs, see 8.2.3 in Aggarwal (2018).

7.2 Benchmarks

We use the Equal Weighted Portfolio (EWP) as benchmark where, as the name implies, every asset has an equal weight in the portfolio. In our data, we have chosen to have 10 different assets, meaning the EWP is a vector of size 10 with

$$\boldsymbol{a}_{EWP} = \mathbf{0.1}.$$

The effectiveness of the different allocation policies are measured on three metrics: total return, Sharpe ratio and Sortino ratio. The Sharpe ratio is calculated as

$$Sharpe = \frac{TR}{s_r},$$

where TR is the total return, and s_r is the sample standard deviation of the returns of the allocation policy. The idea behind the Sharpe ratio is to calculate the total return per unit of risk. A problem with the Sharpe ratio is the use of the standard deviation of returns as a measure of risk. In principle, if an allocation policy always gets great positive returns but the returns themselves vary greatly, the Sharpe ratio can be sub-par, even though anyone would call the allocation policy a fantastic one. Therefore, we also look at the Sortino ratio where the sample standard deviation of the negative returns are used instead of the sample standard deviation of returns used for the Sharpe ratio. It should be noted that the total returns, Sharpe ratios, and Sortino ratios in no way correspond to what could be expected on real world data. Only the relative values of the allocation policies on those metrics should be used. In practice, the different metrics are usually scaled for appropriate periods, i.e. a yearly Sharpe ratio. Since we are only using the metrics to compare methods, and the methods all allocate in the same time periods, no such scaling is necessary.

Using data simulated from a model also allows us to use our knowledge of the model to analyze the allocations from the policies. Generally, we want allocations that put less weight on assets with negative mean return. Furthermore, it is desirable for the top weighted assets to be negatively correlated. This will reduce the overall risk, defined as the return variance of the portfolio. All in all, we look for allocations that strikes some kind of balance between maximizing expected return and reducing the risk of the portfolio.

7.3 Choice of Seed

Since training and tuning various models takes considerable time, we chose one seed to tune hyperparameters for. Using this seed, we determine combinations of network topologies, data inputs and validation metrics to use on other seeds. In partilcular, we chose seed 1, see figure 2 in section 3.3. In the real world, different asset classes, countries and sectors can have vastly different underlying structures. Testing for transferability of combinations of network topologies, data inputs and validation metrics on different seeds could therefore prove interesting. It could indicate that it is possible to determine network topologies that work on different financial price series.

The seed was chosen arbitrarily, though we decided to avoid seeds deemed too ”unrealistic”. This was done by looking at the price series generated by the given seed. A number of seeds had too many asset prices going towards 0 or had a single asset vastly outperforming the others, see figure 20. This indicates

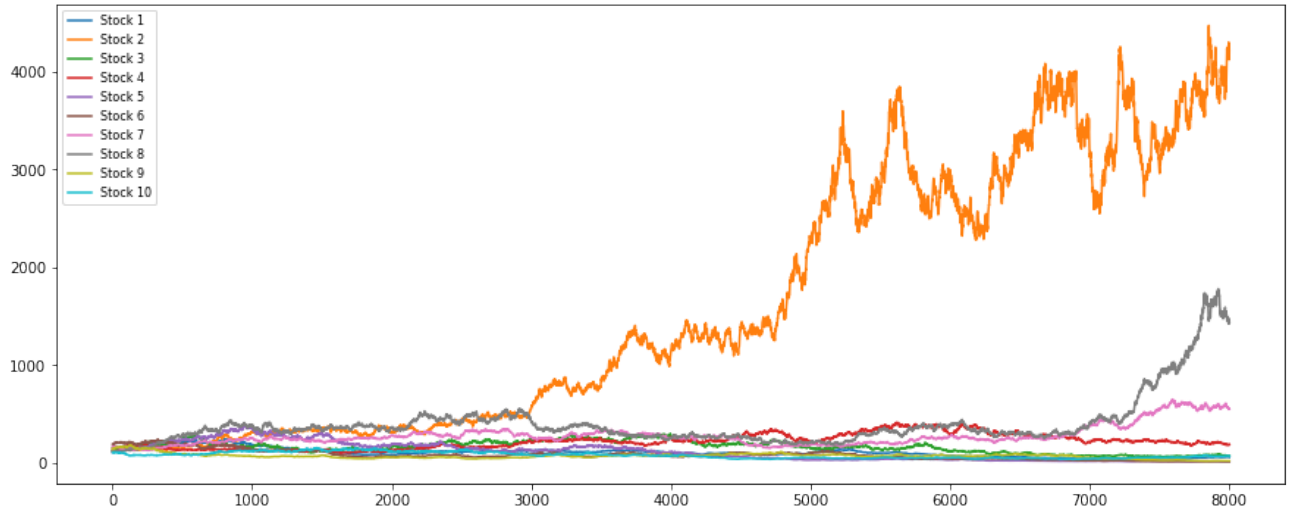


Figure 20: Price series generated from seed 3

that the expected returns for each day were too close to 0 relative to the noise for those assets converging to a price of 0. As noted in section 3, we try to choose a $\boldsymbol{\mu}$ so as to avoid this. However, this does not necessarily prevent the prices from going towards 0. Further tuning of the distributions the models are drawn from would be needed to improve the simulation of data generated from the seeds. This would take some trial and error. Since this project is about the efficacy of the PG method for approximating an

optimal portfolio policy on simulated asset data, it was simpler to sift through ”bad” seeds than to tune the distributions the model is drawn from. This does mean we have some jumps between chosen seeds. See the seeds chosen in table 4 in section 8.

7.4 Training Models

The data is split into training, validation, and test data like explained in section 4.4. The training data is used to train the models. The validation data is used for tuning hyperparameters. The hyperparameters resulting in the model with best performance on the validation data is then tested on the test data for the final performance measurement. Different metrics are chosen for validating models. For every relevant metric to measure performance on, we validate models on them, i.e. we choose the model with best total return on validation data, and we choose the model with the best Sharpe ratio, etc. The chosen validation metric can have great impact on the performance on the test data.

It should also be noted that a model validated on total return will not necessarily have a better total return on the test data as opposed to a model validated on their Sharpe or Sortino ratio. For the performance on test data, we look at all three metrics. We will generally be sceptical of a model measuring high on total return whilst scoring low on Sharpe and Sortino. This would indicate the allocation policy is simply less risk averse than the others and is being compensated for that extra risk. And while we will generally weigh Sortino higher than Sharpe, we will still take both into account when looking at the relative effectiveness of the different methods.

To look for potential over- or underfitting, we can look at the loss curves. Training on both training and validation data after validation makes it impossible to create loss curves for the training. Instead, we have to extrapolate from the loss curves from training solely on the training data and validating on the validation data. Furthermore, in our case of RL for asset allocation, we can also take a look at the allocation vector. An agent have clearly overfitted on training data, if it always allocate to a single stock. On the other hand, if the allocation is almost always the same as the EWP, the agent might be underfitted. When training on the combined training and validation data, we also use the allocations from the policy

to get a feel for whether or not the model is over- or underfit. In this case, we do not have any loss curve to use. We use L2 regularization and dropout fairly aggressively in order to not end up using early stopping. We do this because a validation loss curve is needed for early stopping. Since early stopping is therefore not really possible, the other regularization methods need to be that much more aggressive. This also means that it generally would not hurt us to use more epochs than we do.

The number of epochs are more so chosen from a time-return tradeoff perspective, meaning whether or not the extra time needed for training on more epochs is worth the resulting lower loss. Due to the size and complexities of the models and the fact that we wish to test on different seeds, we usually end up stopping at an epoch that would be before the one dictated by early stopping. However, it mostly looks like the validation loss curve converges to the training loss curve, flattening out. Therefore, we have judged that the substantial extra time needed for training the models are not worth the seemingly small improvement to loss.

7.5 Hyperparameter Tuning

Tools exists for automating hyperparameter tuning. We have chosen to use the Python hyperopt package with its fmin-function. This function allows us to set a hyperparameter-space; a space of potential hyperparameters for it to try. It searches using the Tree-Structured Parzen Estimator (TPE) algorithm that first randomly chooses different combinations of hyperparameters to train with and then bases future hyperparameter selections on how well the other hyperparameter combinations worked. We set the distribution of the different hyperparameters for the algorithm to draw from. We have chosen to exclusively use continuous or discrete uniform distributions for this. We do not see any particular reason to prioritize certain ranges of hyperparameters so uniform distributions make sense from this perspective.

To limit the search space somewhat, we have chosen to keep the network topology constant, i.e. the same layers with the same number of hidden nodes in those layers (though it can naturally differ from one layer to the next). The hyperparameters are therefore things like the dimensions of the filters in CNNs, regularization parameters, dropout rates, batch size, learning rate, momentum and whether or not

to use Nesterov with momentum.

7.6 Neural Network Topologies

Tuning for regularization parameters and dropout rates, automatic model selection can be performed meaning that being aggressive enough with these hyperparameters, we choose deeper and wider neural network topologies than we could otherwise. Yet, using regularization parameters and dropout rates are not all-powerful. If the model appear to be underfit, we increase the number of hidden nodes, add another layer, or increase number of epochs in training depending on the loss curves. If instead it appears to be overfit, we increase the bounds of the distributions L2 regularization and/or dropout rates are drawn from. This is under the assumption that the model is still feasible to train. Otherwise, the number of hidden nodes or layers are lowered instead. This has led to three different topologies for each of the network types.

The RNN topology consists of three RNN layers with 64 nodes in each RNN layer with a softmax layer on top. The LSTM topology consists of 3 LSTM layers with 64 nodes in the first LSTM layer, 32 nodes in the second layer and 16 nodes in the last layer, also with a softmax layer on top. The CNN topology consists of 3 convolutional layers, a flattening layer, a regular NN layer, and a softmax layer on top. The first convolutional layer has 16 filters of size (4×3) and a max-pooling layer of size (2×2). The second and third convolutional layers have 32 filters of sizes (2×4) and (4×2) respectively as well a max-pooling layer of size (2×2) after each of them. The regular NN layer has 32 hidden nodes.

7.7 Data Input

The models can use different types of input. They can use the price series or the return series. Furthermore, the values of those series can be standardized. We test the models on the standardized, and non-standardized price series as well as the return series. We do not standardize the return series since the returns are generally already around the same size, meaning a standardization will probably not do too much. The data input is then transformed into a series of input windows. This means that for an input

vector $\boldsymbol{Y}_t$, a matrix is created

$$\begin{bmatrix} \boldsymbol{Y}_t & \boldsymbol{Y}_{t-1} & \dots & \boldsymbol{Y}_{t-9} \end{bmatrix}.$$

This is the input matrix used as input for the RNNs, LSTMs, and CNNs. For CNNs, we also need to reshape the input matrix to be a $(10 \times 10 \times 1)$-tensor. When CNNs are used on images, each pixel contains a vector of colour values. The filters of the CNN are then used on each channel, meaning each colour. In our case, we only have the single channel. The number of channels still have to be explicit for a package like keras, making the reshaping necessary.

8 Results

In this section, the result of the models are presented and evaluated. The results of the models will first be compared to one another after which the performance of the best models will be compared to the EWP. Finally, this section will stress test the best models ability to perform on different seeds.

8.1 Comparison of Models

We chose seed 1 to be the seed generating the data for the test of the models. All of the models in this project have been trained on raw and standardized price data, as well as return data and have also been validated in order to maximize respectively the Sharpe ratio, the Sortino ratio and the total returns. Naturally, we want to assess all the combinations that the models were trained and validated on. When evaluating the models, we are looking for models that achieve a high total return while having high Sharpe and Sortino scores. We want to stress that if two models achieves similar returns, then the model that scores the highest Sharpe and Sortino will be preferred as this indicates that this model are more risk averse. We especially look for models dominating other models, i.e. models that have higher scores on all three metrics. Furthermore, when evaluating the models, we have highlighted with bold the best performing combinations of each network type. These models will later be compared to the benchmarks proposed in section 7.2. The results of the models are summarized in table 1.

It can be seen that validating for total returns does not result in substantially more risky models. Val-

	RNN			LSTM			CNN		
	Sharpe	**Sortino**	**Returns**	**Sharpe**	**Sortino**	**Returns**	**Sharpe**	**Sortino**	**Returns**
Validating for the Sharpe ratio									
Raw Data	46	78	0.24	**70**	**120**	**0.28**	62	107	0.23
Standardized Data	76	116	0.61	68	117	0.23	72	126	0.26
Return Data	25	43	0.25	36	62	0.23	67	116	0.23
Validating for the Sortino ratio									
Raw Data	**147**	**246**	**0.89**	62	107	0.21	64	111	0.25
Standardized Data	59	102	0.26	67	115	0.26	64	112	0.23
Return Data	81	138	0.50	57	98	0.26	**81**	**139**	**0.28**
Validating for total returns									
Raw Data	47	80	0.22	64	110	0.20	66	113	0.25
Standardized Data	56	97	0.36	54	93	0.18	68	118	0.24
Return Data	70	120	0.24	25	43	0.21	72	124	0.25

Table 1: Results of the models on raw, standardized and return data

idating for Sharpe- or Sortino ratio also does not result in substantially more risk averse models. This could be a sign that the models acknowledge that allocating too much to single assets is too risky, but weighing some towards assets with higher expected return is prudent.

It is interesting to note that the CNNs generally perform better when its inputs are returns rather than price data. It could be the case that the CNN has an easier time determining how the assets' returns are correlated by looking directly at the returns. It is also worth noting that the LSTM generally performed better on the data not standardized. Intuitively, LSTM would perform better on the return series, since it is stationary. For regression, they usually do not extrapolate well, meaning they work better on data it has seen before. The prices in the price series are not too dissimilar to prices it has seen before on the training or validation data. Thus, we would not expect it to have too big an impact on performance. However, one would still normally not expect it to perform worse on the stationary data.

It generally seems like validating for Sharpe- or Sortino ratios is preferable. It does intuitively make

sense that validating for returns could result in sub-par models due to too their resulting portfolio policies being too risky. It is easier for a model to be lucky with total returns on validation data than to be lucky on Sharpe- or Sortino ratios. Meaning, it is harder for a more generalized portfolio policy to outcompete a lucky model when the validation metric is total returns.

Some combinations of network, data input, and validation metric dominate other combinations by having higher total returns as well as higher Sharpe- and Sortino ratios. The best CNN was trained on return data validating for Sortino ratio. This CNN dominates any other CNN on seed 1. The same can be said for the LSTM trained on raw data validating for Sharpe ratio. And lastly, the RNN trained on raw data validating for Sortino ratio dominates all other RNNs. Furthermore, this RNN vastly outperforms any other combination of model, data input, and validation metric on all performance metrics.

8.1.1 Differences in Policies

One remarkable difference in the models' policies is whether or not their allocations change over time. We find that the RNN changes its allocations during the test period as opposed to the CNN and LSTM models which have constant or near constant allocations throughout the test period. In figure 21, the price series of the test data is plotted while figure 22 plots the allocations for the RNN model in the test period. From this, it is very interesting how the RNN model changes the majority of its allocation between

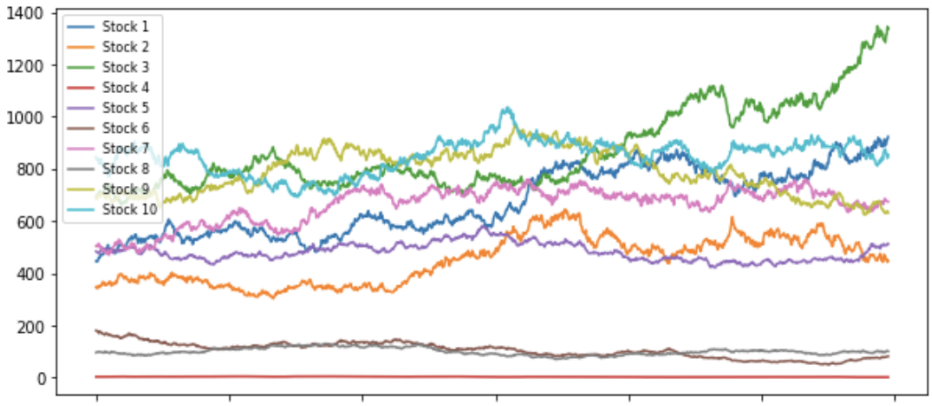


Figure 21: Price Series for the Test Data

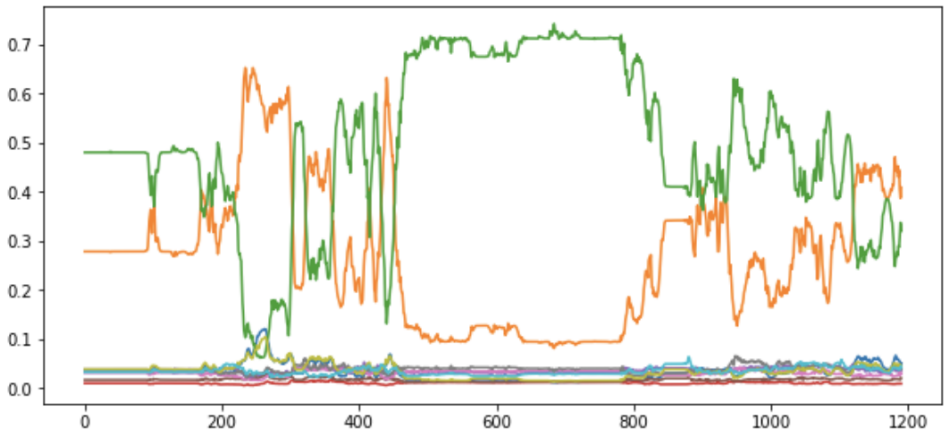


Figure 22: The change in weight allocations for the best RNN model

asset 2 and 3. Inspecting the conditional covariance matrix of the price series in appendix A.1, it can be seen that asset 2 and 3 are relatively highly negatively correlated compared to the other assets. Thus, it seems like the RNN has learned a policy that actively exploits this fact. It can be seen in figure 22 that if the model allocates a greater proportion to asset 3 then it offsets this by reducing the allocation in asset 2. Investigating μ in appendix A.3 shows that asset 2 has the second highest expected return of all the assets while asset 3 has the highest expected return of the assets that are negatively correlated to asset 2. In general, the policy learned with the RNN seems to result in allocations that make a lot of sense considering the model generating the data.

The fact that the RNN actively adjust its allocations in this manner is an indication that the policy learned from the RNN is fitted to the conditional distribution whereas the policies of the LSTM and CNN seems to converge toward a policy that is fitted to the unconditional distribution. The changing allocations in the RNN model might stem from the fact that the RNN suffers from the vanishing gradient problem and hence suffers a weak long-term memory. The vanishing gradient problem makes the RNN lose information pertaining to prices we know to not be part of the conditional distribution. On the other hand, it seems like the better long-term memory of the LSTM and the CNN makes them weigh data not part of the conditional distribution. This forces their respective policies to converge toward more constant allocations fitted for the unconditional distribution instead.

The chosen constant allocation of the CNN and the near constant allocations of the LSTM (see appendix B) do seem to make some sense looking at the model generating the data. Looking at $\boldsymbol{\mu}$, they generally weigh assets with negative mean returns less than the other assets. The top two allocations are also negatively correlated. Thus, the allocations do seem to make some sense wrt. the underlying structure of the data. Looking at $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}_u$ (see appendix A.2), it does seem like weighing asset 9 would make sense, since it has the second highest mean return and the second lowest variance of returns. However, it can be hard to say whether or not this would necessarily improve the performance. Looking at the model generating the data, the policies seem make some sense but they do not necessarily appear perfect.

We do want to stress that the differences in portfolio policies could be explained by other factors such as the topologies of the network, the hyperparameter tuning or something third. The differences in the models and their structure is one possible explanation and it does not seem unreasonable to believe those differences to be the cause.

8.2 Comparison to Benchmark

Until now, the models have only been evaluated relative to other models. In order to get a better understanding of whether the models achieve a performance that is noteworthy, they must be compared to portfolios that are not generated from closely related ML models. In this project, the best models are compared to the EWP as described in section 7.2. The EWP achieves a 17.7% return, with a Sharpe value of 32.6 and Sortino value of 52.5 on seed 1.

Since the CNN has constant allocations, we can create a Markowitz "Mirror" Portfolio (MMP) that has the same expected daily returns as the CNN agent. This is done using equation (2) from section 2.1. This portfolio would be a the theoretical optimal portfolio for getting the same expected daily returns as with the CNN. Thus, we would not expect to beat this benchmark. It is more so there to see how well the CNN has managed to minimize risk even without being able to short any assets. In figure 23, the best performing models of each type of network have been plotted against the EWP. The plot shows the

development in wealth, if an agent starts with 100 from the beginning of the test set. The clear conclusion

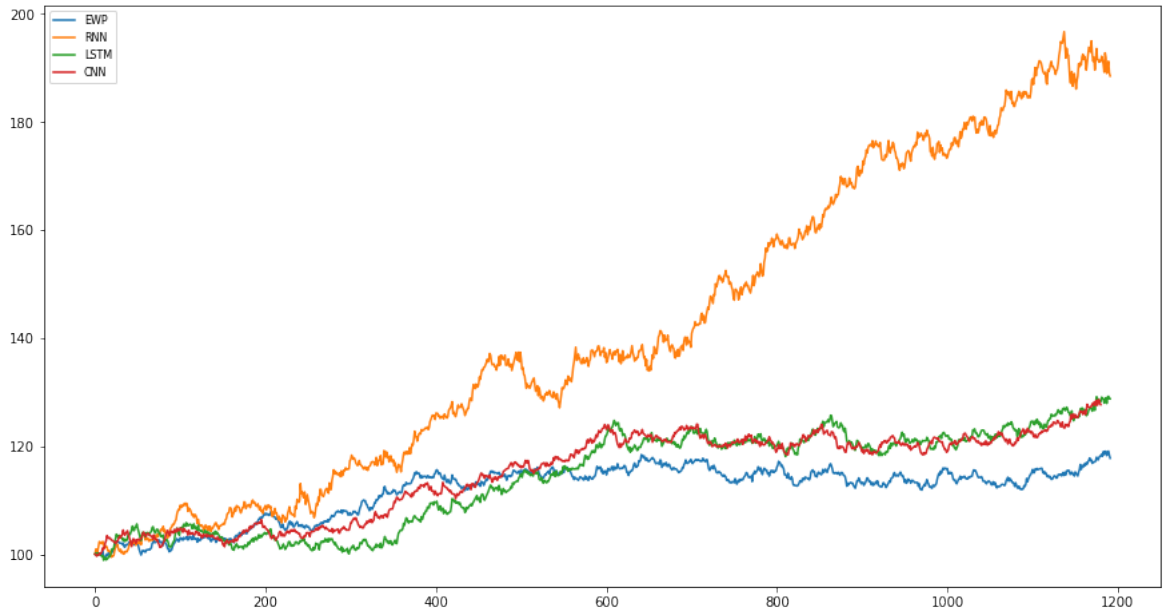


Figure 23: Comparison to Benchmark

is that all the models outperforms the equal weighted portfolio and that the RNN is the clear winner of all the models.

Model	Sharpe	Sortino	Returns
CNN	81	139	0.28
MMP	127	224	0.29

Table 2: Comparison of CNN to Markowitz

Looking at figure 24, the wealth of CNN is clearly more volatile when compared to the MMP. It should be noted that since the MMP is made to have the same expected daily return, comparing the total returns between the two models make little sense. Instead, we look solely at the Sharpe- and Sortino ratios. As can be seen in table 2, the CNN is clearly worse off. We would like to once again stress that the MMP is a theoretical optimum taking advantage of knowledge of the model that generated the data. Thus, we would not expect the CNN to outperform this benchmark. The CNN could perform worse because its impression of the unconditional distribution is imperfect, or it suffers from not being able to short any

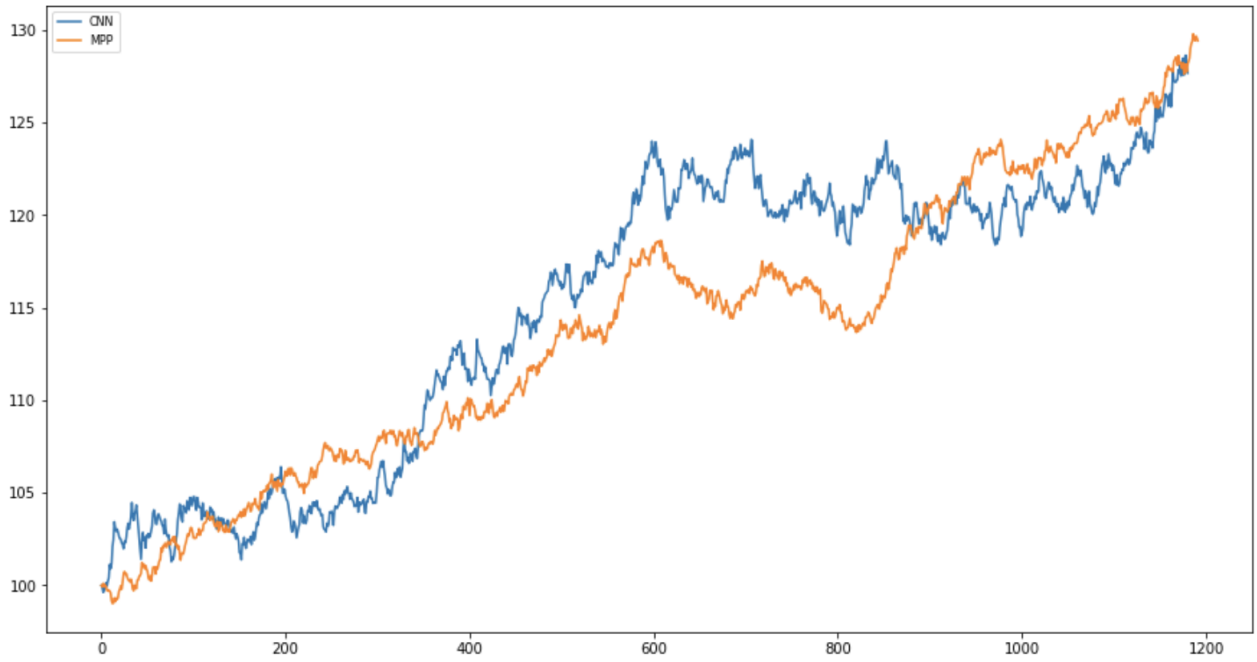


Figure 24: Comparison off CNN to Markowitz

assets, or a combination of the two. From table 3 it can be seen that the two portfolios are not too dissimilar. The biggest difference in the two portfolios is the MMP's ability to short assets. In section 9.6, we discuss possible ways to allow the DRL agent to be capable of shorting assets.

	1	2	3	4	5	6	7	8	9	10
CNN	0.093336	0.109347	0.141493	0.024361	0.133014	0.069969	0.099902	0.045589	0.086168	0.196821
Markowitz	-0.179927	0.112961	0.461220	0.104844	0.126382	0.007140	0.102186	-0.049165	0.147178	0.167181

Table 3: Comparison of the allocations of CNN and Markowitz

8.3 Test on different seeds

In this section, we wish to stress test the best performing models on different seeds. The motivation behind this is to see if the network topologies and hyperparameter spaces that were found to be the best on seed 1, also have the ability to perform well on other seeds with a different underlying data structure. Appendix C show the price series on the different seeds. Table 4 shows performance of the different

models on those price series. The results clearly indicate that the same topologies and hyperparameter

	ML Model	Training Data	Validation Metric	Sharpe	Sortino	Returns
Seed 20	RNN	Raw	Sortino	25	43	0.22
	LSTM	Norm	Sharpe	0.64	0.99	0.008
	CNN	Return	Sortino	110	193	0.46
	EWP			163	274	0.63
Seed 30	RNN	Raw	Sortino	-43	-71	-0.26
	LSTM	Norm	Sharpe	-18	-29	-0.10
	CNN	Return	Sortino	-24	-41	-0.08
	EWP			-10	-16	-0.04
Seed 43	RNN	Raw	Sortino	3	6	0.04
	LSTM	Norm	Sharpe	-6	-10	-0.06
	CNN	Return	Sortino	21	36	0.10
	EWP			28	48	0.13
Seed 59	RNN	Raw	Sortino	24	39	0.31
	LSTM	Norm	Sharpe	71	118	0.27
	CNN	Return	Sortino	53	89	0.30
	EWP			70	116	0.30
Seed 64	RNN	Raw	Sortino	23	38	0.17
	LSTM	Norm	Sharpe	32	51	0.15
	CNN	Return	Sortino	30	49	0.15
	EWP			30	50	0.14

Table 4: Results of the best performing models selected seeds

spaces do not necessarily work when the data is vastly different. It could be that the hyperparameter spaces and/or network topologies need to be altered depending on the data. It could also be due to the expectations for the returns changing too much from the training and validation data to the test data. In section 9.3, we discuss a training approach that could potentially alleviate this problem. Another possibility is that the models are not capable of seeing through the underlying VAR(1)-models of the various seeds. Further research is needed to determine whether or not altering the hyperparameter space and network topologies is sufficient for the models to perform across other various seeds.

9 Discussion

9.1 Reward & Loss Functions

One of the pros of using RL is the ability to generalize the optimization problem to include factors other than return. By choosing other reward functions, it is possible to take things like ESG into account, as well as other risk factors such as currency risk. This makes it possible to create an agent that in theory can be fitted to every individual investor's preferences. If an investor cares greatly about sustainability, you could include a term in the reward function that punishes portfolios with low scores on ESG-factors.

Another example could be to set the reward function to be the Sharpe ratio which would force the RL agent to learn how to create portfolios that takes the volatility into account. In order for this to work, it is necessary to specify an episode window to measure the volatility within the period. An example could be to choose an episode of 100 days after which the Sharpe ratio is evaluated and the model updates its parameters according to this. If an individual investor only want to minimize the downside volatility or the left tail of the return distribution, the reward function could be changed to the Sortino ratio. In reality, the reward and loss functions can take any ratio the investor want and as many as the investor want. Thus, if an individual investor want to create a portfolio that recognizes ESG factors but also maximizes according to the Sharpe ratio, this would be no problem at all.

A problem with the way we handle rewards is the fact that returns, and thus rewards, are stochastic. In other words, a good action does not necessarily result in a higher reward. A bad action can still result in a high reward. If the signal-to-noise ratio of the rewards is too low, the agent will learn policies that fit too much to the noise as opposed to the signal. A possible solution to this could be to use other RL methods that try to approximate better rewards.

9.2 Other Reinforcement Learning methods

We choose the PG method to directly approximate an optimal allocation policy but more indirect methods are also available. Action-value methods approximate a state-value function or an action-value function

instead of approximating a policy. In those methods, the state-value function is a function measuring the value of a certain state $\boldsymbol{s} \in \mathcal{S}$ under the policy π, $v_\pi : \mathcal{S} \mapsto \mathbb{R}$. However, this method can only be used in setups where the agent's actions have influence on the environment and thus the resulting state. This has not been the case in our setup where the agent has no influence on the environment. The action-value function, $q_\pi : \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$, is a function measuring the value of taking a certain action $\boldsymbol{a} \in \mathcal{A}$ at a certain state $\boldsymbol{s}$ under a policy π. These methods get simpler in our setup, again due to the environment not being influenced by the agent's actions.

Q-learning is, like our PG, an off-policy RL method, see figure 25 for an overview of RL methods. In Q-learning, the action-value function is approximated under a greedy policy. An agent with a greedy policy will try to maximize its reward for the next time step not taking future rewards into account. The approximated action-value function under a greedy policy, denoted $Q_t : \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$, in effect attempts to alter the reward so that a greedy policy will implicitly maximize for cumulated reward instead of just the next reward. The values of $Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ depend on future rewards, thus encoding this information into the function. This could potentially fix the problem of stochastic rewards. Intuitively, $Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ would attempt to determine expected reward (from expected returns). Thus, the agent would attempt to maximize expected rewards rather than actual rewards. This could help alleviate the problem when having a lower signal-to-noise ratio.

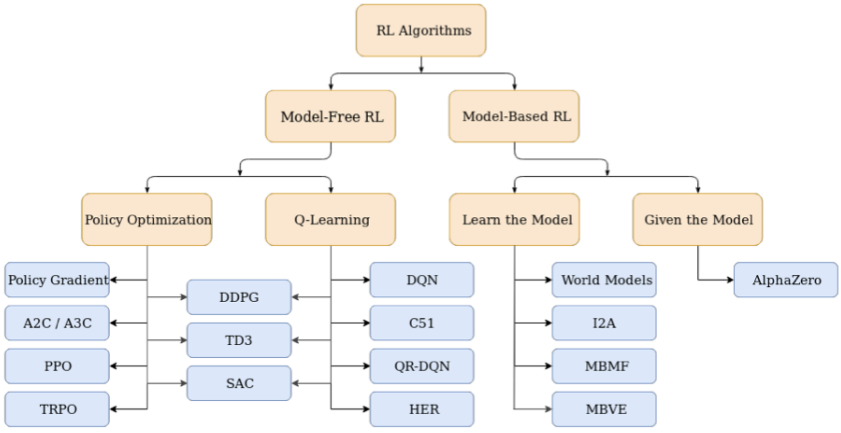


Figure 25: Map of RL methods

A problem with Q-learning is the fact that it is designed for problems with discrete action spaces. In this case, maximizing $Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t)$ wrt. $\boldsymbol{a}_t$ is simply going through all possible actions, picking the action that maximizes the approximated action-value function. This is not possible for continuous action spaces. One possibility could be to discretize the action space in order to be able to use the same approach. However, better methods are available such as Google DeepMind's Deep Deterministic Policy Gradient (DDPG). The DDPG combines Q-learning and PG in a way that works on continuous action spaces.

9.3 Online learning

One of the nice things about using DRL is the option of using online learning. This would mean the model trains on new data even after being trained on training data. This could potentially mean the model would be able to learn from more recent structures in the data and more easily disregard older structures. It could aid the models in determining a conditional distribution, changing allocations depending over time. Exactly how fast it learns is still adjustable through the learning rate. Through that, the model's adaptability is adjustable.

We can also use the notion of replay memory. In RL, this is usually done by storing transitions $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$. Due to the agent not being able to influence the environment and thus states, it is only necessary to store the states $\boldsymbol{s}_t$. To update parameters from new data, we would sample a mini-batch from this buffer and perform an SGD update. This could further aid the agent in discerning changes in structure (i.e. the conditional distribution) and adapt to those changes. Older data points would lose their influence on the approximated policy over time.

This approach could potentially fix the problem of the models vastly underperforming on certain seeds, especially where the conditional distribution might have changed drastically throughout the time series. It could help the models weigh older data less and more easily adapt to the new structure of the data.

9.4 Other Inputs

A natural next step would be to test the efficacy of the methods on real data. The results indicate that the methods with adequate tuning could be promising for real world allocation problems. The models seem to capture part of the underlying structure resulting in performant allocation policies. In this project, the inputs have only been the price or return series. The reason for this was that the data input originates from simulations. Thus, it is not possible to incorporate other possibly meaningful inputs. If this project were to be tried on real world data, it would be of great interest to try to investigate how the proposed models could handle other additional inputs rather than just the price or return series. This would probably require some more computational power than what has been available for this project. With only price or returns series as input, the computational power has shown to be a limiting factor in this project, forcing the models to be potentially less complex than what could have been desired.

One interesting type of input for the proposed models would be to feed them with the famous factors proposed by the Fama-French 3-factor model. The model includes the market β, the size premium, and the value premium. The market β captures information about the systematic risk associated with an asset. The size premium refers to the historical tendency of assets with a small market capitalization to outperform assets with a higher market capitalization. The value premium refers to the tendency of assets that seem cheap relative to some fundamental value to outperform assets that seems expensive relative to some fundamental value. Including such factors will give another dimension of information to the models which might help the models to figure out the underlying structure within the market.

It could also be interesting to incorporate economic factors. This could be things like inflation, unemployment rate, etc. In our model, the conditional distribution changes but the process converges to a stationary distribution. It is not unreasonable to think that certain macro economic factors could potentially alter the underlying structure between the assets. Analogous would be to alter the stationary distribution the process converges towards. Economic factors could potentially hold information the models can use to see through the structure of the market and allocate accordingly.

Though economic and financial factors can affect the market greatly, the public perception and opinion on those factors could affect it even more so. Sentiment analysis on the economy and the market done using modern natural language processing (NLP) methods could be fed as input to the agent. A combination of all three alternative input streams would result in an agent that takes the state of the economy, the market, and the public perception of the two into account.

9.5 Convolutional Neural Networks on Financial Data

We used CNNs due to their general superiority in determining structure through space. The concrete method they use does pose a new interesting ”problem”. The filters of a CNN used on images locate structures between neighbour pixels. The sequence of pixels in an image is locked in place and cannot be changed. In our case, the order of assets is not locked in place. This means CNNs could potentially fare better or worse on the same input data depending on the order of the assets.

It can be hard to determine the best way to order the assets before using a CNN. One way that could make sense is to do it by correlation. So a random stock could be chosen for the first one. For the second asset, the one most positively correlated with the first one would be chosen. Then, the third asset could be the one most correlated with the second asset of the assets left. This process would be continued until all the assets are ordered. In general, it could be interesting to investigate different methods of ordering the assets and how it would affect the performance of the CNNs. Potentially, the images resulting from the different ordering methods could be used as different channels for the CNNs.

Other possibilities could be to use financial factors as different channels. So for a certain stock, there could be a channel for each financial factor used in the model. If we do not believe there is any worthwhile information in the time series, i.e. we should only use the current prices, returns, or financial factors as input, then we could use the different lags as channels. In our project, this would make the ”image” one-dimensional. Though CNNs are usually used on two-dimensional images with colors, they can absolutely be used on one-dimensional ”images” with different channels.

9.6 Other Output Functions

We chose softmax as the activation function in the output layer due to it being the most natural out-of-box activation function to use. As mentioned in section 6.2, this comes with the limitation that the agent is not able to short any assets. If the agent is not able to short assets, its ability to minimize the overall risk off the portfolio will be reduced. If the agent is long an asset that it is confident will soar, it would be able to reduce the risk of the overall portfolio by going short in an asset that is positively correlated to the long position. Furthermore, not being able to short reduces the agents action space from which profitable strategies could emerge. Also, if the agent predicts that the market is falling, it would be a profitable strategy to simply short some of the assets by taking advantage of the fall in asset prices.

A way to circumvent this could potentially be to use the $\tanh$ activation function for each output node. This would allow the agent to short certain assets. A problem here is that there is no guarantee the resulting allocation would sum to 1. A solution could be to have a risk-free asset or cash-bias, and allocation to this asset would simply be so that the allocation weights sum to 1. The reason for using a risk-free asset instead of another risky asset is because the method would be introducing some asymmetry in the way asset allocation are approached, since the last asset is allocated to indirectly rather than explicitly. This asymmetry is already present with the risk-free asset, so the method would not introduce new asymmetry.

10 Conclusion

This project shows how DRL can be used to solve the optimal portfolio policy problem. This is achieved using DNNs in a RL framework using PG. This allows the model to learn a policy that at any time can generate a portfolio that fits an individual investor's preferences specified by the model's loss and reward function. We find that the various DRL methods are capable of solving the problem, outperforming the benchmark of the EWP. Generally speaking, the allocations seem to indicate that the models have a fair impression of the underlying structure of the data.

The CNNs and LSTMs determine policies that result in more or less constant allocations. This could

indicate that the CNNs and LSTMs determine policies that fit to the stationary unconditional distribution which the VAR(1)-process converges towards. On the other hand, RNNs determine policies that changes its allocation over time. This could be a sign the policies of the RNNs fit to the conditional distribution instead. If this is the case, it is likely because of the vanishing gradient ”problem” of RNNs. The CNNs and LSTMs have stronger long term memories so it would make sense that it would be harder for them to fit to the conditional distribution which only depends on the last time step. Fitting to the conditional distribution is naturally more potentially powerful than fitting to the unconditional distribution. We also see this from the results being the superior performance of the RNN when compared to the CNN and LSTM.

Further research is needed for testing the efficacy of the methodology for real world allocation problems. The results of this simulation study indicate that the models are capable of balancing expected return with risk. Further investigation is needed to determine whether altering hyperparameter spaces and network topologies allow the models to perform across data with different underlying structures. Further work could also be done in trying different types of input data, other RL methods and the use of online learning. It would also be interesting to look at whether the models would perform markedly better given more computational power allowing the training of deeper and more complex networks.

References

[1] Jiang Z., Xu D., Liang J. (2017) A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem

[2] Ledoit O., Wolf M. (2017) Nonlinear Shrinkage of the Covariance Matrix for Portfolio Selection: Markowitz Meets Goldilocks

[3] Hillier D., Grinblatt M., Titman S. (2012) Financial Markets and Corporate Strategy (2nd)

[4] Green J., Hand J., Zhang X. (2013) The supraview of return predictive signals

[5] Zivot E., Wang J. (2002) Modeling Financial Time Series with S-PLUS. Springer-Verlag

[6] Sutton R., Barto A. (2018) Reinforcement Learning: An Introduction

[7] Dixon M., Halperin I., Bilokon P. (2020) Machine Learning in Finance: From Theory to Practice

[8] Charu C. Aggarwal. (2018) Neural Networks and Deep Learning

[9] Israel R., Kelly B., Moskowitz T. (2020) Can machines "learn" finance?

[10] Tanesh Balodi, (2019), Convolutional Neural Network (CNN): Graphical Visualization with Python Code Explanation
https://www.analyticssteps.com/blogs/convolutional-neural-network-cnn-graphi

[11] David Silver, (2015), Lectures on Reinforcement Learning
https://www.davidsilver.uk/teaching/

[12] Picture of AI landscape
https://www.datacamp.com/community/tutorials/machine-deep-learning

[13] Understanding LSTM Networks
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[14] Learning Curve Diagnostics
https://rstudio-conf-2020.github.io/dl-keras-tf/notebooks/learning-curve-dia

[15] Chollet, François and others, (2015), Keras

https://keras.io

A Properties of the VAR(1) Process on Seed 1

A.1 Conditional Covariance Matrix

	1	2	3	4	5	6	7	8	9	10
1	0.000170	0.000060	0.000028	0.000039	0.000007	-0.000040	0.000036	-0.000030	0.000030	-0.000014
2	0.000060	0.000263	**-0.000083**	-0.000007	0.000012	-0.000131	0.000022	-0.000096	0.000042	0.000052
3	0.000028	**-0.000083**	0.000109	-0.000076	-0.000023	0.000053	-0.000057	0.000085	-0.000018	-0.000049
4	0.000039	-0.000007	-0.000076	0.000344	0.000032	-0.000130	0.000042	-0.000010	0.000008	-0.000026
5	0.000007	0.000012	-0.000023	0.000032	0.000069	0.000022	0.000022	-0.000062	-0.000005	0.000012
6	-0.000040	-0.000131	0.000053	-0.000130	0.000022	0.000257	-0.000052	0.000052	-0.000021	0.000004
7	0.000036	0.000022	-0.000057	0.000042	0.000022	-0.000052	0.000169	-0.000033	0.000025	0.000027
8	-0.000030	-0.000096	0.000085	-0.000010	-0.000062	0.000052	-0.000033	0.000216	0.000010	-0.000073
9	0.000030	0.000042	-0.000018	0.000008	-0.000005	-0.000021	0.000025	0.000010	0.000094	-0.000026
10	-0.000014	0.000052	-0.000049	-0.000026	0.000012	0.000004	0.000027	-0.000073	-0.000026	0.000108

A.2 Unconditional Covariance Matrix

	1	2	3	4	5	6	7	8	9	10
1	0.000170	0.000056	0.000027	0.000043	0.000011	-0.000041	0.000042	-0.000031	0.000020	-0.000020
2	0.000067	0.000265	-0.000094	-0.000014	0.000015	-0.000154	0.000038	-0.000109	0.000020	0.000051
3	0.000029	-0.000075	0.000110	-0.000079	-0.000023	0.000061	-0.000060	0.000093	-0.000010	-0.000049
4	0.000037	-0.000002	-0.000075	0.000345	0.000033	-0.000151	0.000073	-0.000030	0.000009	-0.000031
5	0.000004	0.000012	-0.000023	0.000035	0.000069	0.000024	0.000021	-0.000066	-0.000004	0.000013
6	-0.000039	-0.000113	0.000047	-0.000113	0.000017	0.000258	-0.000056	0.000046	-0.000009	0.000012
7	0.000032	0.000005	-0.000056	0.000013	0.000024	-0.000046	0.000167	-0.000029	0.000021	0.000026
8	-0.000030	-0.000085	0.000079	0.000007	-0.000060	0.000058	-0.000034	0.000216	0.000020	-0.000072
9	0.000043	0.000066	-0.000028	0.000008	-0.000005	-0.000037	0.000033	-0.000003	0.000092	-0.000029
10	-0.000009	0.000053	-0.000050	-0.000022	0.000012	-0.000004	0.000030	-0.000075	-0.000024	0.000108

A.3 Mean Return Vector

1	0.000151
2	0.000382
3	0.000137
4	-0.000189
5	0.000280
6	-0.000060
7	0.000284
8	0.000033
9	0.000318
10	0.000248

B Model Allocations on Seed 1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
CNN	0.093336	0.109347	0.141493	0.024361	0.133014	0.069969	0.099902	0.045589	0.086168	0.196821
LSTM	0.11429	0.09092	0.151174	0.011544	0.077414	0.059832	0.051231	0.033401	0.066814	0.34338

C Price Series on Seeds

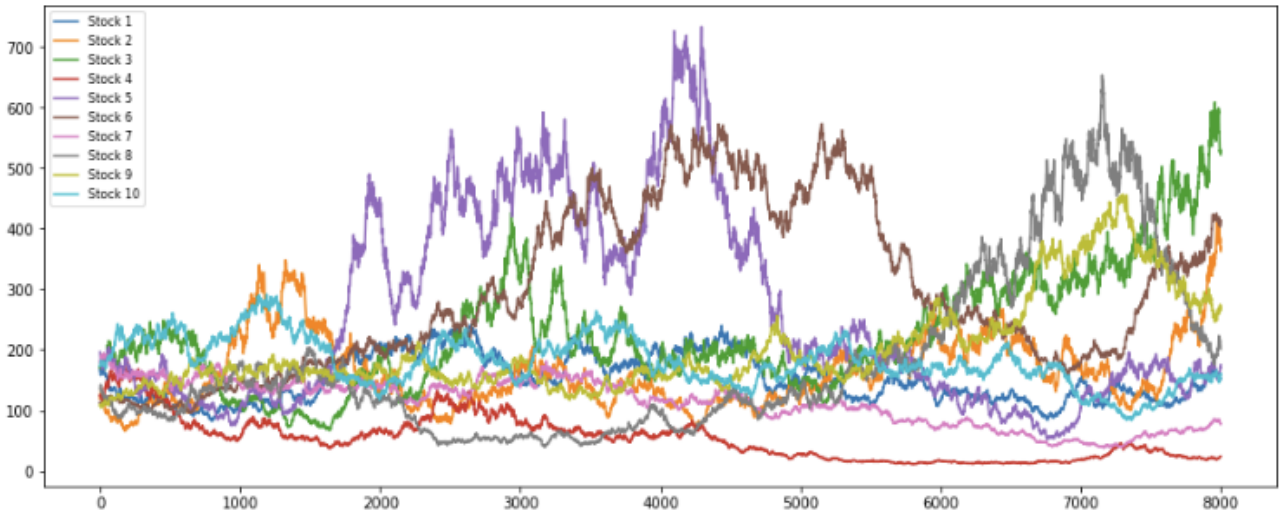


Figure 26: Price series on seed 20

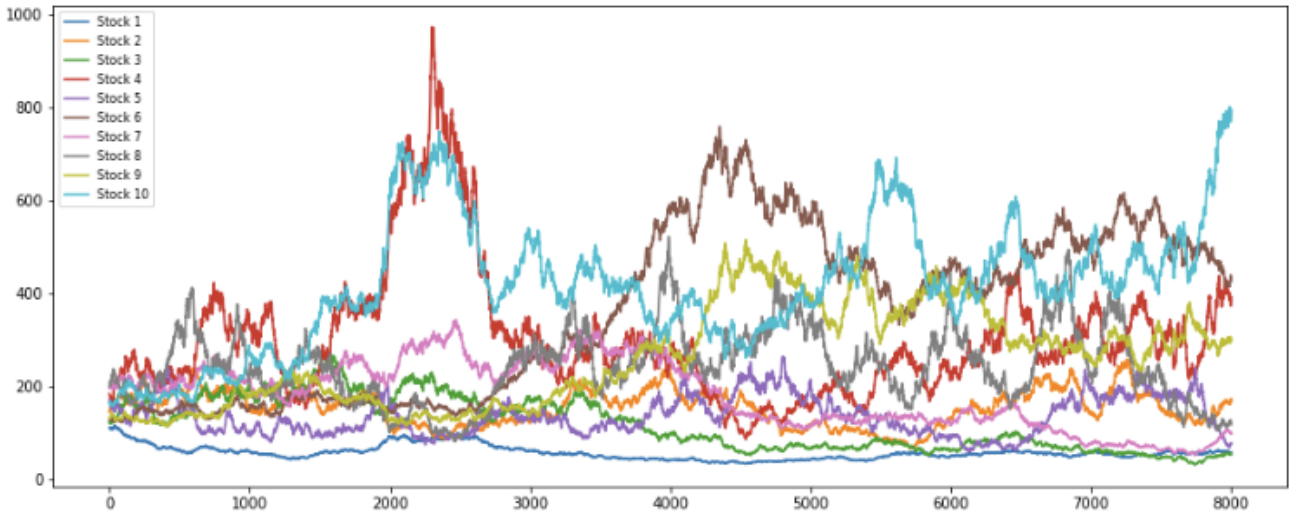


Figure 27: Price series on seed 30

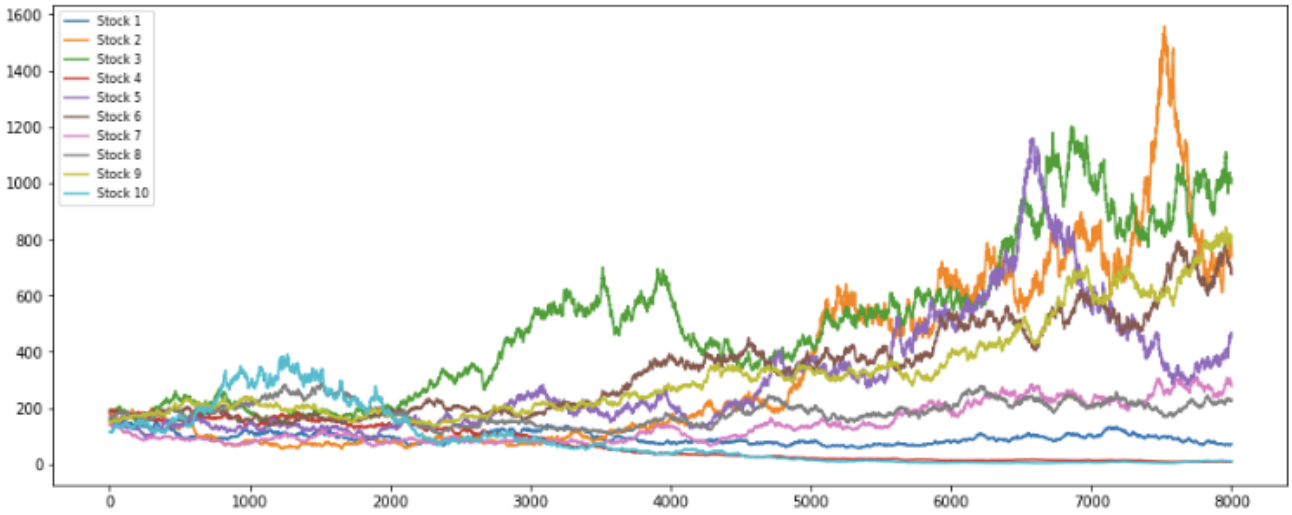


Figure 28: Price series on seed 43

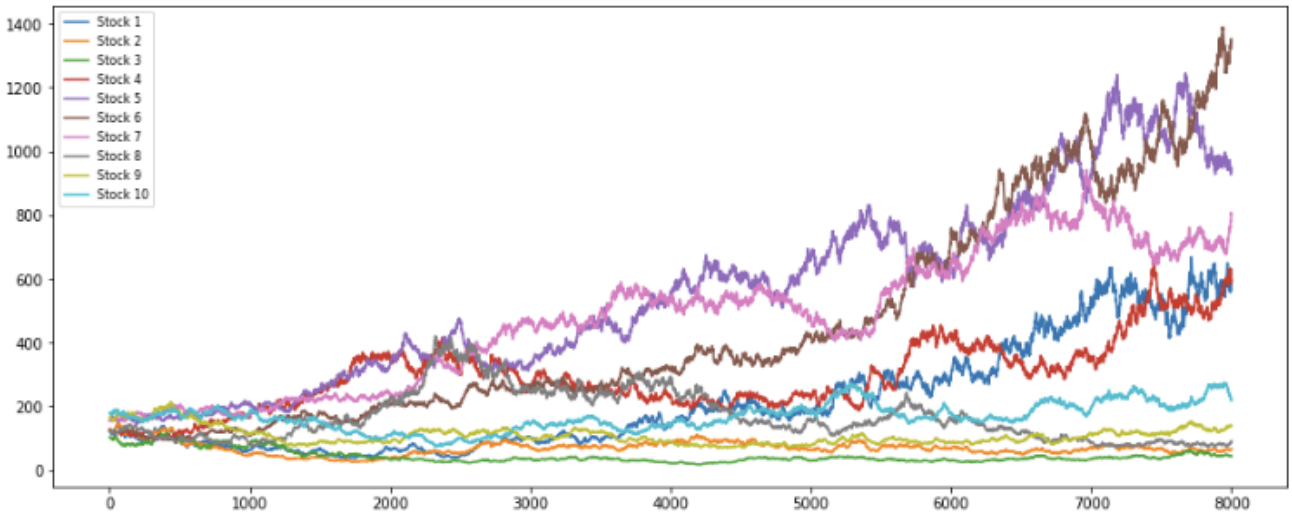


Figure 29: Price series on seed 59

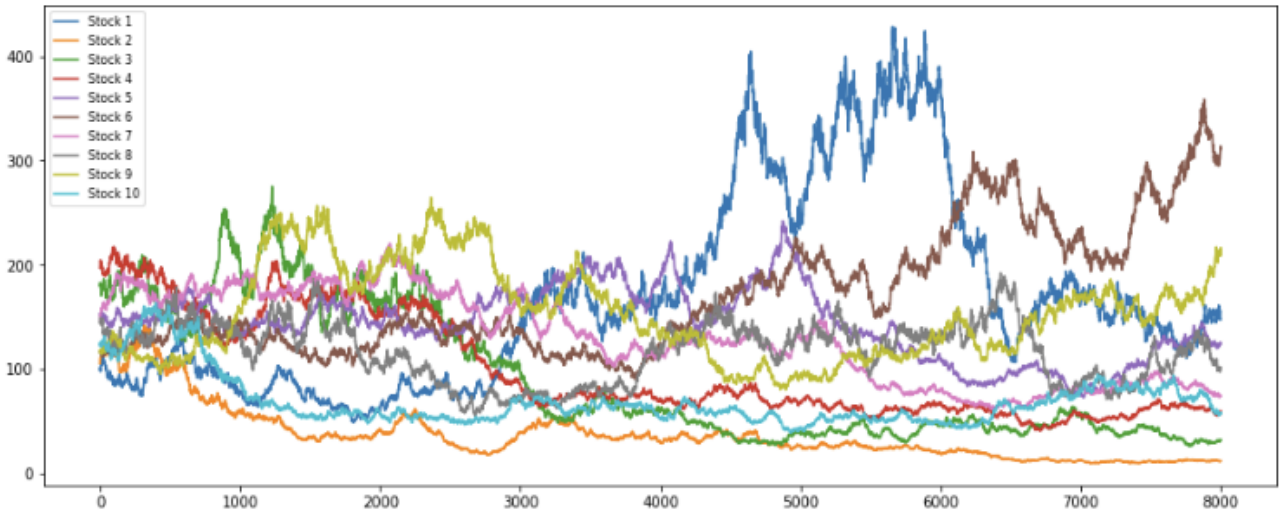


Figure 30: Price series on seed 64