FernUniversität in Hagen

Institut für Mathematik und Informatik

Master Thesis

Using Reinforcement Learning to enhance the investment decision process

Betreuer: Dr. Fabio Valdés

Dr. Johannes Winkler

Matrikelnummer: 8500100

Studiengang: Master of Data Science

March 30, 2025

List of abbreviation

- ADF - augmented Dickey-Fuller-test
- API - application programming interface
- CAPM - Capital Asset Pricing Model
- CPI - consumer price index
- DJIA - Dow Jones Industrial Index, equity index of the 30 largest use US companies trading on stocks
- DP - Dynamic Programming
- ECB - European Central Bank
- EURIBOR - EUR interbank offering rate, interest rate at which bank lend each other money
- EST - eastern standard time
- FED - Federal Reserve System, central bank of the USA
- FRED - Federal Reserve Economic Data
- MDP - Markov Decision Process
- MPT - Modern Portfolio Theory
- NFP - Non-Farm-Payroll, number of newly created jobs in the US except for the farming sector
- RL - Reinforcement Learning
- SHAP values - SHapley Additive exPlanations values
- S&P500 - equity index of the 500 largest US companies trading in stocks
- TD - temporal differences
- TS - Thompson Sampling
- VAR - vector autoregressive model

Contents

1 Introduction **1**

2 Modern Portfolio Theory **6**

2.1 Portfolio Selection 6
2.2 Risk-averse investors 9
2.3 Optimal Portfolio 12

3 Thompson Sampling **15**

3.1 Choice between two risky assets 17
3.2 Multi normal conjugate distribution 19

4 Reinforcement Learning **21**

4.1 Model based learning 22
4.2 Temporal Difference, Monte Carlo and q-learning 28
4.3 Deep neural networks 32
4.4 Applying deep q-learning to Gridworld 37
4.5 Related work 38

5 Macroeconomic and synthetic data **39**

5.1 The actual data 40
5.2 The Box-Jenkins-Method and VAR models 43

6 Portfolio construction and performance **47**

6.1 Modern Portfolio Theory 47
6.2 Thompson Sampling 51
6.3 Reinforcement Learning 54
6.3.1 Implementing RL algorithm using a neural network 55

7 Evaluation **62**

7.1 Results from training data set 62
7.2 Results from the test data set 67
7.3 DQN, target network and experience reply 70
7.4 Results of DQN on the test data set 73
7.5 Investment with a finite horizon 80

8 Summary **81**

1 Introduction

In this thesis we investigate if *Reinforcement Learning* (RL) can be used in the portfolio construction process. That is, we want to find out if an RL algorithm can learn from macroeconomic data when to raise the investment in a risky asset class such as equity and when to invest more in a safer money market account.

Generally, people invest their capital in financial markets with different motivations. They want to save for retirement, to reach a certain amount of capital to make a large purchase, such as a house, or simply to increase their capital to pass it on to their children. There are various financial instruments available to achieve these goals, ranging from savings accounts and government bonds to stocks and even gold.

It is well understood that financial products differ in their inherent risk and, closely related to that, in their expected return. A money market account is considered one of the least risky investments, as capital can be withdrawn daily and in many countries, these are even insured by the government up to a certain threshold[1] for private individuals. Hence, the yield on a savings account is accordingly relatively small as the investor does not need a big incentive in terms of return to invest into a money market account. Furthermore, the usage of this capital to a bank is limited due to the possibility of daily withdrawal. Therefore, the bank is willing to pay only a relatively low interest on such saving accounts.

A fixed income instrument, such as a bond, pays a higher interest rate compared to the savings account. This can be explained in two ways. First, the investor is willing to commit the funds for a longer time period and hence asks for a higher yield in return. Second, the investor might be able to sell the bond in the so-called secondary market, but if the general interest rate level has gone up since the time of purchase, she will suffer a loss when selling the bond. Thus, the bond bears a higher risk compared to the savings account. In fact, a bond carries two risks. There is market risk as its price changes with the level of interest rates. Secondly, it carries credit risk, as there is a chance that the issuer of the bond might not be able to repay the bond due to default. Thus, if the creditworthiness of the issuer of the bond deteriorates after the purchase of the bond (e.g., a rating agency like Standard & Poors reduces the rating of the issuer), the investor will see its market value decrease. Due to the higher risk of the bond compared to

[1]In Germany a savings account of a private individual in ensured up to EUR 100,000 by the government via the Einlagensicherung.

the savings account, the investor needs a bigger incentive to invest in the form of a higher yield.

The third instrument we take into account in this study is shares. An investor who buys stocks of a company becomes a part-owner of this company. The value of the shares is directly linked to the financial success of that company. If the company goes bankrupt, her shares will be worthless, while if the company is able to increase its profit, the price of the shares will increase. Of the four investment possibilities considered here, equity is historically the most risky but also the most lucrative investment.

The last[2] financial instrument considered here is gold. Although it pays no interest like a bond or dividends like a share, many investors use it as protection against inflation or as a store of value when uncertainty about future developments in financial markets increases. It is considered a "safe haven" investment in which individuals invest their money in times of crisis.

In finance it is well established, see for example [Fam17], that the expected return increases with the risk of the investment. Hence, the expected return of shares as an asset class ranks above the return of bonds, which in turn have a higher expected return than a savings account[3]. However, it is less clear how much each individual should invest in each asset class. Harry Markovitz introduced the *Modern Portfolio Theory* (MPT) (see [Mar52]) to address this question. Markovitz's work was followed by William Sharpe, who introduced the *Capital Asset Pricing Model* (CAPM) (see [Sha64]) and has shown that a rational investor should invest in a combination of a save asset, at the so-called risk-free rate, and a certain portfolio of risky assets that exhibits the highest risk-adjusted return. The exact proportion of the risky portfolio and the save asset depends on the individual's risk preferences. We will discuss the details in section 2.

Portfolio construction applying the MPT is a two-step process. First, one chooses a portfolio of risky assets. Second, the proportion of capital invested in this risky part of the portfolio is determined. The remaining capital is invested in the safe asset. Most research in finance is dedicated only to the construction of the optimal portfolio of the risky assets, since the second step is relatively easy and depends on the risk-aversion of the investor. In this thesis, however, we want

[2]We will not cover other asset classes like real estate or private equity as these are not freely trade-able on exchanges or even accessible for private individuals.

[3]This does not imply that at any given point in time the actual return of these asset classes are in that order. However, on average the returns should be in that order otherwise no investor would invest in the riskier asset classes.

to include the safe asset in the portfolio construction using an RL approach.

As we will show in section 2, for a rational investor the mixture of risky assets like bonds, shares and gold and non-risk bearing asset like a savings account can be shown to lie on a straight line in the risk-return-space between the risk-free rate and the return of a specific portfolio of the risky assets. Where exactly is determined by the individual risk preferences of the investor. This is usually modeled by an individual utility function which describes the risk preferences of the investor. Each investor maximizes her individual expected utility by choosing a personally optimal allocation between risky and save assets.

The MPT framework assumes a static environment in which markets conditions such as expectations about future payoffs of financial instruments and the associated risks are constant over time. In reality, however, this is not the case. It is obvious that the risk associated with the shares of a single company fluctuates with any news concerning that company or the industry in which the company operates, while the entire market reacts to events affecting the general risk perception of all investors, e.g., the COVID-19 outbreak.

To give an example, the share price of Apple fluctuates with every new information concerning the market for mobile phones, semiconductors, and the US-China trade relationship since China is one of the most important markets for Apple's iPhone line, to name only three factors. These idiosyncratic news affects only certain companies. For example, BMW's share price will be largely unaffected by news about the mobile phones while news regarding semiconductors or China is likely to have an impact on BMW's share price as well. The same argument can be made for the entire market and the general perception of risk. If the general public becomes more pessimistic about future economic conditions due to, for example, a statement by the Fed, a break-out of a pandemic similar to COVID or the start of a military conflict similar to the Russian attack on Ukraine in 2022, it would have an impact on all share prices. As a result, many, if not all, investors would not only adapt their expectations concerning future income but would also become more risk-averse. Hence, the optimal mix between risky and save assets of these investors would change. In these kinds of extreme situation, one can observe the so-called "flight to quality" in financial markets. The demand for save assets like US-government bonds and gold increases, while assets associated with more risk are selling off. This can be interpreted as a general shift in the perception of risk, leading most investors to change their portfolio allocation between save and risky assets.

If we assume that not only the perception of risk can change over time but also that the risk of individual assets, measured for example as the standard deviation of its share price, can change over time, then the MPT framework cannot capture these changes as it only gives a backward-looking portfolio allocation, which is predominately determined by the distribution (i.e., the historic mean and standard deviation) of asset prices before the systematic change. Examples of such systematic changes are the bursting of the dot-com bubble in 2000, the financial crisis in 2008 with the default of Lehman Brothers or the break-out of COVID in 2020. All of these events caused the financial markets to abruptly reevaluate the expected future cash flows of *all* financial assets and move to a new equilibrium by selling risky assets like shares and buying safer assets like government bonds, cash and gold.

If there is a systematic change in the underlying distribution of the prices of financial assets, which is by definition unknown[4], is generally possible, then it is worthwhile to explore whether other mechanisms are able to adjust quicker or more efficiently to such changes. Hence, the mixture of risky and save assets of a general investor is not constant overtime but should vary with market conditions. Thus, an investor would want to modify her investments accordingly. An alternative way to calculate historical averages of prices of financial assets and their standard deviations and use the estimated distribution of prices to determine the optimal investment portfolio using MPT is to update the underlying distribution using Thompson Sampling. This algorithm starts with a prior distribution of the variables of interest. By drawing multiple samples from this prior distribution, the algorithm updates this distribution using Bayesian statistics to obtain a posterior distribution. Applied to financial markets, we start with a prior distribution based on historical prices of financial assets. When observing new prices, an investor updates the distribution using Bayesian statistics. Based on the new posterior distribution, she then determines the new optimal portfolio. We will discuss this idea in detail in section 3.

An alternative to classic portfolio optimization is to use an RL algorithm to analyze when and how an investor should change her portfolio allocation. In general, an RL algorithm interacts with the environment based on observations on how different actions lead to different rewards. The algorithm learns from posi-

[4]If the distribution were known to an investor, she would trade on that knowledge, prices of financial assets would change accordingly so that no more risk-free profits are possible. This is known as the efficient market hypothesis.

tive and negative rewards by minimizing a loss function. That is, it learns how to maximize the rewards by choosing certain actions in a given situation. In our setting, the goal is to maximize the investor's utility from the financial rewards of her investment decisions. The action the algorithm has to decide on is the allocation of funds between the risky assets and the risk-free money market account. The reward is the financial pay-off of the chosen portfolio in the next period, which depends on the current allocation decision. The theoretical advantage of this approach over classical portfolio optimization is that macroeconomic data such as the consumer price index (CPI) measuring inflation can carry valuable information the RL algorithm can learn from, while such context variables do not enter the MPT at all. Hence, this approach opens the possibility for a wider range of attributes to have an impact on the allocation decision of an investor. In classic portfolio theory, these factors enter the picture only implicitly via the prices of the financial assets reacting to these variables, as the MPT only takes prices of financial assets and their correlation into account.

In a Reinforcement Learning algorithm, a neural network can be utilized to perform deep learning, capturing complex nonlinear relationships between actions, given a specific environmental state, and their corresponding rewards. The idea of this set-up is that by implementing such deep learning algorithm, we are not only in a position to find and build on the theoretical solutions to the optimal portfolio allocation in a dynamic setting, but also introduce macroeconomic variables in addition to the price data of financial instruments such as inflation, unemployment data, or economic sentiment data to ideally reduce the portion of risky assets before or at the beginning of an economic downturn and increase it again once these macro data indicate a recovery. If we can successfully design an environment in which an algorithm is able to optimize the portfolio allocation by retrieving additional valuable information from macroeconomic data, we can analyze if the impact of these variables matches our economic intuition.

The thesis is structured as follows. First, we describe the theoretical concepts of Modern Portfolio Theory in section 2, as this serves as benchmark for our approach. Thereafter, we discuss the theory of two concepts of artificial intelligence applied to the investment problem. First, in section 3 we introduce Thompson Sampling (TS). Second, in section 4 we describe how RL can be applied to the investment problem. Here we also briefly discuss related work where in fiance Reinforcement Learning has been applied to. In section 5 we introduce the data on which the analysis is performed and how to generate synthetic data to train the RL model on. Finally, in section 6 we apply the theoretical concepts to the data and evaluate results in section 7. The thesis is concluded in section 8.

2 Modern Portfolio Theory

In 1952 Harry Markowitz (see [Mar52]) introduced *Modern Portfolio Theory* (MPT) in which financial assets are measured in two dimensions: their expected return and their individual risk. He made the groundbreaking discovery that it is possible to construct portfolios that have less risk than the sum of the individual assets by choosing uncorrelated or even negatively correlated assets. The diversification of risks thereby allows us to construct portfolios that have less risk but the same expected return as the individual assets of which they are comprised. For this concept he received together with William Sharpe the Nobel prize in Economic Sciences in 1990. Today, the idea of diversification in a portfolio of financial assets is common knowledge among asset managers and has even led to the creation of financial products like mutual funds and exchange traded funds or ETFs, which enable private investors to invest into predefined diversified portfolios with the purchase of only one financial instrument instead of investing in many individual shares, which would cause more transaction costs.

In the 1960s, William Sharpe built on Markovitz's work of diversification by combining it with individuals' utility maximization of investors and their risk preferences to establish the *Capital Asset Pricing Model* or CAPM. In the next subsection we will derive these concepts as they will serve as the benchmark of the portfolio construction process we aim to beat with the RL algorithm.

2.1 Portfolio Selection

As stated above, in modern portfolio theory, financial assets are considered in two dimensions: their expected return, denoted by μ, and their risk, measured by the standard deviation of asset prices, σ. This view allows to display the available assets in a two-dimensional graph with the expected return on the vertical axis and the risk on the horizontal axis. When all assets an investor could invest in are presented like that, it is easy to see what her goal is: for given level of risk, she wants to maximize the expected return or move up in the $\mu - \sigma -$space. Equivalently, for a given level of return, she wants to minimize the risk or move left in the $\mu - \sigma -$space. This problem can be formulated as a minimization problem under constraints which can be solved using Linear Programming.

We assume that there are a finite number of assets $i = 1, 2, 3, ..., N$ available. We denote the expected return of asset i by μ_i, the risk of asset i by its standard deviation σ_i and the percentage of the portfolio of the investor in asset i by w_i. In matrix notation, a portfolio is given by the vector $\mathbf{w}$ with $\mathbf{e}^T\mathbf{w}$=1 or $\sum_{i=0}^{N} w_i = 1$, where $\mathbf{e}$ is the unit vector of dimension N. If we denote with $\boldsymbol{\mu}$ the vector of expected returns and with $\mathbf{\Sigma}$ the covariance matrix of all assets, then the expected return and standard deviation of the portfolio are

$$\mu_P(\mathbf{w}) = \mathbf{w}^T\boldsymbol{\mu} \text{ and } \sigma_P(\mathbf{w}) = \sqrt{\mathbf{w}^T\mathbf{\Sigma}\mathbf{w}}. \tag{1}$$

An investor wants to find for a given level of return $\bar{\mu}$ the weights $\mathbf{w}$ so that the resulting portfolio has the lowest possible risk. That is,

$$\min \frac{1}{2}\mathbf{w}^T\mathbf{\Sigma}\mathbf{w} \text{ subject to } \mathbf{e}^T\mathbf{w} = 1 \text{ and } \mathbf{w}^T\boldsymbol{\mu} = \bar{\mu}. \tag{2}$$

This problem can be solved using a Lagrangian[5] function for constrained optimization.

$$\mathcal{L}(\mathbf{w}, \boldsymbol{\lambda}) = \frac{1}{2}\mathbf{w}^T\mathbf{\Sigma}\mathbf{w} + \lambda_1(1 - \mathbf{e}^T\mathbf{w}) + \lambda_2(\bar{\mu} - \mathbf{w}^T\boldsymbol{\mu}) \tag{3}$$

The derivative of $\mathcal{L}(\mathbf{w}, \boldsymbol{\lambda})$ with respect to $\mathbf{w}$ yields

$$\mathbf{\Sigma}\mathbf{w} = \lambda_1\mathbf{e} + \lambda_2\boldsymbol{\mu}, \text{ which is equivalent to } \mathbf{w} = \lambda_1\mathbf{\Sigma}^{-1}\mathbf{e} + \lambda_2\mathbf{\Sigma}^{-1}\boldsymbol{\mu}. \tag{4}$$

The other first order derivatives of the Lagrangian function are the constraints:

$$1 - \mathbf{e}^T\mathbf{w} = 0 \text{ and } \bar{\mu} - \mathbf{w}^T\boldsymbol{\mu} = 0. \tag{5}$$

To simplify the expressions, one often uses the following definitions (see for example [Bru24]):

$$A = \boldsymbol{e}^T\mathbf{\Sigma}^{-1}\boldsymbol{e}$$

$$B = \boldsymbol{\mu}^T\mathbf{\Sigma}^{-1}\boldsymbol{e}$$

$$C = \boldsymbol{\mu}^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu}$$

With these definitions and requiring $AC - B^2 > 0$, there is a closed-form solution

[5]The Langrange method, named after the mathematician Jospeh-Louis Lagrange, is a method to find local optima of a function subject to constraints.

for the optimization problem which we denote by $\mathbf{w}_{EF}$ for efficient frontier:

$$\mathbf{w}_{EF} = \frac{C - \bar{\mu}B}{AC - B^2}\boldsymbol{\Sigma}^{-1}\boldsymbol{e} + \frac{\bar{\mu}A - B}{AC - B^2}\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}. \tag{6}$$

This equation implies that the optimal portfolio is linear in expected return (the second term of the equation) and that there is a portfolio with minimal risk (when $\lambda_2 = 0$ the second term of the equation disappears). The equation describes a hyperbola[6] in the $\sigma - \mu -$ space, the so-called *efficient frontier*.

MPT assumes a perfect market: investors act perfectly rational, they are perfectly informed, prices change instantly, the market is perfectly liquid, all assets are traded without any transaction costs and tax, and lastly, all investors can borrow or lend at the same rate. Hence, the theory assumes an ideal capital market that does not exist. In particular, the last assumption, that investors are able to borrow money at the same rate at which they lend, has an implication on the constraint of the portfolio weights because w_i could be negative. Since this is unrealistic, an additional constraint on the weights is $0 \leq w_i \leq 1$ for all i from $1, 2, ...$ to N.

When investments in individual assets are forced to be non-negative and the possibility of short-selling is gone, the optimization problem in equation (2) has no longer a closed-form solution. One way to determine a solution is to use a solver for a quadratic programming problem. The Python listing 10 in the appendix shows the relevant code to calculate the efficient frontier, which uses sequential quadratic programming (method = 'SLSQP') by invoking the Scipy package to solve this problem. That is, for a given level of return, i.e., the return target denoted by $\bar{\mu}$, the algorithm finds the portfolio weights that minimize the risk of the portfolio, thus starting from equally weighted investments in all available assets. The two constraints from equation (2) are defined by the tuple of dictionaries called *constraints* and the condition for the weights to be between 0 and 1 is maintained in the tuple *bounds*.

For a simple portfolio of two assets, the resulting solution to the optimization problems is given by the hyperbola shown in Figure 1.

With the efficient frontier known, we have a set of portfolios to choose from. The logical question to ask is then, which one of the portfolios on the efficient

[6]The hyperbola, i.e., the efficient frontier, describes the two solutions of a quadratic optimization problem. Note that solutions below the minimum risk portfolio are dominated by portfolios on the upper part of the efficient frontier due to their higher expected returns.

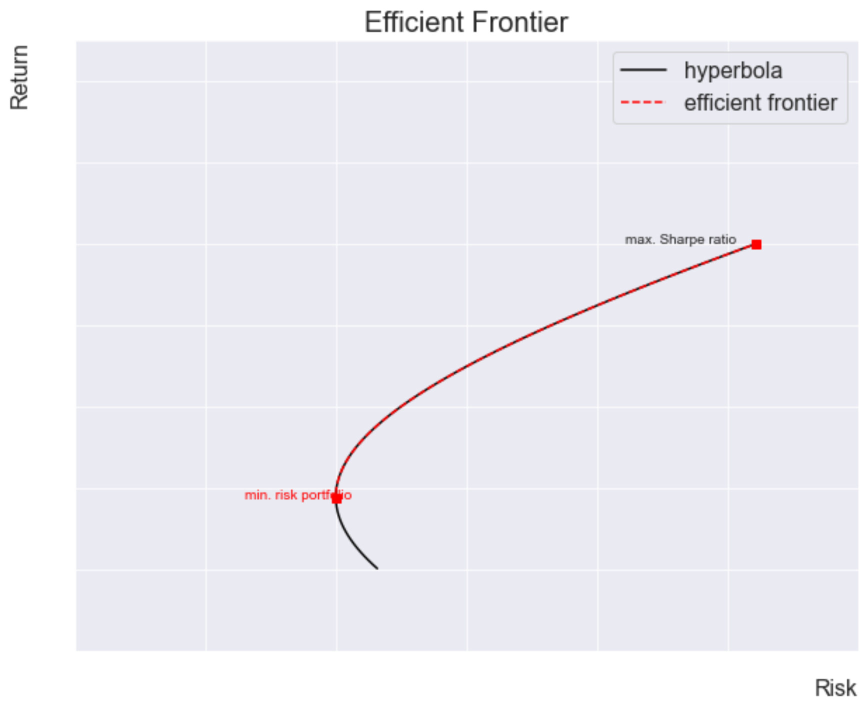


Figure 1: Efficient Frontier

frontier an individual investor should choose?

To answer that question, we have to look at the individual utility function of a rational investor. She would choose a portfolio that maximizes her expected utility of the return of the portfolio. Hence, not all investors choose the same portfolio as they differ in their risk appetite. This leads us to the concept of risk aversion.

2.2 Risk-averse investors

Many people would rather receive EUR 90 instead of participating in a lottery in which they win EUR 50 or EUR 150 with an equal chance of 50%. This behavior[7] is not rational as the expected profit from the lottery is EUR 100, clearly better than receiving EUR 90. The amount an investor considers equal to the lottery depends on her individual level of risk aversion. Figure 2 (source:[Wikb]) shows a utility function that exhibits these preferences.

In our lottery example, we have the following values: w_0 = EUR 50, w_1 =

[7]The risk-aversion is one of the reasons why investors tend to sell ”winner” stocks too early and keep ”looser” stocks too long: loosing again the profit from the ”winners” hurts more (has a larger negative utility) than having a loss from the ”looser” stock.

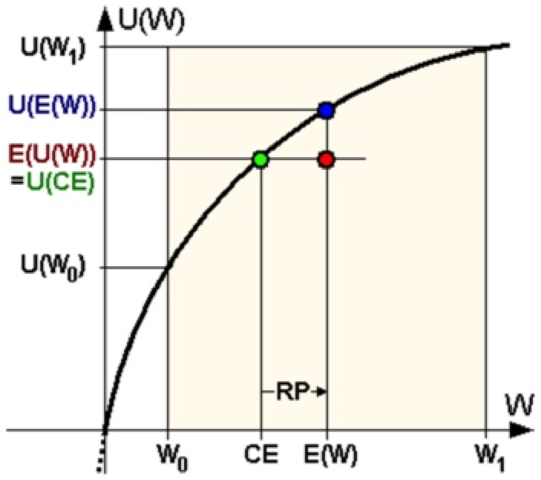


Figure 2: Utility function exhibiting risk-aversion

EUR 150 and $E(w) = 100$. The *certainty equivalent*, CE, of our investor is EUR 90 as the utility of this value is equivalent to the expected utility of the lottery, $\mathbb{E}(U(w))$. The difference between the expected value of the lottery, $\mathbb{E}(w)$, and the certainty equivalent, *CE*, is called the *risk premium*, RP.

For a risk-averse person, the risk premium is positive, for a risk-neutral person it is zero, and for a risk-affine person RP is even negative.

An often used utility function (see for example [Sti03]) to describe such behavior is called CARA, for *constant absolute risk aversion*:

$$U(w) = \begin{cases} \frac{1-\exp^{-\gamma w}}{\gamma} \text{ for } \gamma \neq 0, \\ w \text{ if } \gamma = 0 \end{cases} \tag{7}$$

The parameter γ captures the degree of risk-aversion of the individual with this utility function. When plotting the function, it also defines the curvature or the degree of concavity of the function. The absolute risk-aversion, defined as $\frac{-u''(c)}{u'(c)}$ is equal to γ, which explains the name of the function.

Note from Figure 2 that $\mathbb{E}(U(w)) = U(CE)$. That is, the expected utility of participating in the lottery is equal to the utility of CE. This holds not only for a risk-averse investor but for any kind of investor. To see that, note that for a risk-neutral investor the utility curve is a straight line while it is convex for a risk-seeking investor.

Since we aim to maximize $\mathbb{E}(U(w))$, we can alternatively maximize the utility of CE. It can be shown (see for example [Rao20], where the expression is found by Taylor-expanding the utility function) that the risk premium, RP, can be defined as $w - CE \approx -\frac{1}{2}\frac{u''(w)}{u'(w)}\sigma_w^2 = \frac{\gamma\sigma_w^2}{2}$, where σ_w^2 the risk associated with w. This expression makes intuitive sense as the RP increases with the degree of risk-aversion of the investor, γ, as well as with the risk of the asset, σ_w. Hence, $CE \approx w - \frac{\gamma\sigma_w^2}{2}$.

A common assumption for the distribution of the return of an asset i, which can be interpreted as the annual change in wealth resulting from an investment in that asset, is that the annual return is normally distributed, $\mathcal{N}(\mu_i, \sigma_i^2)$.

If we finally assume that next to the risky assets, which are all comprised in the portfolios described by the efficient frontier, there exists a risk-free investment opportunity that is uncorrelated to all other assets, then a rational investor will invest into a linear combination of this risk-free asset and a portfolio on the efficient frontier. A proxy for such a risk-free rate for a private individual is a simple money market account, since this has, as stated earlier, often a governmental guarantee up to a certain threshold.

The annual return of the risk-free rate is denoted by r_f. The annual return for an investor who invests portion p in a sub-portfolio of risky assets somewhere on the efficient frontier and $1-p$ in the risk-free rate is $p(\mu - r_f) + r_f$, where we denote the annual return of a given portfolio from the efficient frontier by μ and its standard deviation by σ. The distribution of the portfolio consisting of the risk-free rate and a portfolio from the efficient frontier is then $\mathcal{N}(p(\mu - r_f) + r_f, p^2\sigma^2)$ since the risk-free rate has by definition no risk (i.e., the standard deviation is zero) and is by assumption not correlated with any other asset.

Given the expression of the certainty equivalent, we can now enter the corresponding value of the expected return of the portfolio and its associated risk to get $CE = p(\mu - r_f) + r_f - \frac{\gamma p^2\sigma^2}{2}$. As we want to maximize the expected utility of the return of the portfolio by choosing p optimally, we have to solve the following problem:

$$\max_p \ \mathbb{E}(U(w)) = \max_p \ U(CE) = \max_p \ U(p(\mu - r_f) + r_f - \frac{\gamma p^2\sigma^2}{2}).$$

Since the utility function is monotonically increasing in w[8], this is equivalent to

$$\max_p \ p(\mu - r_f) + r_f - \frac{\gamma p^2\sigma^2}{2}.$$

[8]More wealth is better than less.

Hence, when setting the first derivative equal to zero, the optimal ratio to invest into the risky portfolio is

$$p^* = \frac{\mu - r_f}{\gamma \sigma^2}, \text{ with } p^* \in [0,1] \text{ and } \gamma > 0. \tag{8}$$

This result matches our economic intuition: the investor invests into the risky portfolio only if its expected return is greater than the risk-free rate[9], $\mu > r_f$. Furthermore, investment in the risky portfolio decreases with the degree of risk-aversion of the investor, γ, and risk of the portfolio, σ^2. Put differently, the investor raises the amount of money she puts into the money market account, $1 - p^*$, until she feels comfortable with the risk of the combined portfolio, which comes solely from the sub-portfolio of risky assets somewhere on the efficient frontier. In the next section we will determine the conditions to identify this particular portfolio on the efficient frontier.

2.3 Optimal Portfolio

We can now combine the efficient frontier and the maximization of the expected utility of the investment to determine the optimal investment decision of an investor depending on the available assets and the personal risk preferences. Of all portfolios available on the efficient frontier, a rational investor would choose the one that earns the highest expected return over the risk-free rate for a given level of risk. In the risk-return space of Figure 1, this is the portfolio on the efficient frontier, called the market portfolio, where a line starting at the risk-free rate, point (0, r_f), is tangent to the efficient frontier. To illustrate, we can imagine drawing a vertical line starting in (0,r_f), then tilting the line to the right until it just touches the efficient frontier. At this point, the ratio $\frac{\mu - r_f}{\sigma}$ is maximized and the so-called excess return of the portfolio over the risk-free rate is as large as possible. William F. Sharpe determined this particular portfolio (see [Sha64]) and the slope of the line between the risk-free rate, (0,r_f), and the market portfolio (σ_m, μ_m) is named after him:

$$\text{Sharpe-ratio} = \frac{\mu_m - r_f}{\sigma_m}. \tag{9}$$

[9]Note that p^* is between 0 and 1. Hence, if μ is smaller than r_f, then p^* is 0. If μ is so much bigger than r_f that the Sharpe ratio would be above 1, then $p*$ is equal to 1 as the investor cannot borrow at r_f to further increase the investment in the risky sub-portfolio further.

The subscript m denotes the expected return and the standard deviation of the so-called *market portfolio*, the sub-portfolio of risky assets. The Sharpe-ratio measures the excess return, $\mu_m - r_f$, per unit of risk, σ_m. Note that this portfolio is independent of the risk preferences of the individual investor.

A rational investor only needs to consider the risk-free rate and the market portfolio for her investment decision, as an optimal choice will be a mixture of the two. Hence, we can draw a straight line between these two points $(0, r_f)$ and (σ_m, μ_m), representing all linear combinations of these two portfolios. The slope of this line is determined by the Sharpe-ratio. Figure 3 illustrates this relation.

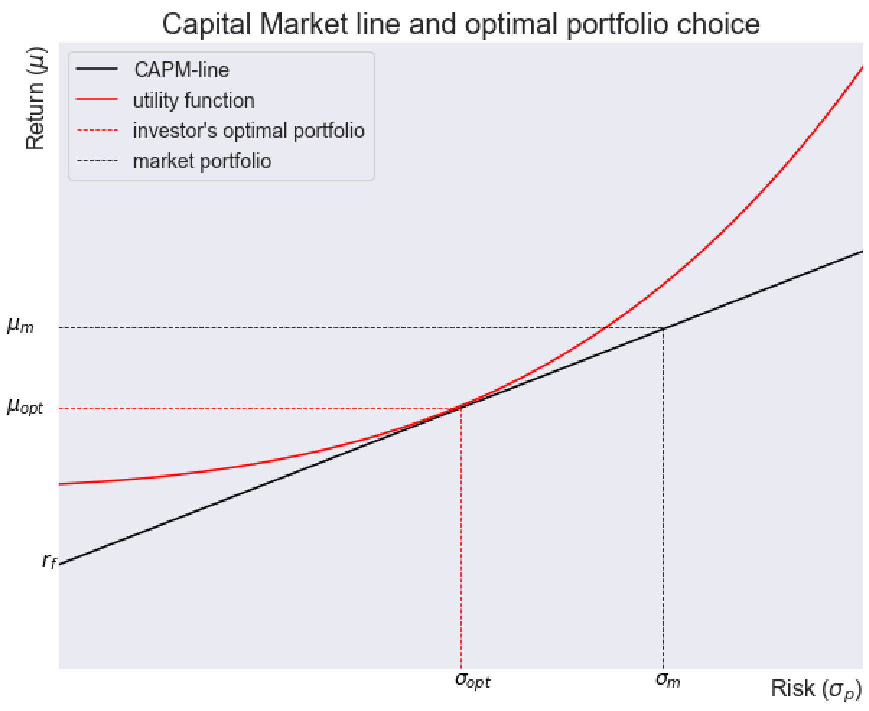


Figure 3: Optimal portfolio choice

The market portfolio, where the ratio of excess return to a unit of risk is maximized, is shown by the point (σ_m, μ_m). This portfolio is connected to the risk-free rate, which is indicated by r_f and is characterized by the point $(0, r_f)$ in the figure. The line is called the *Capital market line* or CAPM-line. The slope of the Capital market line is the Sharpe-ratio $\frac{\mu_m - r_f}{\sigma_m}$ and it crosses the vertical axis at the risk-free rate. The CAPM-line is therefore defined as

$$\text{Capital market line} = r_f + \frac{\mu_m - r_f}{\sigma_m}\sigma_p, \tag{10}$$

where σ_p is the risk of the resulting portfolio (or the unit of the horizontal axis in Figure 3) .

As a rational investor would choose only between the market portfolio and the risk-free rate, we know that an individual's optimal choice must lie somewhere on the Capital market line. Which point on the line an individual investor chooses depends on her individual risk-aversion parameter γ, see equation (8). Figure 3 shows a typical utility function in red. It is easy to see that the highest reachable utility is achieved at the point $(\sigma_{opt}, \mu_{opt})$, where the utility curve is just tangent to the Capital market line.

To summarize, the MPT offers a framework for finding the weights of the risky assets for an optimal sub-portfolio independent of the individual investor's risk preferences (the market portfolio denoted by (σ_m, μ_m) in Figure 3). In combination with the risk-free rate, the optimal portfolio composition of risk-free rate and risky assets is somewhere on the Capital market line. Where exactly depends on the individual preferences of an investor, γ, the risk of the market portfolio, σ_m, and the risk-free rate, r_f. Hence, in the MPT framework an investment portfolio is constructed in a two-step process. First, the optimal sub-portfolio of risky assets, the market portfolio, is found. Second, given the market portfolio, the risk-free rate and the individual's risk preferences, the optimal[10] portfolio for the investor is determined.

The MPT concept can be criticized, as the resulting portfolio construction depends only on historic prices and is not forward-looking. Furthermore, the risk-free rate is assumed to be uncorrelated to the risky assets, while in reality it is considered a viable substitute to any risk-bearing asset. If the risk-free rate is close to the expected value of a risk-bearing asset, many investors will shy away from an investment in the latter. Hence, the risk-free rate is in reality correlated to the return of risky assets and, therefore, to prices of other assets.

One advantage of the RL approach that we will investigate in later sections is that the algorithm will decide simultaneously how much to invest in the risk-free rate and how much in the risk-bearing assets. This should be a more realistic model of portfolio construction compared to the MPT. Hence, the RL approach has the opportunity to retrieve additional information from the correlation of the risk-free investment with all other assets to potentially construct a superior portfolio. This concludes the introduction of the MPT. In the next section we will introduce a portfolio construction approach that builds on the MPT but has

[10]Since the second step is a logical consequence and rather simple to do, most research is focused on the construction of the market portfolio.

the potential of incorporating new information efficiently.

3 Thompson Sampling

One of the criticisms of Modern Portfolio Theory is that selection of the investment portfolio is backward looking and static. The investor's decision is based on historical data. She collects historic asset prices, calculates the expected return and covariance matrix, and chooses her optimal portfolio accordingly. But what if the current distribution of asset prices is significantly different from the distribution in the past? Then the historic mean and the covariance matrix will not properly reflect the currently prevailing underlying distribution of asset prices.

This situation is related the multi-armed bandit problem, a classical problem of probability theory that also finds application in Reinforcement Learning. The problem is as follows: a gambler can choose between n different slot machines that are free to use for some reason but only for a limited time. The gambler therefore faces the question which of the slot machines (also known as one-armed bandits as they tend to rob the users of their money) to play. The slot machines have random payouts and differ in their pay-out distributions, which are unknown to the gambler. If the gambler plays the first slot machine and receives a decent payoff, she asks herself if she should continue to play this machine or switch to another slot machine hoping that this machine will have an even better payout distribution. Hence, the gambler faces a explore-versus-exploit dilemma. Should she keep on exploring the other slot machines or should she exploit the slot machine that generated the highest payout so far?

Although the problem sounds very theoretical, there are many real-life applications: the question how to allocate founds to different research facilities, the choice between different experimental drug trials to find the most effective treatment, or the search of optimal paths through a network while minimizing delays.

To see how the multi-armed bandit problem relates to portfolio design, one needs to consider the different assets or different portfolios of assets as the slot machines. These have different (future) payout distributions that are unknown to the investor. The investor wants to invest most of her money, if not all, in the asset with the highest expected payoff. Only after her investment decision she will learn the payoff of her portfolio (and of all other assets). With the new information about payoffs, she has to decide whether she wants to explore other asset allocations (i.e., explore and redistribute her wealth to other investments) or if she wants to stick to her current choice (i.e., exploit).

There are different ways to address the explore-exploit-dilemma in the multi-armed bandit problem, which cannot all be described here as this would exceed the scope of the study. An intuitive approach is the *epsilon-greedy strategy*. The gambler chooses with a large probability, $1 - \epsilon$, the strategy with the highest expected payout. This is a so-called greedy strategy, as this strategy maximizes the current needs of the decision maker while disregarding any strategies that yield a lower payoff currently but could yield a higher payoff in the future. However, with a small probability, ϵ, the gambler chooses a random strategy. Hence, in a limited number of cases, she continues exploring alternative strategies for their payoffs.

The epsilon-greedy strategy is easy to implement but also easy to criticize. The early choices of the gambler of which slot machine to play are very important as the speed of adjustment to bandits with a higher payoffs is very small. So, it is conceivable that the gambler first chooses a slot machine that pays a small but positive payoff. Although the payoff is small, it is bigger than the unknown payoffs of the other slot machines, and the gambler keeps on playing this particular machine for too long until she learns via the epsilon strategy of trying other machines that she is playing a suboptimal machine. Furthermore, it is also possible that the gambler found the slot machine with the highest expected payoffs. But despite this fact, she will continue to choose other machines with probability ϵ, which in this case is a suboptimal strategy[11].

An often better and more sophisticated approach to the multi-armed bandit problem is the so-called *Thompson Sampling* (TS). The algorithm is named after William R. Thompson who (see [Tho33]) developed mathematical strategies to balance the wish to maximize immediate performance and gathering additional information that may enhance future performance. The interest in Thompson's work and the usage of the algorithm has picked up significantly with the need to model online decision making, e.g., which advertisement to show in the banner of a webpage to a given user.

The TS algorithm usually involves an agent (i.e., the gambler in the multi-armed bandit setting), who obtains a specific context of the situation, denoted by

[11] A common way to address this criticism is to start with a relatively high value of ϵ and reduce it the longer the game is played.

$x \in \mathcal{X}$[12], and can choose an action a out of a range of options $\mathcal{A}$. Upon choosing the action a, the agent receives a reward $r \in \mathbb{R}$. The agent has an idea of the distribution of r, denoted by $P(\theta)$, which is called the prior distribution, where θ are the parameters for the distribution of r given the historical observations $\mathcal{H}$, which consists of triplets $\mathcal{H} = (x, a, r)$. The actual distribution of r, $P(r|\ \theta, a, x)$, is unknown to the agent. Hence, the agent forms the idea of the distribution given the information she has received so far. If at the beginning of the process the agent has no idea yet of how the distribution of r might look like, one often assumes a uniform distribution over all possible values of θ. Once the agent chooses an action a and receives a reward r, she will update her belief about the distribution of r, the so-called posterior distribution $P(\theta|\ x, a, r)$. The player will choose the action a^* that maximizes the expected reward: $a^* = argmax\ \mathbb{E}(r|\ \theta, a, x)$.

Algorithm 1 The Thompson Sampling algorithm

```
for $t = 1, 2, ...$ do
    /* sample the distribution */
    Sample $\hat{\theta} \sim P(\theta)$
    /* choose the best action */
    $a_t \leftarrow argmax\ \mathbb{E}_{\hat{\theta}}(r|\ \hat{\theta}, a, x)$
    Apply $a_t$ and observe $r_t$
    /* update the distribution */
    $P(\theta) \leftarrow P(\hat{\theta})$
end for
```

The Thompson Sampling algorithm describes how the agent updates her beliefs about the distribution of r from the prior to the posterior distribution. Its convergence for the multi-armed bandit problem (as well as for Markov Decision Processes, which we will introduce in the next chapter) has been shown (see [Str00]). In the next section, we describe how an investor repeatedly chooses between two risky assets. The risk-return parameters of the two assets are unknown to her, and she learns over time which of the two assets maximizes her individuals risk preferences.

3.1 Choice between two risky assets

With the following example, we want to demonstrate how Thompson Sampling works by modeling an investor, who can choose between two risky assets next to

[12]We re following the notation of [SB18].

a risk-free money market account. The investor has no previous information about the expected return or the risk of the two assets[13]. She will learn about the characteristics of the assets over time and will form her beliefs about their distribution accordingly. The return of the asset x and y per period follows $x \sim \mathcal{N}(\mu_x, \sigma_x^2)$ and $y \sim \mathcal{N}(\mu_y, \sigma_y^2)$. Following the framework of section 2, the investor is risk-averse and her utility of the return follows equation (7). Hence, the investor's preference depends not only on the expected return of the assets but also on their risks. For simplicity, we assume that the risks of the two assets, i.e., their standard deviations, are known to the investor and that only the expected return is unknown.

Initially, the investor has no prior information regarding either asset except for the respective standard deviation. Hence, the prior distribution of both assets is $\mathcal{N}(0, \sigma_x)$ and $\mathcal{N}(0, \sigma_y)$. The investor will therefore randomly invest into one of the two assets to find out about their expected return, will receive a return, and update her belief of the distribution, i.e. the posterior distribution of asset prices. The webpage [Wika] shows for different distribution functions how the prior distribution is updated to become the posterior distribution.

Following the TS algorithm described by algorithm 1, the investor randomly invests into one of the assets, learns about the payoff of that asset and updates the prior for that asset using the following two equations:

$$\sigma_t^2 = \left(\frac{1}{\sigma_{t-1}^2} + \frac{n}{\sigma^2} \right)^{-1}$$

$$\mu_t = \frac{1}{\frac{1}{\sigma_t^2} + \frac{n}{\sigma^2}} \left(\frac{\mu_{t-1}}{\sigma_t^2} + \frac{\sum_{i=1}^n x_i}{\sigma^2} \right),$$

where n denotes the number of samples taken, σ denotes the known standard deviation of the assets and x_i the observed value from the *ith* sample. The investor chooses between the risk-free rate, r_f, and either asset x or y according to the equation (8). She chooses the portfolio of risk-free rate and asset x or y that gives her the highest expected utility, where the weights of the risky asset is determined by equation (8) and the weight for the risk-free rate is $1-p$. This step is repeated multiple times and the investor learns over time the true distribution of the assets. We visualize this adjustment process in separate Jupyter notebook

[13]The investor invests into the risk-free savings account and in addition in one of the two risky assets as one of these will provide a strictly higher utility to her than the other.

(see [Win25e]), where we adjusted the code from [Rit24] to fit to this example.

Although it is obvious that the assumptions made for this example are unrealistic (e.g., why should the investor know the risk of the asset but not its expected return?), it demonstrates nicely the learning process within the TS algorithm. The more samples the investor collects, the more the posterior distribution converges to the true underlying distribution of the asset. It can be shown that if n is large enough, the difference between the asset most often chosen and its actual distribution tends to zero[14]. Hence, the investor learns the true distribution over time.

3.2 Multi normal conjugate distribution

In this subsection we will apply the TS algorithm in the investment context. One of the criticisms of MPT framework is its backward looking view on the market. We have to calculate the expected return and risk of the efficient frontier using historic prices, thereby implicitly assuming that the distribution of the asset prices themselves and the correlation between them are constant over time. It has been shown (see [Pet21]) that portfolios built according to the MPT tend to under-perform out of sample. Standard market indices like the S&P500, which are constructed according to market capitalization[15], i.e., the most valuable companies have the highest weight in the index, often yield a higher return than a portfolio constructed according to MPT (see [Lop16]).

The Thompson Sampling algorithm can deal with the possibility that the distributions of prices change over time. To give an example, consider a technology and a chemical company during a business cycle. During the boom phase, the technology company is likely to sell more of its products (e.g., mobile phones, high-resolution TV sets or electric vehicles) as consumers are more likely to spend money while the demand of the products of the chemical company is likely to stay roughly constant. In a recession, the chemical company probably only sees a small dent in the demand for its products because these are used constantly and are hard to replace, while the demand for the products of the technology

[14]However, this might not be the case for assets that exhibit not the best payoff and are therefore less often chosen. In contrast to the epsilon-greedy strategy, once one option is clearly superior to all others, this choice will always be taken in TS. Thus, the distribution of the non-optimal strategies will no longer converge.

[15]These kind of indices are regularly rebalanced as the market capitalization of the index members changes. Hence, "winner" stocks get a higher weighting while "looser" stocks might even fall out of the index.

company might decrease significantly. These developments are usually reflected in the share prices of the two companies: the shares of the technology company are likely to outperform the share price of the chemical company during a boom phase and the opposite is true in a recession. Hence, the expected return as well as the risk of the shares of the companies are not constant over time but vary over the business cycle. This variability over time cannot be adequately reflected in MPT as the distribution of asset prices is assumed to be constant. In contrast, TS can incorporate changes in the underlying distribution of asset prices as new prices are considered when the posterior distribution is updated.

We are considering contextual Thompson Sampling. This means that we are considering not only the distribution of asset prices, but also other time series that provide context. This could be data on macroeconomic variables such as inflation, unemployment, or variables that describe the state of the business cycle to refer to our early example[16]. Thus, TS is more than just a smart updating rule. It offers the possibility to consider correlations between context variables and assets which can be an information advantage over MPT.

We start with the historical mean and covariance matrix of the asset prices as the prior distribution. We assume that there are k assets (or asset classes) that the investor can invest in. The change in price of each class follows a normal distribution, which is unknown to the investor. However, from historical data, she is able to calculate the mean, a k-dimensional vector $\boldsymbol{\mu}$, and the covariance matrix, the $k \times k$ matrix $\boldsymbol{\Sigma}$. Thus, the prior distribution of the k assets is $\mathcal{N}(\mu_{t-1}, \Sigma^2_{t-1})$. The prior distribution is formed based on κ_{t-1} observations.

According to the webpage [Wika] the way to update the multivariate normal distribution, for which the mean and the covariance is unknown, is as follows:

$$\boldsymbol{\mu}_t = \frac{\kappa_{t-1}\boldsymbol{\mu}_{t-1} + n\bar{\boldsymbol{x}}}{\kappa_{t-1} + n}, \tag{11}$$

$$\kappa_t = \kappa_{t-1} + n, \tag{12}$$

$$\boldsymbol{C} = \sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})^T \tag{13}$$

$$\boldsymbol{\Sigma}_t = \kappa_{t-1}\boldsymbol{\Sigma}_{t-1} + \boldsymbol{C} + \frac{\kappa_{t-1}n}{\kappa_{t-1} + n}(\bar{\boldsymbol{x}} - \boldsymbol{\mu}_{t-1})(\bar{\boldsymbol{x}} - \boldsymbol{\mu}_{t-1})^T. \tag{14}$$

[16]We will use the same context variables for TS as for the reinforcement learning algorithm. These variables will be introduced in section 5.

Following algorithm 1, the investor chooses the asset class or the mix of asset classes that maximizes her expected payoff. However, we know from section 2 that in the context of investing, the payoff is not the monetary payment as such but the expected utility from this payoff, which takes into account the risk of the investment. Thus, we apply the same two-step process of the MPT framework to the updated posterior distribution of the asset prices[17]. This gives us a rule on how to incorporate newly gathered information and how to optimally allocate funds thereafter.

In summary, the TS algorithm offers the possibility to efficiently incorporate new information into the process of portfolio construction while still applying risk minimization approach of the MPT framework. So we can use the same two-step approach of first finding the efficient frontier and then determine the CAPM-line using the risk-free rate and the tangency point. Only the distribution used to determine the efficient frontier is here TS' posterior distribution. Hence, we expect the TS to yield a portfolio with a higher risk-adjusted return than the MPT approach as the underlying distribution can contain more information. In addition, we have a second benchmark against which we can measure the performance of the RL algorithm.

4 Reinforcement Learning

In this section, we want to formulate the investment problem in a way that allows us to investigate whether an RL algorithm can be used to improve the investment decision process. Whereas the TS approach gives us an indication on how to update our view on the market and how to adjust the portfolio accordingly, the RL approach offers the possibility of learning an optimal policy function that gives us a probability distribution over the various assets, which can be interpreted as portfolio weights.

The term "reinforcement learning" in general describes a situation in which an agent experiences the environment in different states and reacts to it by choosing an action. She learns how her action influences the environment, which moves to a new state due to her action, by receiving a reward. The goal of the agent is to maximize the rewards she receives after choosing an action in each state. Figure

[17]This ensures two things. First, the investor is again combining the risky sub-portfolio with the highest risk-adjusted return and the risk-free rate to maximize her expected utility. Second, the results are directly comparable.

4 (which is Figure 3.1 from [SB18]) illustrates this idea.

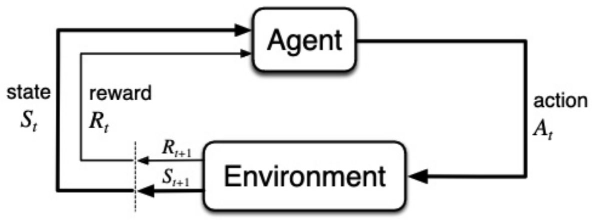


Figure 4: Reinforcement Learning

To derive the necessary framework, we will first define a *Markov Decision Process* (MDP). Next, we will introduce *value iteration* as a concept to learn in a setting in which the environment is perfectly known. We will illustrate this concept using a game called Gridworld.

Then we move to a situation in which the environment is no longer completely known to the agent. We will introduce the concepts of *Temporal Difference* (TD) and *q-learning*. These concepts are used to model the learning process through experience when an agent acts in an environment in which the transition probabilities from state s to state s' are not only unknown to her but which are furthermore probabilistic. That is, the chance of going from state s to s' choosing action a can be viewed as taking a sample from a random distribution. This description of the environment fits the problem of investing well, as the underlying distribution of asset prices is also unknown to the investor, changes over time, and is influenced by the actions of all investors together[18].

As the state space is no longer finite (in contrast to Gridworld), learning the optimal strategies cannot be achieved by trying each possible strategy in each state. As we will see, deep neural networks can be used to approximate the q-function or policy function, thus learning the probability distribution of the action space in the various states.

4.1 Model based learning

To introduce some of the concepts we will need for reinforcement learning, we start with a simple game in an environment that is completely known to the agent. Such an environment is sometimes called *model based* (see [SB18]) as

[18]A large investment into an asset might drive up its price but we will not consider this case as the asset classes considered here are traded in so deep and liquid markets that a single investment cannot move the price.

the entire state space and all transition probabilities are known. Hence, there exists a complete model of this environment. Although these assumptions are rarely met, there are already many of the same concepts needed to analyze such a relatively easy environment that will also be used for more complex and thereby more realistic settings.

The first such concept is the MDP. The agent can choose action a from the action space $\mathcal{A}$. The agent is currently in state s of the state space $\mathcal{S}$ and action a moves the environment into state s'. The MDP describes the dynamic of this move or the transition probabilities if this change is probabilistic: $p(s'|s,a) = Pr(S_{t+1} = s'|S_t = s, A_t = a)$. The name of the process indicates that all necessary information that the agent needs to know is given when she knows the state the environment is in. That is, it does not matter how it got in state s, all that matters is the fact that the world is currently in state s, which is known as the Markov property[19].

By choosing the action a in state s, the agent aims to maximize the sum of the rewards she will receive. Here, not only the immediate reward $r = R_t$ is important, but also all future rewards. Hence, it might be beneficial for the agent to accept a smaller immediate reward when it puts her in a state s' in which higher future rewards are likely. However, future rewards are discounted by a factor δ, which is defined to be positive but less than one, so that the sum of all rewards $G = \sum_{i=0}^{\infty} \delta^i R_{t+i}$ is finite. When the problem has a fixed ending, like a poker game, one often looks at an entire episode, meaning the game is played until the end is reached. In these kind of problems the agent can also be modeled to maximize the reward of the whole episode and the discount factor can be set to one.

Gridworld is a well-known game to illustrate various reinforcement learning concepts as it is part of the OpenAI Gym. This is an API where different environments are available to train RL algorithms for various challenges. The game of Gridworld generates a board of size $n \times n$. The tile with the player is somewhere randomly located on the board, denoted by 'P'. The player's objective is to reach the goal, denoted by '+'. If she reaches that tile, she gets a reward of, for example, 10. However, there is a pit on another random tile, denoted by '-'. If the player falls into the pit, she gets a reward of, for example, -100. Lastly, there is somewhere on the grid a wall, denoted by 'W'. The player cannot move

[19]A stochastic process is called memory-less or markovian after the mathematician Andrei Markov, when its future evolution is independent of its past.

onto that tile. If she tries anyway, there is a negative reward of -50 (the size of rewards is arbitrary as long as the reward for winning is positive, every move yields a small negative reward, and the tiles '-' and 'W' carry a large negative reward).

The setting of the game can be downloaded in the form of a python file Gridworld.py (see [Win25c]). The board can then be instantiated using the following code.

```python
from Gridworld import Gridworld
game = Gridworld (size = 4, mode ='random')
```

Listing 1: Instantiate a Gridworld game

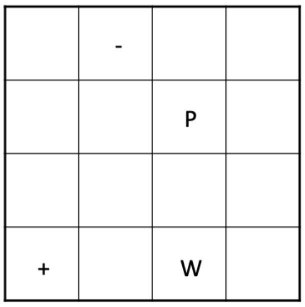


Figure 5: A random Gridworld board

Figure 5 illustrates how a board of this game might look like. The game is given in the form of a numpy-array. The board consists of 16 tiles, numbered from (0, 0) as the upper left corner to (3, 3) as the lower right corner[20]. In this particular instance, the player starts on tile (2, 1) and needs to move to tile (0, 3) to win the game.

The game of Gridworld represents a finite MDP as the agent knows the current state of the world, i.e. her position on the board and the position of the target, the pit, and the wall. Her action space is also known: she can move up, down, right or left. If she is on the border of the board and her next move would take her off the board, she simply stays where she is.

Obviously, the objective of the game is to reach the goal as quickly as possible. This objective is reflected in the reward scheme, where the player gets a reward of -1 for every step she takes. Hence, she has an incentive to reach the goal with the fewest steps possible.

[20]The numbering of the tiles starts at zero and is given by OpenAI.

How can an algorithm learn a general way to successfully play this game since the positions of the goal, the pit, and the wall are set randomly every time? An approach is to use the concept of value and policy iteration. The value of being in state s under *policy* π is given by the so-called *value function*:

$$v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \delta^k R_{t+k+1}|S_t = s\right), \tag{15}$$

where G_t is the sum of all rewards. That is, equation (15) states that the value of being in state s is the expected value of all future discounted rewards under policy π. A policy is simply a strategy which action to take in which state. A policy can be a fixed mapping, in state s always choose action a, or a probability distribution assigning a probability to each available action given the player is in state s.

In Gridworld, the player selects an action a from the action space, i.e., $a \in \{\textit{up, down, left, right}\}$, under policy π, which, when π is optimal, should maximize her expected reward.

Similarly, the value of taking the action a in state s under policy π, denoted by $q_\pi(s,a)$, where we again follow the notation in [SB18], is described by the so-called *state-action* function or *q-function* for short:

$$q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \delta^k R_{t+k+1}|S_t = s, A_t = a\right). \tag{16}$$

The value function returns the expected value of being in state s under policy π, while the q-function returns for a combination of state s and action a the expected value of being in state s choosing action a and following policy π thereafter. Following the equations (15) and (16), the optimality conditions for the value and the state-action-function are:

$$v^*(s) = \max_a v_\pi(s),$$

$$q^*(s,a) = \max_\pi q_\pi(s,a).$$

Using the Bellman optimality condition (see for example [Bel52]) from Dynamic Programming (DP), the value of being in state s under an optimal policy π^* must be equal to the expected return of choosing the best strategy in that state and following the optimal strategy thereafter:

$$v^*(s) = \max_{a \in \mathbb{A}} \sum_{s',r} p(s'|s,a)\,(r + \delta v^*(s')) \tag{17}$$

Hence, the Bellman equation expresses a relationship between the value of being in state s and the value of entering the successor state s'. Equation (17) states that the value of being in state s is equal to the immediate reward the agents receives when moving to state s', r, plus the discounted expected value of the successor state, $\delta v^*(s')$.

For a game like Gridworld with a finite number of states, the Bellman optimality condition can be used to successfully learn how to play the game. First, the value functions of all states are initialized to some arbitrary value, i.e. $v(s) = 0$ for all states $s \in \mathcal{S}$. As the value of $v(s')$ in (17) is now known, we can determine the optimal action a in each state. With the update in the policy of each state, the value function of each state can be updated.

Following [SB18], the iterative relationship between updating the value function and the improvement of the policy is described by the pseudo-code in algorithm 2.

Algorithm 2 Policy iteration algorithm

```
/* Input: MDP=(S, A, P), $\delta \in$ [0,1] */
/* Output: optimal policy $\pi^*$ and value function $v_\pi^*$ */
$i \leftarrow 0$
$\pi_0 \leftarrow$ arbitrary
$v_0(s, \pi_0) \leftarrow 0$
  repeat
      $\forall s \in \mathcal{S} : v_{i+1}(s, \pi_i) \leftarrow \sum_{s' \in \mathcal{S}} P(s, \pi_i, s')[r + \delta v_i(s', \pi_i)]$
      $\forall s \in \mathcal{S} : \pi_{i+1}(s) \leftarrow \mathrm{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s, \pi_i, s')[r + \delta v_{i+1}(s', \pi_i)]$
      $i \leftarrow i + 1$
  until $\pi_{i+1} = \pi_i$
  return $\pi_i, v_i$
```

To illustrate this concept for Gridworld, we start by initializing the value function of all states to zero and choosing an arbitrary policy, i.e., choosing randomly from all available actions. As the value of being in the successor state, s', is zero, the value of being in a state s is identical across all states. Hence, the optimal policy at this stage is randomly choose the next tile since they are all equally good.

Next, we update the value function in each state. Since the value of any following state is zero at this stage, the value of being in a certain state is the immediate reward. When updating the value function by looking at the Gridworld board in Figure 5, we see that the value of being in state (0, 3) is 10, since it is the target tile, and -100 in (1, 0) since it is the pit, the values of the successor states are changed. Given these new values for $v(s')$, we are now in a position to update the policy. As tile (0, 3) can be reached from tile (0, 2) with the action "down" and from tile (1, 3) with the action "left", the optimal policies in these states are clearly identifiable, and the value of being in these states are $v((0,2)) = v((1,3) = -1 + \delta * 10$.

These steps of updating the value function and improving the policy for each state are being repeated until these do not change any longer. Through this process, the distance to the target tile works its way through the neighboring tiles until the shortest way to the target is found. Similarly, the negative value of the pit tile and the wall tile radiates to neighboring tiles, and the agent will adopt her policy to avoid these. Figure 29 in the appendix visualizes the different steps of the iteration process without discounting ($\delta = 1$). You find the Python code for Gridworld in GitHub [Win25c].

Thus, the repeated iteration between updating the value function and policy evaluation (see Figure 6, which is Figure 4.7 from [SB18] for illustration) is a way to learn how to play a model-based game in which the agent knows the entire MDP and in which the number of states are finite so that complete sweeps through all states are possible. But how can an agent learn to play a game when the transition between states are unknown and probabilistic and the number of states infinite?

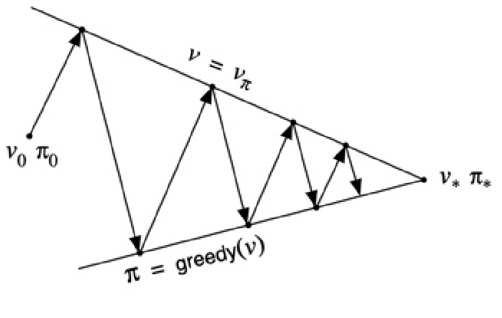


Figure 6: Iteration between value function and policy evaluation

4.2 Temporal Difference, Monte Carlo and q-learning

Learning by using the policy iteration algorithm is only applicable when the state space is finite, since the policy is being evaluated in all possible states, a so-called sweep. However, even for games like chess or Go with bounded state spaces, these can be so large that a single sweep through all possible states would take years for the fastest available computers. Furthermore, in many real-live examples it is also not necessary to evaluate every single state as some will never be reached, i.e. a situation in chess in which one player has only three prices left while the other has not lost a single piece is almost impossible to be reached as the first player is very likely to be chess mate.

Temporal Difference learning is an alternative RL approach in which the value of being in a certain state is updated *while* the agent is playing the game. That is, she is making experiences and the value of being in a state is immediately processed. With every new observation by the agent, the value function is adjusted so that the new experience is compatible with the value function. Based on past experiences, the agent forms an expectation regarding the value of being in a state. The expected value is compared with the current value. If the current value of being in state s, receiving reward r and moving to state s', $r + \delta v_\pi(s')$, exceeds the expected value of being in state s, $v_\pi(s)$, the agent learns that the current policy is beneficial and it is positively reinforced.

The TD update for the value function is

$$v_\pi(s) = v_\pi(s) + \alpha[r + \delta v_\pi(s') - v_\pi(s)], \tag{18}$$

where $\alpha \in [0, 1]$ is the so-called learning parameter. It is important to note, that TD does not need a model of the environment. The transition probabilities between the states do not enter equation (18) explicitly. The transition probabilities are implicitly considered in TD as the states are being visited according the respective underlying distribution. Hence, TD learning is sometimes referred to as *model-free* learning.

Furthermore, note that TD methods are using estimates for the value of being in state s based on earlier estimates. This is guessing based on guesses is often referred to as *bootstrapping.* An advantage of TD learning is that does not require a model of the environment, of the reward structure or of the transition probabilities. It has been proven to converge if the learning parameter is chosen sufficiently small.

Lastly, it is also important to note, that the agent is learning or updating her value function with each single experience. This is referred to as *online* learning. A variation of this approach is the so-called *n-step* learning as the agent will only learn after n actions. The reason to use this approach is that it is often more stable as fluctuations are averaged out (we will discuss in more detail below). The other extreme to online learning is to first play an entire game (if the overall setting permits this set-up) and only then let the algorithm learn from it. This method is sometimes referred to as *Monte Carlo* method. An advantage of this approach is that the algorithm might easier learn strategies that turn out to be successful if the entire game is considered and when rewards occur only at the very end (e.g., chess) and are sparse. Figure 7 (a similar picture can be found in [ZB20]) visualizes the spectrum of the different learning methods.

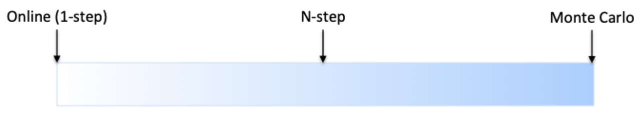


Figure 7: Visualization of the different learning methods

The reinforcement learning approaches TD and Monte Carlo are often used in combination with the so-called *q-learning.* We have already introduced state-action function, or q-function for short, in equation (16). It consumes a pair of state s and action a and maps these to an expected value. The relationship between the q-function and the value function $v(s)$ and the policy π are therefore

$$v(s) = \max_{a \in \mathbb{A}} q(s,a) \text{ and}$$

$$\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}} q(s,a).$$

The TD approach can be applied to the q-function as the agent is updating it with every experience (s, a, r, s') she is making while playing the game. The updating follows the logic of equation (18):

$$q(s_{t+1}, a) = q(s_t, a) + \alpha[r_{t+1} + \delta \max_{a' \in \mathcal{A}} q(s'_{t+1}, a') - q(s_t, a)], \tag{19}$$

where α is again the learning parameter and δ the discount parameter. The agent chooses in the current instant action a and in all following states a', the optimal action in that state. Note that action a does not have to be an optimal action.

It can be a so-called *off-policy* action by which the agent simply learns about the environment. The choice of action a brings the environment from state s to state s' and earns her reward r at time $t+1$. To ensure that the agent is trying all relevant actions, q-learning is often combined with the epsilon-greedy strategy.

Q-learning, also known as Monte Carlo Control method, also offers the possibility to learn successfully in an environment in which the agent does not have complete information. The underlying idea is to learn the value of being in a state, which is the current reward plus expected discounted future rewards of all following states, by averaging the experienced returns whenever the agent was in that particular state. As the number of visits to that state grows, the average of the experienced returns converge to the expected value of being in that state. Thus, to ensure convergence we have to make sure that the number of visits to each state is sufficiently large. In a model-free environment we have to find q-values, rather than state values, as these are already sufficient to deduct a policy (the policy is to choose the action that gets the agent into the next state with the highest value function).

In a model-free environment our goal is to estimate the q-function $q_\pi(s,a)$, the expecting return of choosing action a in state s and thereafter following policy π. If we have an estimate of $q_\pi(s,a)$, we can deduct the corresponding policy as $\pi = \text{argmax}_a \; q_\pi(s,a)$. The trick is to use a neural network to estimate the q-function. The final problem with this approach is that some state-action pairs might never be visited. For the Monte Carlo Control estimation of the q-function to work, we have to implement a continued exploration. An often used approach is to ensure that every state-action pair has a non-zero probability of being visited. This guarantees that, if the number of episodes approaches infinity, every state-action pair is being visited sufficiently often (in fact infinitely often). The following pseudo-code describes this approach (see also equation (18)).

Algorithm 3 Monte Carlo Control algorithm

```
/* Output: $q^*(s,a)$ and $\pi^*(s)$
initialize $q(s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily and $q(terminal, \cdot) = 0$
initialize $\pi(s) \forall s \in \mathcal{S}$ arbitrarily
initialize empty list $returns(s,a)$
  repeat
      choose $s_0$ and $a_0$ randomly
      generate an episode starting from $s_0$ and $a_0$ and following $\pi$
     for each pair $s$ and $a$ in the episode do
      append return to list $returns(s,a)$
      $q(s,a) \leftarrow$ average($returns(s,a)$)
     end for
     for each $s$ in the episode do
      $\pi(s) \leftarrow \text{argmax}_a \, q(s,a)$
     end for
  until convergence criteria is satisfied
```

In comparison to the policy iteration process, the Monte Carlo Control approach is simpler. First, the q-function and the policy is initialized arbitrary. The agent then follows some random action and random policy and receives some rewards. Based on these experiences, the agent updates the q-function and based on the new q-function and the current state, she updates the optimal policy in this state. The algorithm stops if there is no more progress occuring.

As one might expect from this description, the algorithm has been shown to have a slower speed of convergence than other learning approaches. Nevertheless, DeepMind used a variation of the q-learning in their "AlphaGoZero" software in 2013 to beat a human grand master in the game of Go. This is considered to be a major break through in reinforcement learning as Go has an even higher number of possible states than chess and it was widely believed that winning strategies could only be found by intuition.

DeepMind combined q-learning with a deep neural network to accomplish this task. That is, they implemented an environment with the rules of Go, used an artificial neural network (ANN) to approximate the q-function and let the software play against itself. By experiencing millions of Go games, AlphaGoZero was able to find strategies no human player had thought of.

Before using a deep neural network of the simple game of Gridworld, we will describe in the next section how neural networks work in general.

4.3 Deep neural networks

Machine-learning algorithms (ML) like Support Vector Machine or Random Forest work well on smaller problems, but they are regularly outperformed by Artificial Neural Networks (ANN), sometimes also called feedforward networks, on larger tasks[21]. The idea to build a computational model inspired by a human brain cell had McCulloch and Pits already in 1943 [MP43]. They proposed a very simple model of an artificial neuron that has one or more binary inputs and one binary output. Like a brain cell it activates its output, i.e., it fires an electric impulse, if a certain number of its inputs are active. McCulloch and Pitts were able to show that a network of these artificial neurons can perform logical computations.

In 1958 Frank Rosenblatt (see [Ros58]) introduced a slightly different version of a neuron, which in itself is already a simplified version of a neural network. Figure 8, which is Figure 10-4 from [Gér22], visualizes Rosenblatt's so-called *Perceptron*.

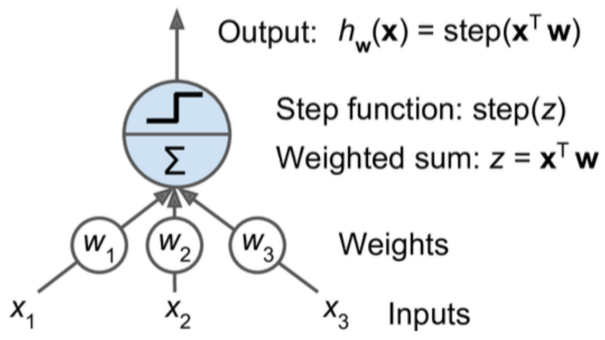

Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{x}^T \mathbf{w})$
Step function: step(z)
Weighted sum: $z = \mathbf{x}^T \mathbf{w}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Figure 8: The Perceptron

In Figure 8 we already see many ingredients of an ANN: there are three inputs, X_1 to X_3, whose outputs are multiplied with a respective weight w_1 to w_3. These three weighted inputs are summed and the cell activates its output if this sum exceeds a certain threshold. In Figure 8 this is determined by a step function while in today's ANNs this would be determined by an activation function.

The choice of the activation function depends on the problem to be addressed. The Sigmoid function is used for binary classification. Its generalization for a n-dimensional case, the *Softmax* function, is used for classification with n possible

[21]This section draws from chapter 10 of [Gér22].

classes:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}} \text{ with } z_j \geq 0 \text{ and } \sum_{i=1}^{n} z_i = 1.$$

The most heavily used activation function is probably the Rectified Linear Unit function, or *ReLU* for short, which is simply

$$ReLU(z) = max\{0, z\}.$$

The ReLU activation function sets any output from the previous layer that is below zero equal to zero while passing on any output that is above zero to the next layer. Although the derivative of the ReLU function does not exist at $z = 0$, it yields very good results in practice.

There are other activation functions like the *Hyperbolic Tangent* function or *Maxout* but since we will not use them in study we will not describe them here.

Note that all activation functions are non-linear. The reason for this is that since the gradient is determined using the chain rule, a linear activation function would imply that a linear transformation in one layer would be chained with another linear transformation of the next layer. This could be simplified to a single linear transformation and we would end up with effectively a single layer instead, which might not be able to pick up more complex patterns in the data.

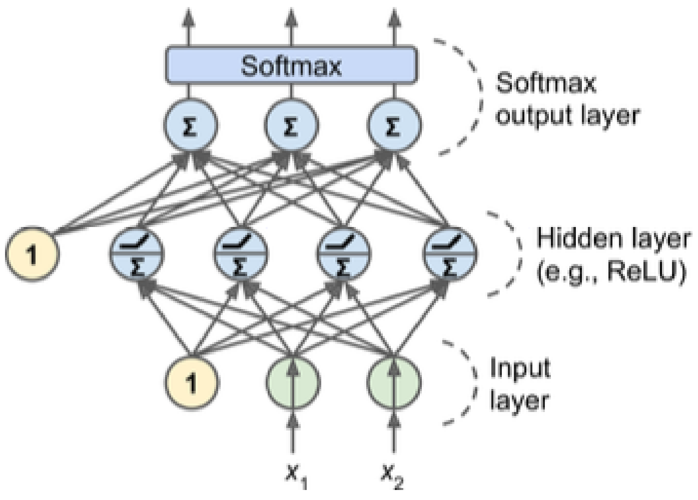


Figure 9: An example of an ANN for classification

Figure 9, which is Figure 10-9 from [Gér22], shows an ANN with three layers build to perform a multi-class classification task. It has one input layer with two neurons, displayed here in green. The output layer has three neurons and the activation function of the output layer is Softmax, which ensures that the output is between zero and one and that it sums overall neurons in this layer to one. Hence, the purpose of this neural network is a classification task among three possible outcomes. The network has one hidden layer with four neurons. The activation function of this layer is ReLU. This is a common set-up: the ReLU function is often used as the activation function in the hidden layers while the activation function of the output layer might be of a different kind, depending on the problem at hand. In Figure 9 the bias term is highlighted in each layer as the yellow cells containing a one. The bias term can be thought of as a neuron whose output is constantly activated. Its signal can therefore amplify the weighted sum of output signals of the other neurons in the respective layer so that it reaches a certain level (i.e., zero in case of the ReLU activation function). Note that all neurons of one layer are connected to all neurons in the next layer, which is called a dense layer. Hence, the number of parameters in this example ANN are: three times four for the first layer (two input plus one bias neuron) plus five times three. So, there are already 27 connection weights or parameters[22] whose values have to be found during the training process of the network.

The general way an ANN works is as follows. It makes a prediction (at the beginning these are purely random), calculates the prediction error by comparing the prediction with the actual data using a *loss function* and finally adjusts the weights between the neurons to minimize this prediction error. Thereafter, these steps are repeated. Thus, we need an efficient way to find a minimum of the loss function by changing the weights between the neurons of the ANN.

One of the prerequisites for ANNs to become such powerful tools was the work of David Rumelhart, Geoffrey Hinton, and Ronald Williams in 1986, who introduced the *backpropagation* training algorithm (see [RHW86]). This is an efficient way to calculate the gradient of the network's error with regard to every weight. The method works in two passes, once moving forward through the network ending in a prediction. Thereafter, calculating the difference between the prediction and the actual data or the label in case of a classification problem. The prediction error is then used to adjust the weights between the neurons accordingly, thereby working backwards through the network. Such a round trip is called an *epoch*, which starts when a data point is ingested by the input layer. The weights of the connections between the neurons are initialized randomly to

[22]Modern ANNs can have billions of parameters and need corresponding amount of compute and time to be trained.

avoid symmetry as this would cause all layers to behave similarly which would effectively reduce the number of layers to one. The output of the input layer is calculated and send to the next layer, the first hidden layer. This step is repeated until the last layer, the output layer, has calculated its output. This completes the forward pass-through.

The idea of the backpropagation algorithm is to calculate the partial derivatives of the weights, i.e. by how much should the weight in the current neuron be adjusted to reduce the prediction error, given the partial derivatives of all following neurons in the network. Thus, by starting in the last layer, the output layer, the weights are adjusted according to

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \alpha \nabla_{\mathbf{W}} L(D, \phi_{\mathbf{W}}),$$

where $\mathbf{W}$ is a vector of weights for the connection between the neurons, α is a learning parameter, L($\cdot$) is a general loss function, D denotes the data set, $\phi_{\mathbf{W}}$ the feedforward network and $\nabla_{\mathbf{W}} L(D, \phi_{\mathbf{W}})$ the gradient determined using backpropagation. Hence, the weights are adjusted by the partial derivative of the loss function given the current prediction error. Once the values are calculated for the output layer, one can move one layer backward to the second to last layer. The partial derivative of the loss function with respect to the weights connecting the second to last layer with the output layer can be combined with the partial derivatives of the weights in the output layer via the chain rule. These adjustment steps are repeated until the input layer is reached and the adjustment of all weights of all connections between neurons was calculated. Once these adjustments are implemented, the next data point is fed to the network, the next prediction error is calculated and the adjustments of the weights are repeated. These steps are repeated until a (local) minimum of the loss function is found.

Hence, the backpropagation algorithm of Rumelhart et al. allowed the usage of already known concept like gradient decent or stochastic gradient decent to efficiently find the minimum of the loss function, thereby training the ANN. Open-source platforms like Pytorch (see [IPK21]) or Tensorflow (see [Dev22]) have pipelines that provide the backpropagation algorithm out of the box. The code snipit in listing 2 shows the relevant steps. First, we define the loss-function, the learning parameter (α in the earlier equation), and the optimizer (here we use Adam which is a popular optimizer using gradient decent to find the minimum of the loss-function (see [IPK21])). Next, we set the optimizer to zero.

```
# define a loss function, an optimizer and a learning rate
loss_fn = torch.nn.MSELoss() # define MSE as the loss function
learning_rate = 1e-3 # define a learning rate
optimizer =
    torch.optim.Adam(model.parameters(), lr=learning_rate)
# input layer
optimizer.zero_grad() # set the optimizer of weights to zero
loss.backward()  # perform backpropagation
optimizer.step() # adjust weights according to prediction error
```

Listing 2: Typical steps for backpropagation in Pytorch

Then we calculate how to change the weights between the neurons for each layer by performing backpropagation. With this result, we adjust the weights according to the previous result and the given estimation error. This completes one training step of the neural network. If these changes indeed reduce the loss (or improve the estimate of the q-function), then the network is truly learning.

This development was later combined with generally more computing power and with the usage of Graphical Processing Units (GPUs) to parallelize the computational steps, which allows for today's training of ANNs with billions of parameters.

To summarize the general algorithm for today's ANNs: for each observation the input layer makes a forward pass to the next layer. In the next layer the weighted sum is the input of the activation function and its output is passed onto the next layer. Once the final output layer is reached, its prediction is compared to the corresponding label and the prediction error is calculated. This step concludes the forward pass. The prediction error is calculated according to a given loss function, which, similar to the activation function, often depends on the type of problem that the ANN is supposed to solve. The prediction error is the start of the reverse pass. After performing backpropagation, the weights of each layer are adjusted in the direction of the gradient to reduce the prediction error.

The success and popularity of neural networks is not only due to the fact that they often outperform Machine Learning algorithms such as Support Vector Machine or Random Forest but also because of their versatility and their scalability. ANNs can be used for classification problems in which case the activation function in the last layer is either the sigmoid function for a binary classification problem or Softmax when more than two classes are possible. These activation functions ensure that the output of the network can be interpreted as a probability since the sum of the outputs over all classes is equal to one. For classification problems, the loss function is usually *Cross-Entropy*, which measures the quality of the implied probability distribution.

But ANNs can also be used for regression problems, in which case the output layer contains the number of neurons equal to the number of regressors that are to be estimated simultaneously. The activation function is generally ReLU (or sigmoid or tanh if the variables are bounded) and the loss function is either mean-squared error (MSE) or mean-absolute-error (MAE).

While ML-algorithms generally do not scale well, ANNs can handle large data sets and often perform better the more data are available.

4.4 Applying deep q-learning to Gridworld

In this subsection we show how a feedforward neural network can learn to play the game of Gridworld. That is, we want to use an ANN to estimate the q-function so that the decision in which direction the player should move is compared to the estimated value of being in that state and choosing that action (i.e., Temporal Difference). If the final value is higher than the expected value, the action is positively reinforced. If the value is less than expected, the ANN learns that the action should not be chosen if the same state is reached again in the future. The corresponding code can be found in the Juypter notebook "Gridworld_q_learning" in [Win25c].

First, we instantiate a neuronal network that can consume the data of a Gridworld board[23]. As the board of a 4×4 game consists of four matrices with 16 positions each (one for the player, one for the wall, one for the pit and one for the target), the input layer must be able to consume an array of 64 data points (the first 16 determine the position of the player and so on). The output layer needs to consist of 4 neurons as their activation will decide whether the player should move up, right, down, or left.

An ANN is used to estimate the q-function which returns an expected value of each of the four possible moves. The index with the highest value of the resulting array determines the best strategy. This strategy is chosen with a likelihood of $1 - \epsilon$. With a probability of ϵ, a random strategy is chosen. Next, the agent makes the move that was just determined to yield the highest expected reward and the player moves to the according tile, which means the environment is in the

[23]The idea is take from [ZB20].

next state, and the resulting reward is collected. The q-function of the following state is determined where it is important to note that for all following states the optimal strategy is being played. The q-value of the following state is discounted by δ. The result, reward plus δ times q-value of the following state, is compared to the estimate of the q-function using the loss-function, which in this case is MSE-function. The resulting loss-function shows, that the model is really learning to play Gridworld as the loss becomes smaller when more epochs are being played.

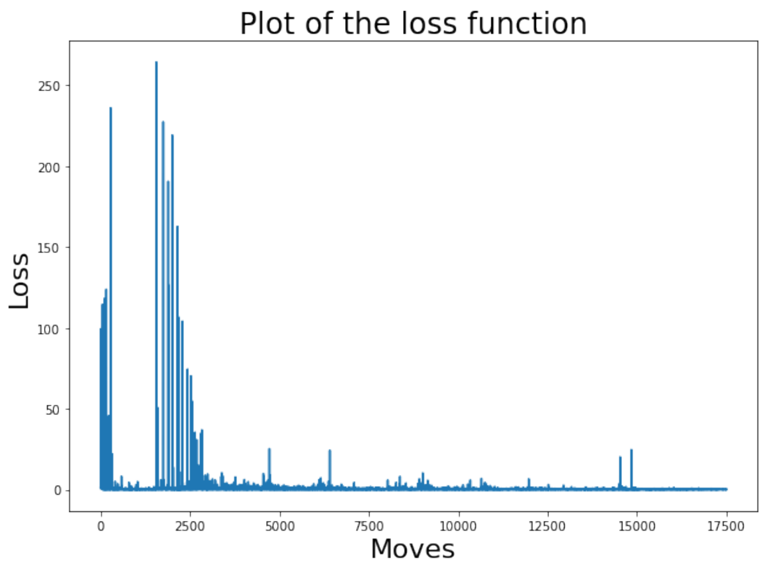


Figure 10: MSE-loss is decreasing as the network learns to play Gridworld

4.5 Related work

In the last subsection of this chapter we briefly describe related work. Due to the success of RL in fields ranging from gaming over robotics to natural language processing, a number of researchers have started applying RL concepts in the field of finance. There are three broad categories in which this has taken place: automated stock trading, risk management, and portfolio optimization.

Since RL deals with maximizing a reward by learning in an at least partially unknown environment, it fits well to the world of automated stock trading. There are multiple research papers published in recent years on this topic (and probably even more research is being done without the goal of ever being published but simply with the motivation of developing profitable trading strategies). Two exemplary publications in this particular field of research are [TE21] and [ZZR19].

The first paper presents an algorithmic solution inspired by the popular Deep Q-Network (DQN) algorithm by Mnih et al. (see [Mni+15]), which has been adapted to determine the optimal position size at any point in time given a particular trading situation in the stock market. The second paper develops a deep learning model to predict price movements from order book data of cash equities[24].

Since one of the strength of neural networks is the handling of nonlinearity, deep reinforcement learning can also be applied to the pricing of derivatives which are often characterized by their non-linear payoffs. An example in which RL is being applied to risk management of such derivatives is the paper [Bue+19]. The authors develop a framework for hedging a portfolio of derivatives in the presence of market frictions such as transaction costs, liquidity constraints, or risk limits. Another example is [Du+20], who also apply RL to derivatives pricing. They use deep reinforcement learning to replicate options subject to discrete trading, round lotting, and transaction costs.

Probably the most work is done in the field of portfolio optimization. Many studies have been published on determining the policy network configuration to determine the optimal composition of the portfolio (see for example [Wan+19] or [Wan+21]). Other interesting work deals with the incorporation of exogenous data such as news stories (an example is [DT20]) or endogenous data, such as company filings (an example for this research area is [Lim+21]). However, most studies deal with the selection of shares to include only into the risky subportfolio of an investor, whereas the present study takes a step back and aims at choosing the optimal weight of each asset class. Hence, the focus of this thesis is how the relationship between asset classes such as bonds, equities, and money market accounts influences the optimal setup of an investment portfolio and if information from context variables can be used to profitably adjust this portfolio.

5 Macroeconomic and synthetic data

The goal of this study is to test if the classical portfolio optimization approach described in section 2 can be improved using a Reinforcement Learning. The latter is known to need large amounts of data from which an algorithm can learn. Although there is data on financial time series freely available, its history is relatively short and the quality often relatively poor. Thus, there might not

[24]Cash equities refers to actual shares to distinguish them from futures and derivatives as these can also be used to create an equity exposure.

be enough data available to train a RL algorithm on. To overcome this problem, we take data from reliable sources and use it to estimate in the following section a *vector autoregressive model* (VAR) that picks up the relation between the time series we are interested in. Then we use this VAR model to generate synthetic data sets repeatedly to train our RL algorithm on. The data is generated with a certain degree of randomness, so that no two data sets are identical but the interaction between the individual time series will always follow the model estimated on actual data. But first, we introduce the data being used in the following subsection. Thereafter we derive the VAR model.

5.1 The actual data

We use monthly macroeconomic data from the Federal Reserve Economic Data base, called FRED (see [MP21] for a description). Additionally, we use Bloomberg as source for the data on asset prices. The time series considered in the following start in January 1969[25].

To get an overview of the available data, we first look at the macroeconomic data, which hopefully contain useful information the RL algorithm can learn from. Figure 11 shows the monthly time series of some US macroeconomic indicators since 1969. The upper left graph shows the US *consumer price index* (CPI) since January 1969 measured in % p.a.. The consumer price index measures the prices of a basket of goods an average US-household consumes. Hence, a large weight in this basket is rent, price of food and clothing (for the exact set-up of the index refer to the U.S. Bureau of Labor Statistics [Lab]). The change in the index from year to year measures the level of inflation in the USA over this period in %.

The upper right graphic shows the unemployment for the US for the same time period. This time series gives a good indication if the economy is expanding (lower unemployment) or contracting (higher unemployment). Here the COVID-pandemic beginning in 2020 is easy to identify.

The lower left graphic is the Non-Farm Payroll (NFP) for the US since 1969 in absolute terms. This number measures the newly created jobs in the non-farming sectors (as these show less seasonality) of the US economy. The NFP number is published on the first Friday of the month at 8:30. a.m. EST time before the stock market opens on Wall Street. This figure is a very good indicator (with a very

[25]Note that we could not find other macroeconomic variables with a longer continuously available data history.

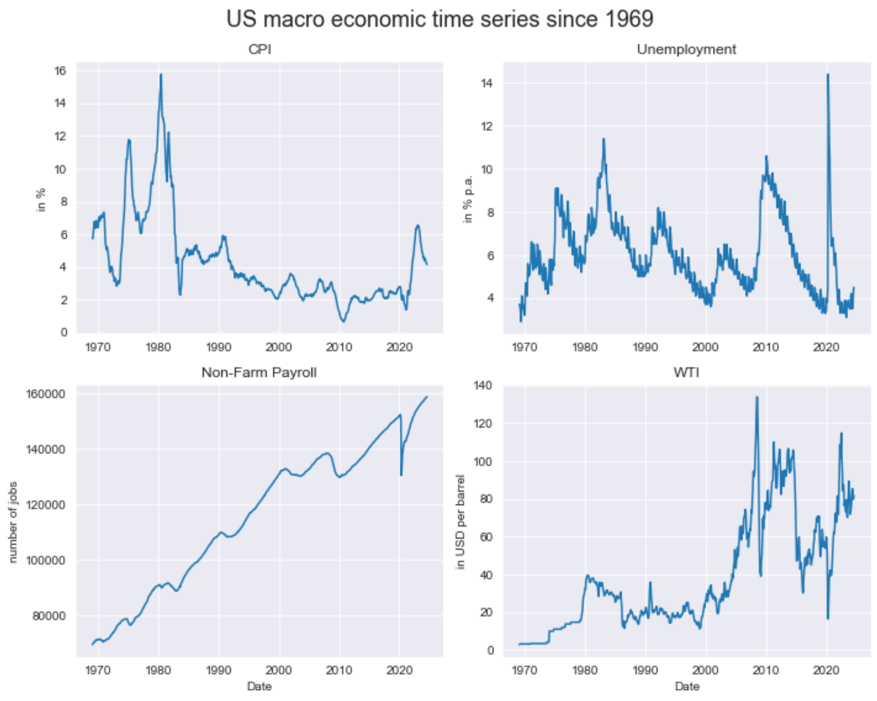


Figure 11: Macroeconomic data from FRED and Bloomberg

short lag) for the strength of the US economy. Its publication is highly anticipated by the markets. Equity indices like the S&P500 or Dow Jones Industrial Average (DJIA) tend to react strongly to any positive or negative deviations of this number from the market expectations. The NFP-number is related to the unemployment figure. But while the latter shows better structural shifts in the economy over a longer time horizon, the NFP number reacts quicker to the business cycle and is viewed by the financial markets as health indicator of the "real economy".

The lower right graphic shows the price of a barrel oil of the kind "Western Texas Intermediate" or WTI. This price reflects the future demand for oil[26] and is therefore also a good indicator of future economic activity in the US.

Next, we show an overview of the four asset classes under consideration. The risky asset classes "equity" and gold are given in price terms (or future points which are calculated by a weighted average of the share prices of the companies in the index) and are shown in blue while the low-risk assets, "US government bond" and "money market", are shown in terms of yield in % p.a. and are drawn in orange for distinction.

[26]Note that we show the price of a so-called future, not the commodity itself. Only if the buyer of the future holds the contract until the settlement day, she will be able to take delivery of the actual oil.

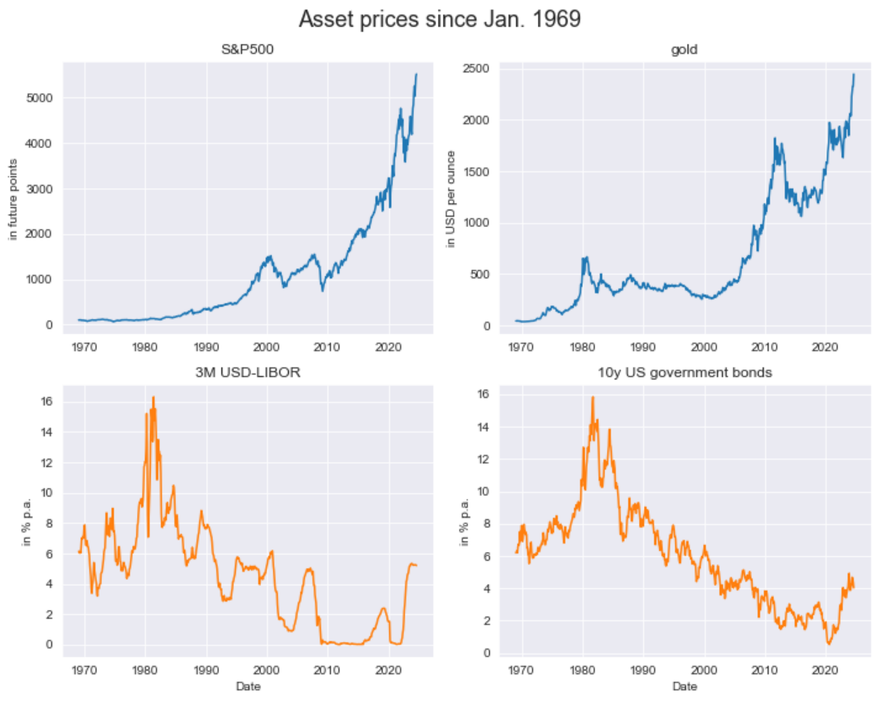


Figure 12: Asset prices from Bloomberg

Figure 12 shows the four different asset classes from which we will construct the investor's portfolio: the S&P500, an index of the 500 largest companies on stock calculated by Standard&Poor's, the price of an ounce of gold measured in USD, an index showing the performance of monthly investments in the 3-month-US-Libor rate, which serves as a proxy of the risk-less asset, and finally an index showing monthly investments into a US-treasury bond with a maturity of 10 years.

To convert the less-risky asset classes into prices, we calculate a synthetic index that starts at 100 and exhibits the performance of a monthly investment into a 10-year US-government bond and a monthly investment into the three-month US-LIBOR. The performance of the latter is simply the accrual of the interest rate, while in the calculation of the performance of the government bond one has to account for the inverse relationship of prices and yield. That is, if the 10-year yield of a US government bond drops by 1BP (= 0.01%), its price goes up by roughly 9BP[27]. For details of the calculation of the index see the code in the respective GitHub repository [Win25b].

[27]In the fixed income world prices and yield are inversely related. This can be seen when considering a bond that was issued at a price of 100% and a yield of 4%. If the yield of the bond drops right after the issuance of the bond by 1BP, the price of the bond increases to roughly 100.09% as investor will receive a coupon of 4% for the next 10 years, which is no longer available on the market.

With this conversion we can compare the performance of the risky and less-risky assets over the considered time period.

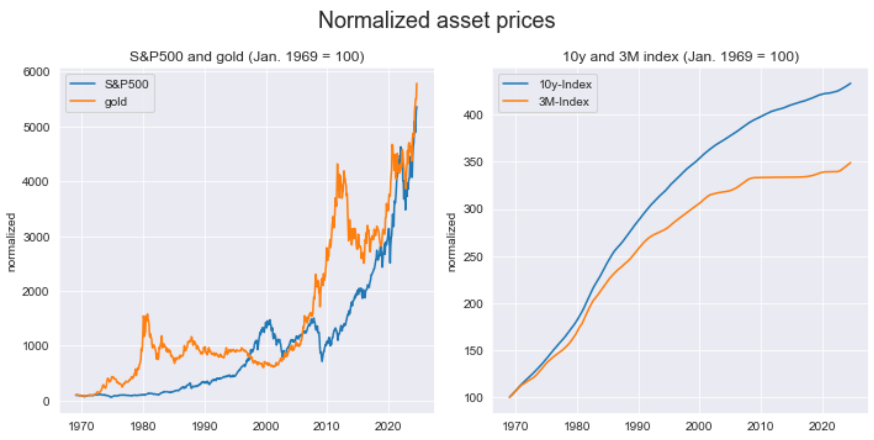


Figure 13: Performance of asset classes in price terms

The left graphic in Figure 13 shows the performance of the equity market and gold from January 1969 to July 2024. We normalize the data to start at 100 at the beginning so that the performance of all asset classes are comparable. Interestingly, although the price of gold rose from 1969 to the beginning of the 1980s and showed little prices increases for the next 20 years, the two risky assets performed very similar of the entire period under consideration. The right graphic of Figure 13 shows the performance of the fixed-income investments. As expected, the performance is less volatile compared to the asset classes equity and gold and the government bond yields a higher return than the less-risky money market account.

5.2 The Box-Jenkins-Method and VAR models

In this section we want to derive a statistical model to generate synthetic data. As can be easily seen by the chart of the S&P500 in Figure 12, some of the time series under consideration exhibit trends. This is a problem when generating data since the expected value of such a trending variable depends on the time at which the expectation is formed. To ensure that we are only considering stationary processes, which are independent of the point in time when the variables are observed, we test for stationerity of each variable using the augmented Dickey-Fuller test (ADF) (see [DF79]). If we cannot reject the null hypothesis that the time series is non-stationary (see jupyter notebook in [Win25b]) for a given variable, we take the first difference: $\Delta X_t := X_t - X_{t-1}$. This step is repeated until we can reject the null hypothesis of the augmented Dicky-Fuller test for this variable. This step is performed for each time series under consideration. As a result, our model will generate ΔX_t instead of X_t. When having found an unbiased estimator of ΔX_t we can calculate X_t by simply adding ΔX_t to X_{t-1}.

Furthermore, economic data as well as asset prices tend to be autoregressive. That is, a variable at time t depends on its own values with some lags. A typical model for such variable is:

$$X_t = a_0 + a_1 X_{t-1} + a_2 X_{t-2} + ... + a_p X_{t-p} + \epsilon_t, \tag{20}$$

where ϵ_t is white noise[28]. The model in equation (20) is called AR(p), which stands for auto-regressive model with period p. Applying this model to economic data have shown that the error term, ϵ_t, is correlated to earlier error terms ϵ_{t-k} for some value k. Hence, ϵ_t is not white noise after all and the time series shows heteroskedasticity, which means that the variance of the error term is not constant. Robert Engle developed in 1982 (see [Eng82]) a test for this phenomenon. This led to extend the AR(p)-model for time series to the auto-regressive moving average model or ARMA(p, q):

$$X_t = a_0 + a_1 X_{t-1} + a_2 X_{t-2} + ... + a_p X_{t-p} + \epsilon_t + b_1 \epsilon_{t-1} + b_2 \epsilon_{t-2} + ... + b_q \epsilon_{t-q}. \tag{21}$$

Here, the question arises which values for the lag terms p and q to use? A common way to answer this question is to use the Box-Jenkins-method named after the statisticians George Box and Gwilym Jenkins (see for example [BO94]).

An often used technique in finding the parameters for ARMA(p, q) as described in equation (21) is to select parameters for p and q, run the estimation of the model, and check if the error term ϵ is white noise. If any kind of structure is detected in the error term, a new model is selected and the analysis is repeated. Following *Occam's razor*, one starts with the simplest model possible and only add parameters if these improve the performance significantly.

Another approach, which we follow and which is also supported by the respective Python packages, is to choose the number of lags by minimizing different information criterion. The usual criteria are:

[28]White noise refers to an error term that is normally distributed with $\mathcal{N}(0, 1)$ and hence contains no systematic information.

- Akaike information criterion or AIC,
- Bayesian information criterion or BIC,
- Hannan-Quinn information criterion or HQC,
- Akaike's Final Prediction Error or FPE.

Usually, one chooses the lag that minimizes the majority of these information criteria, which cannot be described here in detail as this would exceed the scope of this study. We will use the Python package *statsmodels*, which offers a very convenient way to select the order by calculating all four of the aforementioned information criteria.

However, our goal is not to estimate each variable separately but to also account for the relationships between these variables. For this task we are applying the so-called vector-autoregressive model:

$$\boldsymbol{X_t} = \boldsymbol{A} + \boldsymbol{X}_{t-1} + \epsilon_t, \tag{22}$$

where $\boldsymbol{X_t} = (X_t^{(1)}, X_t^{(2)}, ..., X_t^{(k)})$ holds the values of the k variables at time t. One usually uses only a VAR(1) model as a VAR(p) can be transformed to VAR(1) by extending the dimensions of $\boldsymbol{X}$.

To apply the described methodology on the time series of the financial assets in our data set (that is Fed Fund rate, 10y treasury, gold and S&P500), we take the first differences of the time series so that they are stationary (i.e. reject the null hypothesis of ADF-test). We split the data in 80% training and 20% test data with the aim to fit VAR-model on the training data and then predict the four values of the time series for next month on the training data. Hence, we apply the following code to fit a vector auto-regressive model on just the four mentioned time series.

```python
# import package
from statsmodels.tsa.api import VAR
# instatiate model on data
model = VAR(df_train_stationary)
# fit model
model_fitted = model.fit()
```

Listing 3: Estimating a VAR

The code is straightforward. First, we instantiate a vector autoregressive model on the trainings data after the later has been converted to stationary time series.

In a second step, the model is fitted to the data. The model can be summarized using the command "model-fitted.summary()" (see [Win25b]). For a visual test we use the model to predict the asset price for each time series a month ahead and compare these with the actual values out of the test data, which the model has not "seen". Figure 14 shows the four time series over time in black and the

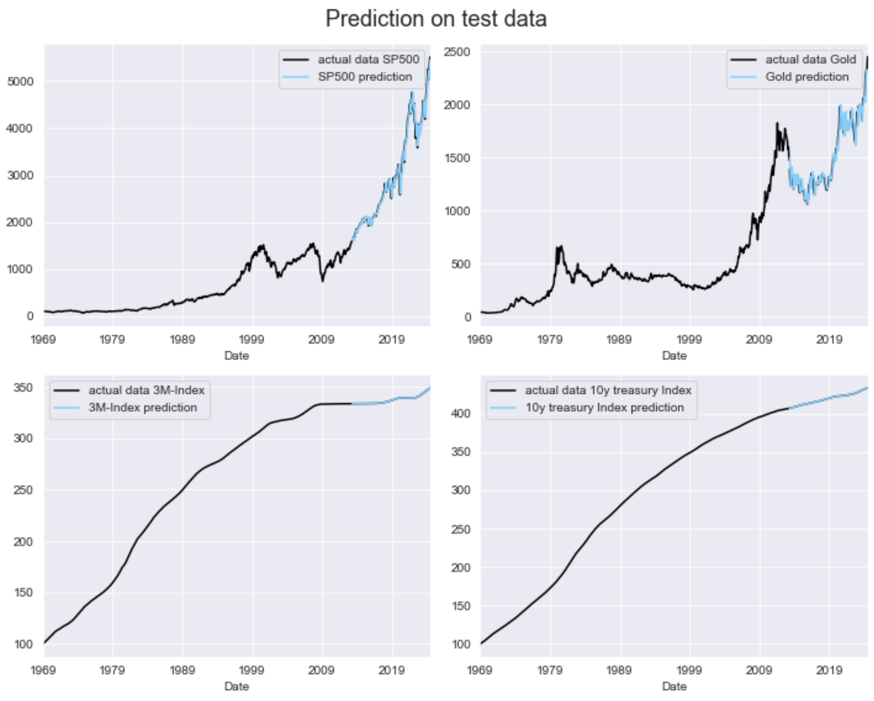


Figure 14: Prediction from VAR model versus test data

prediction of these variables using the test data in light blue one month ahead. On visual inspection the model seems to pick up the details of each variable and the relation between them fairly well. The analysis of the error term, the difference of the actual data and the prediction or, in other words, the difference of the black and the light blue line, reveals no systematic information.

The idea of this exercise is not use the estimated value $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{X}}_{\boldsymbol{t-1}}$ from equation (22), but to generate random data using a var model determined in the way just described that reflects the behavior of and the relation between the time series. Thus, we are now in the position to use $\hat{\boldsymbol{A}}$ to generate synthetic data to train the RL algorithm on.

6 Portfolio construction and performance

In the following chapter we apply the theoretical concepts developed in sections 2 to 4 to the data. To rigorously test our approach, we split the data into a training set consisting of 80% of the observations and a test dataset of the remaining 20%. This allows us to develop different approaches on the training data and to test these in an unbiased way in a latter step as no information of the test data could have influenced the different models during the training phase.

As described earlier, we are comparing three alternative approaches on how to construct an investment portfolio that can invest into the four asset classes: equity, approximated by the S&P500, gold, measured in USD per ounce, US-government bonds, in form of the bond index shown in the right graphic of Figure 13, and a money market account, approximated by the money market index, shown in the right graphic of Figure 13 as well.

In this section we will first describe the specifics of three different approaches. We will show and evaluate the results in the following section.

6.1 Modern Portfolio Theory

The first approach to construct a portfolio follows classic Modern Portfolio Theory as described in section 2. The first question that arises when leaving the theory and applying the ideas to actual data, is how far back should we go to estimate the expected return and the risk, measured as the standard deviation, in order to calculate the efficient frontier and to find the utility maximizing portfolio?

Since the answer to this question is inherently subjective, we decided to use 10 years of data for this calculation. This is an often used time frame, and it seems to be a good compromise between taking a large enough sample to form valid expectations about the distribution and not use up too much of the limited amount of data we have.

Given the concrete data from Bloomberg for the four asset classes, we realize that the correlation of the asset prices and thereby the diversification does not behave as smoothly as the theoretical efficient frontier shown in Figure 1. In fact, given the asset prices the quadratic programming algorithm rarely converges to the optimal risk-return portfolio and often finds only corner solutions.

Therefore, we are using an alternative approach by drawing random portfolio weights and calculating the resulting risk and expected return of these portfolios. When repeating this step often enough, this approach allows us to map the resulting risk-return-combinations in a scatter plot. The following listing shows the most relevant lines of code (see also "find_efficient frontier" in GitHub repository [Win25a]) for this approach.

```
# number of simulated portfolios
n_portfolios = 500000
# simulate random portfolios
for _ in range(n_portfolios):
    # random weights
    weights = np.random.random(n_assets)
    weights /= np.sum(weights)  # normalize to sum equals one
    # calc expected return of portfolio
    portfolio_return =
        portfolio_mu(mu = mu, weights = weights) *12
    # calc portfolio risk (std)
    portfolio_std_dev =
        portfolio_sd(weights=weights, sigma=sigma) * np.sqrt(12)
```

Listing 4: Approximation of the efficient frontier

The explanation of the code is straight forward. We are drawing random weights for the three risky asset classes, normalize these so that they sum to one and calculate the expected return and risk of the random portfolio using the respective function "portfolio_mu" and "portfolio_std". These functions take as inputs a vector of three weights and the expected return respectively the standard deviation of the three time series and return the corresponding values of these measures for the portfolio. The two resulting lists, "portfolio_return" and "portfolio_std_dev" can then be drawn in a two dimensional graph using a scatterplot.

Figure 15 shows the 500,000 random portfolios in the risk-return-space. Each dot depicts one portfolio where the y-coordinate represents the expected return measured in % per year (p.a.) and the x-coordinate represents its risk measured as its standard deviation in % p.a.. The color of the dot represents the risk-adjusted return, which is related to the Sharpe-ratio as given in equation (9), except that only the expected return is divided by its standard deviation and not by the excess return (expected return minus risk-free rate). The color bar on the right-hand side shows the different levels of the risk-adjusted return. Comparing Figure 15 to Figure 1 shows how much the actual result deviates from the theory when applying the concepts to real-world data. But still, the figure demonstrates the fundamental trade-off between risk and return: only if an investor is willing to accept more risk, she will earn a higher expected return on her portfolio.

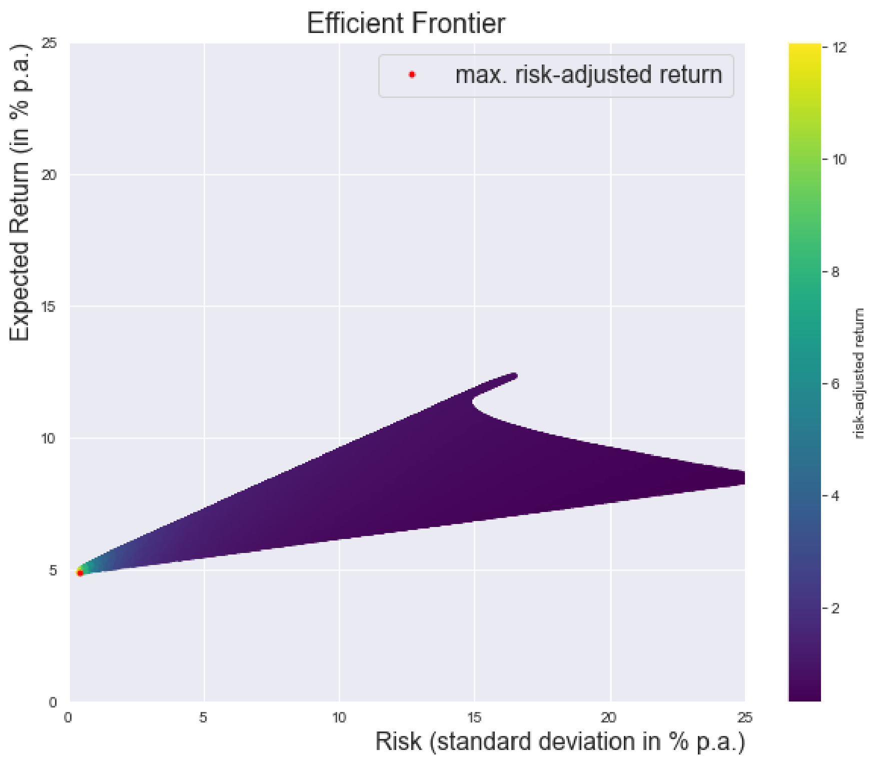


Figure 15: Approximation of the efficient frontier

We notice that with the given three asset classes it is possible to construct an almost risk-less portfolio. This is due the fact that the US government bonds by nature carry very little risk[29] but additionally these bonds performed extremely well during the 1970s. In this period the USA came out of a phase of high inflation and high interest rates due the oil crisis. The return of these bonds were therefore relatively high and as interest rates came down and their prices went up. Hence, they were even less risky than usual during this time period.

This poses a problem in our analysis, since we assigned the role of the risk-less asset to the money market account, whose characteristics are difficult to distinguish from the US bond index. Hence, we drop the US government bond index from list of risky assets[30] and repeat the creation of random portfolios in order to get an idea of the resulting efficient frontier again.

Comparing Figure 16 to Figure 1, we see that this result is closer to the theory as parts of the hyperbola can be identified (in the three assets case the efficient

[29]Government bonds of the USA are, despite the fact that they longer have the highest rating category of agencies like Standard&Poors, considered to be "safe haven" asset into which investors allocate their wealth in times of severe crisis.

[30]However, we keep the time series and add the bond index to the list of context variables.

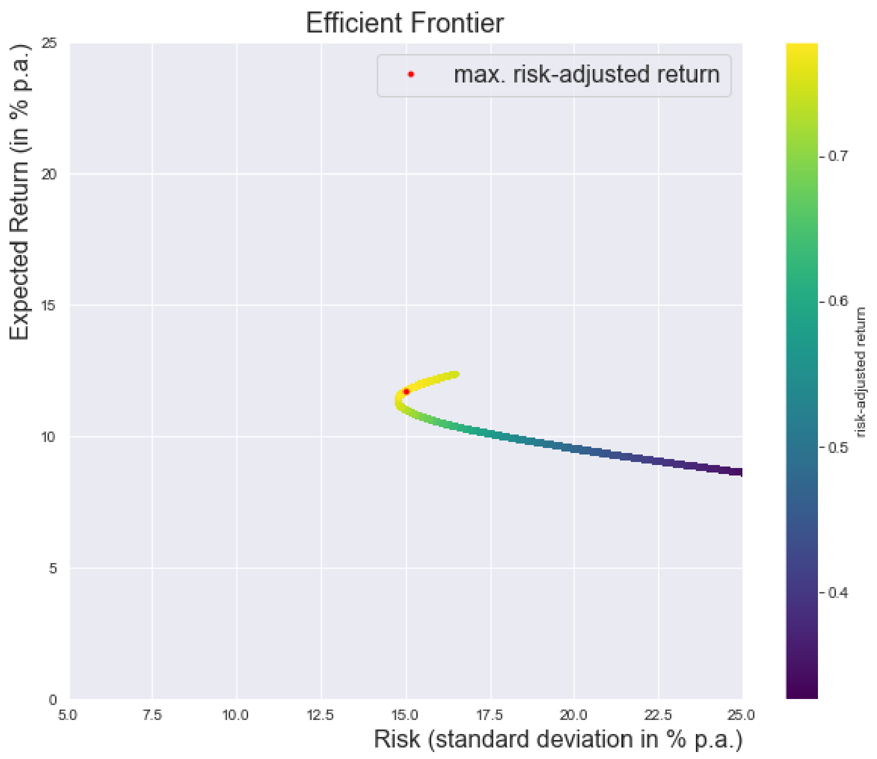


Figure 16: Approximation of the efficient frontier for S&P500 and gold

frontier collapses to one point in the μ-σ-space). The shape of this efficient curve, or its approximation, is due to the specific correlations among the two remaining risky asset classes under consideration: gold and S&P500. In figure 13 we can see that the two risky assets are correlated over the time period under consideration. Hence, the degree of diversification between asset classes is relatively small, which explains the shape of the efficient frontier.

In Figure 16 we indicate the portfolio with the largest risk-adjusted return with a red dot. Based on the data from December 1969 to December 1979 this portfolio consists of 83.6% shares and 16.4% gold. Next, we draw the Capital market line from the risk-free rate to the tangency portfolio of the efficient frontier. For illustration, we show this in Figure 30 in the appendix. Note that the CAPM-line depends on the expected return and risk of the assets via the shape of the efficient frontier as well as on the level of the risk-free rate as the latter influences the location of the tangency portfolio. With a risk-free rate of, for example, 3% the tangency point is at 86.9% of investment in the S&P500 and 13.1% in gold. Hence, relative to the portfolio with the highest risk-adjusted return, it is optimal to increase the equity weight even further and reduce the weight of gold accordingly as the investor has now the option to invest some of her money into the risk-free money market account, which in combination with the risk asset class "equity" offer an even better risk-return-ratio than gold alone.

Following convention, we assume that the investor is risk averse, and hence will invest somewhere on the CAPM-line. If this happens to be closer to the point (0, risk-free rate), which indicates 100% of the funds going into the money market account, or close to the tangency point of 86.9% in equities and 13.1% in gold and 0% in the money market account, depends on the degree of risk aversion of the investor, which we model with the parameter γ. Hence, given a level of risk aversion, we have found the weights of the three asset classes, which constitute the investor's portfolio. This portfolio can serve as a benchmark in terms of risk and return to the alternative approach for portfolio construction using TS and Reinforcement Learning, which will be described next.

6.2 Thompson Sampling

The second approach to construct a portfolio out of the three given asset classes uses the MPT portfolio as input. As described in section 3, the TS approach does not take the distribution of asset prices as described by the expected return and the standard deviation of the three time series as constant over time, but regularly updates these distributions when new asset prices and context variables are observed. Hence, we can expect the resulting portfolio to vary more overtime while the MPT-portfolio only changes if the risk-free rate and with it the tangency portfolio changes.

From algorithm 1 we know that we need to start from a prior distribution. In accordance with the MPT approach, we choose the historic mean and standard deviation of the first ten years of data as the prior distribution. This distribution is updated according to equations (11) to (14) with new observations of asset prices and context variables. Following algorithm 1, we sample from the updated distribution, choose the portfolio weights accordingly, and then observe the reward or return of the resulting portfolio. However, due to the size of the capital markets in relation to the investments of our investor, her investment decision will have no influence on asset prices[31]. Thus, the investor's actions do not change the environment. Hence, we modify our approach here slightly as the influence

[31]There are very few investors who can change asset prices with their investment. We assume that our investor does not belong to this very small group of professional investors around Warren Buffet.

of the chosen actions on the outcome of the experiment is simply not given. This is of course different in the situation of the gambler choosing the optimal slot machine, as there is a direct link between the choice of the slot machine and the reward the gambler receives due to her action.

Hence, our adjustment of the Thompson Sampling algorithm to the investment decision process changes the algorithm to an updating rule of the assumed underlying distributions of asset prices. But in contrast to a simple updating rule, contextual Thompson Sampling updates the posterior distribution not only based on changes of the asset prices but also variation of the contextual variables are considered. That means that external variables such as macroeconomic time series that describe the state of the economy can have an impact on the posterior distribution of the asset prices. Thus, TS has an information advantage of the MPT approach for two reasons: first, it receives new data on a regular basis and second, with the context variables it has access to additional data.

When looking at the update rules described by equations (11) to (14), we notice that the distribution of the covariance matrix of the posterior distribution could in theory increase infinitely. To prevent this, we apply a common trick which does not influence the update or the general behavior of the posterior distribution. We scale the covariance of the posterior function by calculating the ratio of the norms of the posterior distribution to the prior distribution. Note that the covariance of the prior distribution is simply the covariance matrix of the time series of the asset prices for the first ten years (i.e. 120 observations). The idea of scaling down the norm of the covariance matrix of the posterior distribution comes from numerical error analysis, where the matrix norm is used to measure the sensitivity of a numerical calculation if some values of the matrix are incorrect. Thus, by scaling the covariance, we ensure that the sensitivity of the numerical behavior of the posterior distribution cannot deviate too much from the behavior of the distribution coming from the historic data. To illustrate this point, we show the corresponding lines of code in the in listing 5. Note that we have described Thompson Sampling in the form of a Python class so that the repeated steps to update the distribution can be conveniently called via functions of this class (for details, see [Win25e]). The prior and posterior distributions are stored in the respective objects of this class (e.g., "self.prior_sigma" and "self.posterior_sigma" hold the corresponding covariance matrices). To calculate the matrix norm we are using the function "norm" from the Numpy package via the prefix "np.linalg". We tried varying values for the threshold (here 1.5), but the impact on the results is limited. The overall behavior only changes if we set the parameter above 10, i.e, allowing the covariance matrix to increase substantially.

```python
# check covariance matrix to prevent it from exploding
scale_factor =
    np.linalg.norm(self.post_sigma) / np.linalg.norm(self.
   prior_sigma)

if scale_factor > 1.5:
    self.post_sigma =  self.post_sigma / scale_factor  # scale
   covariance matrix back
```

Listing 5: Scaling the norm of the covariance matrix of the posterior distribution

Upon receiving new asset prices on a monthly basis, the investor updates her belief about the distribution, i.e., the posterior distribution is formed. The posterior distribution is used as input to find the investor's utility-maximizing portfolio weights similar to the MPT-approach. Note, however, that the approaches are similar at the outset, as the starting prior distribution is characterized by the same historic mean and standard deviation as the in the MPT approach. Thus, we expect to see a separation of the two portfolios only after some new information has been processed by the TS algorithm.

Hence, we can use the same functions to approximate the efficient frontier and to determine the tangency portfolio as in the MPT approach only using the investor's posterior distribution instead of the historic mean and standard deviation. Therefore, we expect the TS approach to perform similarly to a *momentum strategy*.

Some investors use a momentum strategy to identify a trend in asset prices and assume that the asset price will also follow this trend for some time into the future. Correctly identifying such a momentum allows the investor to enter into a trade that will be profitable if the asset prices indeed continue to move further in that same indicated direction.

We expect the Thompson Sampling approach to give us some similar results to a momentum strategy, as the regular updates of the prior distribution with the new observation of asset prices should favor an increase of the portfolio weights of those assets that have performed best lately. If this is the case, we can expect the resulting portfolio to have a higher return but also a higher risk than the portfolio of the MPT approach.

6.3 Reinforcement Learning

The third and last approach to construct an investment portfolio is using Reinforcement Learning (RL) as described in chapter 4. This approach is fundamentally different to MPT and TS.

First, using RL allows us to directly use context information in the decision process[32]. That is, we provide the RL algorithm not only the time series of the asset prices but also with additional information in the form of macroeconomic variables such as CPI or the price of oil. The aim is that the RL model is able to retrieve additional valuable information from these context variables and adjust the weights of the investments into the corresponding asset classes accordingly. If this is the case, then this approach would have a fundamental advantage over the MPT as its investment decision is solely based on historic asset price data. RL should also have an advantage of TS in this respect, since the latter uses contextual information only indirectly. If the RL model is able to pick up information from the macroeconomic time series, it has the potential to implement beneficial adjustments of portfolio weights earlier than the Thompson Sampling approach, which should lead to a better performance, i.e., a higher annual return.

Second, the RL model directly delivers portfolio weights for each of the three asset classes. In contrast, in the MPT and TS approach we found the portfolio weights of the equity and gold by optimizing the risk-return profile of the portfolio while the weight of the investment into the money market account was implicitly found in a second step via the risk-aversion of the investor. Only in the second step, we determine the position of the optimal portfolio on the CAPM-line and thereby the weight of the money market account and hence the final portfolio. Since the RL algorithm explicitly models the investment in the safe asset, it has the possibility to capture additional information from the correlation among the three asset classes. If, for example, the return of the money market is likely to increase in the future due to rising inflation, the algorithm could already reduce the weight for equity and increase the weight for the safe asset.

In the following subsection, we describe the RL-approach in detail. Here we investigate if it is possible to find valuable information in the five context variables (i.e., CPI, unemployment data, Non-Farm Payroll, 10-year US government bond, and the price of a barrel oil of the type WTI) to improve the portfolio construction process.

[32]TS also uses contextual information but only indirectly via the posterior distribution.

6.3.1 Implementing RL algorithm using a neural network

We use Pytorch to build the neural network as this framework offers a convenient way to apply the backpropagation algorithm with automatic differentiation. Furthermore, it offers all common activation and loss functions and very powerful optimizers.

We follow the standard practice to construct the neural network by defining a separate class (see the Jupyter notebook "reinforcement_learning" in[Win25d]), which inherits from Pytorch's "nn" class, which stands for neural network. This class has two main functions. First, the forward(·) function simply ingest a row of data and produces three portfolio weights. Second, the train(·) function calculates the value of the loss function given the current set of parameters, stores this loss, counts the number of times the model has been trained, resets the optimizer to zero, starts the backpropagation of the entire network in order to calculate how to adjust the weights between the nodes of the network, and finally adjusts these weights given the current prediction error. As described in section 4, the model is trained by minimizing the loss function. Hence, this final step completes one round of learning and the choice of the optimizer, the learning rate, and the loss function has a significant impact on how well the neural network can learn the task at hand.

There are two additional functions in this class. Their purpose is to analyze the performance of the network by inspecting the value of the loss function. If the network indeed learns and is able to increase the reward by retrieving information from the data, then the loss should go down with the number of iterations, called epochs. To verify this, one function retrieves the value of the loss function of the current epoch, while the second plots the value of the loss function to allow for a visual inspection.

When instantiating the neural network class, a network with two hidden layers is being built. That is, the network has four layers in total, but the input layer and the output layer are determined by the problem. The input layer has eight nodes to consume one row of data at time: three observations of asset prices plus five context variables. These nodes are fully connected to the second layer, meaning every node from the first layer has a connection to each of the nodes of the second, the so-called first hidden layer. The second layer consists of 20 nodes. Each connection is assigned a weight or represents a parameter that needs to be set optimally. The activation function of the second layer is the ReLu function, which is quite commonly used in neural networks, since the performance using this activation function is often very good. Furthermore, ReLu is very efficient since its derivative is either zero or one. Hence, the backpropagation algorithm works very fast when ReLu is chosen as the activation function.

The third layer, or second hidden layer, is similar to the second except for the number of nodes. The number of nodes in the last layer, the output layer, is again determined by the problem. In this setting, the number of nodes of the output layer is equal to three, as we have that many assets to invest in. The activation function is the softmax function. Hence, the network delivers three numbers that sum up to one. These can be interpreted in some use-cases as probabilities but in our case as portfolio weights for the investments into the three asset classes.

The following code snippet shows the construction of the neuronal network as well as some other important parameters:

```python
# define model
l1 = 8 # input depending on number of time series
l2 = 20
l3 = 60
l4 = 3 # three assets

self.model = nn.Sequential(
    nn.Linear(l1,l2),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(l2,l3),
    nn.Linear(l3, l4),
    nn.Softmax(dim=0))

# define loss function
self.loss_function = nn.MSELoss()

# define optimizer
self.optimiser = torch.optim.Adam(self.parameters(), lr = 0.001)
```

Listing 6: Definition of the neural network

The first lines of code define the neural network. The *nn.Sequential* function allows us to define each layer separately, which is then stacked on top of each other. Next, we define the loss function, which is to be optimized. Here, we choose the mean-squared-error function. That is, any difference between target value and current value is multiplied by itself to become a positive error value, and the weights of the neural network are in the subsequently step adjusted to reduce this error. We will discuss the reward and the target function in more detail below.

In the following line, we choose the "Adam" optimizer from the available set of optimizers in Pytorch. "Adam" uses stochastic gradient decent and is considered to be one of the standard optimizer packages due to its performance and speed of convergence (see Pytorch documentation for details [Pas+17]). Note also, that we are defining the learning rate of the optimizer in the same line of code. This parameter has an impact on the step size of the gradient decent. We have chosen a standard value for the learning rate. We tested different settings for this parameter, but we could not find a value that improved the results during the training phase of the model.

In contrast, a parameter that has a large influence on the results is the parameter "batchsize". To see its impact, we have to first describe the actual learning process. The algorithm consumes the asset prices and simultaneously the value of the context variables. Given this information, it defines the weights for the investments in the three asset classes. At the beginning, these are purely random, but as the network learns in each period which asset performed best, and hence what the optimal portfolio weights should have been, it learns overtime in which context the weight of which asset class to increase and all other weights to reduce.

With the parameter "batchsize" we steer when and how often the model is trained. To see that this parameter has a significant impact on the results, imagine that the algorithm is trained after each period (i.e., online learning). By chance, asset x performs really well in one month and really poor in the next. Then the algorithm would raise the weight of asset x after the first month only to completely reverse this and lower the weight in the following month. Thus, these fluctuations would make it more difficult for the model to learn systematically from the data. Therefore, we train the model only after a certain number of periods with the average values of the performance of the assets and the corresponding averages of the context variables. Taking averages smooths out the fluctuations, and the algorithm is in a better position to retrieve valuable information. The number of periods after which the algorithm performs the backpropagation step and updates the internal weights to minimize the loss function is defined by "batchsize".

This brings us to the explanation of how we implemented algorithm 3, how the quality of portfolio construction decisions is reinforced, and ultimately how the model learns.

When referring to the pseudo-code of algorithm 3, we notice that "batchsize" determines the number of observations over which we calculate the average of *return(s,a)*. But how can we find the action that maximizes the q-function in each episode which allows us to find the optimal policy $\pi(s)$? This step is implemented by comparing the return of the current portfolio with the return of the optimal portfolio in each iteration. During the training phase we know the optimal portfolio in hindsight. The optimal weights of that portfolio are a vector of zeros for all assets except the one asset with the highest pay-off in this month. Hence, the vector carries a one for that particular asset and zeros for all others.

We calculate the difference of the return of this optimal portfolio and of the portfolio with our current weights using our loss function. Since we are minimizing the loss, we are effectively looking to find the portfolio weights that maximize the return of our portfolio. Thus, we are aiming to find the action that maximizes the q-function or, in other words, to determine $\operatorname{argmax}_{a \in \mathcal{A}} q(s, a)$. The relevant lines of code to implement this idea are given in listing 7.

```python
# get current portfolio weights and profit
prob = nn_model.forward(torch.from_numpy(obs).float())
profit_period = calc_return(df = data, i = i , prob = prob)
# find best strategy
best_investment = calc_return(df = data, i = i , prob = torch.ones(3)) # gets pay-offs of current month
profit_best = torch.Tensor(find_best_dis(best_investment)) * best_investment  # profit of current best strategy
vec_profit_best = torch.Tensor(find_best_dis(profit_best)) * profit_best  # profit of current best strategy
# sum over periods
sum_profit_period += profit_period
target_array += vec_profit_best
# train
if i % batch_size ==0:
    nn_model.train(torch.div(sum_profit_period, batch_size), torch.div(target_array, batch_size))# learn
```

Listing 7: Implementation of the Monte Carlo Control algorithm

In the first line, we receive the weights of the current portfolio by asking the neural network to do a forward pass with the latest row of data. In the next two lines, we determine what would have been the best strategy and the corresponding profit using the function "find_best_disc(·)". The best portfolio exhibits a weight of one for the best performing asset and zeros for the other two. The corresponding profit of this portfolio is simply the performance of the best asset in that month.

The profits of these two portfolios are divided by the number of periods given by "batchsize" (the command is "torch.div(·)"). The two averages are fed into the loss function by calling the function "nn_model.train(·)". In this function, the model minimizes the mean-squared-error between the profits of the portfolio based on the weights suggested by the model and the weights of the ideal portfolio by adjusting the parameters of the neural network. To achieve this, it performs a backpropagation step to calculate how to adjust the weights. This is followed by implementing this adjustment given the current error. If the model learns, then the weights will be adjusted so that the weights of the better performing assets will be increased while the other weights are decreased. Hence, the difference of the profits between the portfolio suggested by the model and the portfolio with the best performing asset over the current time period will be reduced.

The last topic we need to address when implementing algorithm 3, is when to stop. The pseudo code only says "until convergence criteria is satisfied". The question of how long to train a neural network has to be answered not only in this setting but also in every RL application. On the one hand, we do not want to stop training too early when it is still possible to learn more from the data. On the other hand, we do not want to train the model too long so that it starts to overfit the data, which would have negative implications on its performance on "unseen" data. The term overfitting describes a situation in which a model stops to learn the underlying relationship between features and the target variable and starts to memorize how to minimize the loss function given the data. We show a graphical illustration of this idea (taken from [mat25]) in Figure 17.

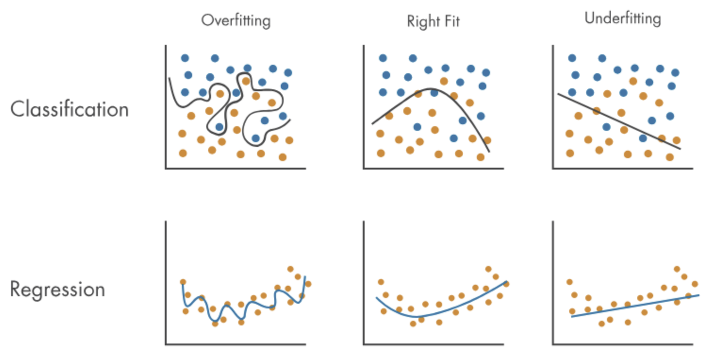


Figure 17: Illustration of over- and underfitting

One way to address this problem is by using the *early stopping* concept. The idea is to observe the loss function and to stop the training process if no new minimum is found for a given number of iterations. Once this number is reached, the algorithm goes back to the parameter settings with which the minimum of the loss function was achieved. Often, this concept is applied to a specific data set, which has been separated specifically for this purpose, the validation set. However, due to the fact that we have only a relatively small data set, we also use early stopping on the training data as well.

We have implemented this concept in the class ”early stopping”, which measures the improvements in the loss function and counts the number of iterations in which the loss function was not improved further. For details, see the Jupyter notebook ”Monte Carlo Control.ipynb” in [Win25d].

Obviously, an important question is whether the model is learning at all. In other words, is it possible to retrieve information from the context variables that enables the algorithm to adjust the portfolio weights so that the difference to the optimal weights and thereby the loss function is reduced? To answer this question, we first plot the loss function in Figure 18.

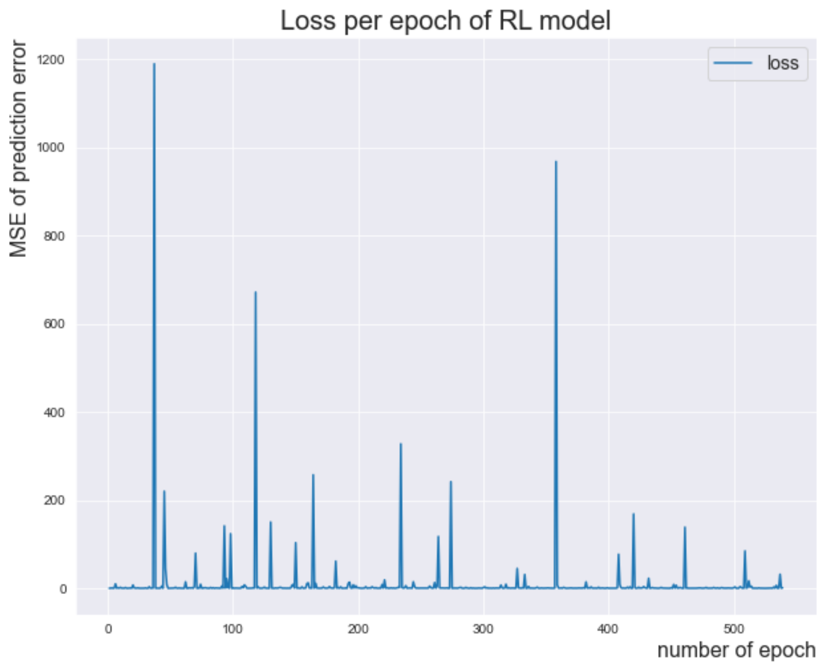


Figure 18: Value of the loss function per iteration

The plot of the loss function is not as clear-cut as we hoped it to be, but there is a trend visible that the loss is getting smaller with the number of iterations.

This gives us some indication that the algorithm is indeed capable of learning from the data. In our setting, this means that the model is finding information in the data that enables it to set the portfolio weights so that the reward for the investor is maximized.

We receive further confirmation of this point when plotting the portfolio weights of the assets at the beginning of the learning process and after a couple of rounds of adjustments. Figure 19 shows the weights for the three asset classes right after the start of the iterations (top left graphic) and after several[33] rounds of learning (top right graphic). The two graphs below show the investment return of the two resulting portfolios at this stage of the learning process. Note that these results are calculated on the test data, which the model has not "seen" since it was only trained on the training data.

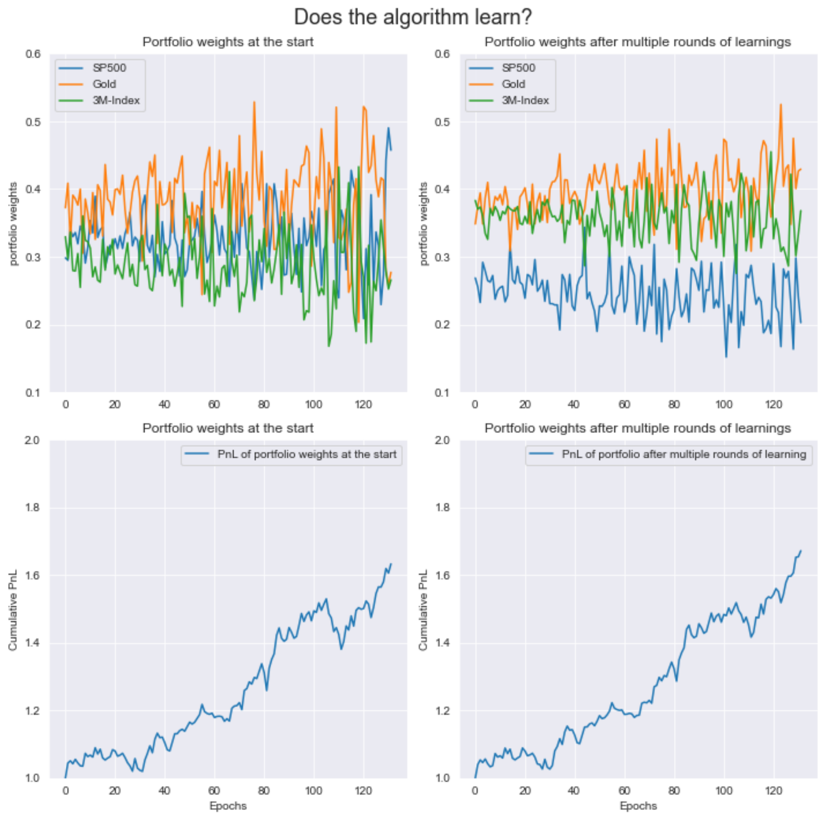


Figure 19: Change in portfolio weights during learning process

The two graphs in the top row of Figure 19 show the portfolio right after the start the algorithm and after multiple rounds of learning. The left graphic shows

[33]Note that the number of rounds the model learned is not fixed. As the algorithm starts with random weights and we are using early stopping to prevent the model from overfitting, the number of iterations is itself random.

the weights after the initialization of the model. These seem purely random. In the graphic on the right-hand side, we see that the weight for gold is raised (plotted in orange) compared to the other two weights. Furthermore, the weight of the money market account (plotted in blue) is significantly lower compared to the other two weights but also compared to itself right at the start of the learning process (top left graph). Hence, the learning rounds definitely had an effect on the weights. Not surprisingly, the overall PnL of the resulting portfolio also increases with more rounds of learning, which is shown in the two graphs in the second row of Figure 19.

The indication that portfolio construction might be improved with a Reinforcement Learning algorithm that retrieves additional information from some context variables is in itself an important finding of this thesis. With this insight, we can now begin to compare the portfolios that are constructed via the three alternative approaches: Modern Portfolio Theory, Thompson Sampling, and Reinforcement Learning.

7 Evaluation

In the following section we evaluate the three methods to construct an investor's portfolio first on the training and then on the test data set. Due to the results on the latter, the more important test as neither approach has "seen" the data, we modify the RL algorithm with the aim of stabilizing the learning process.

7.1 Results from training data set

When analyzing the three different approaches to construct an investor's portfolio, the litmus test is which of these alternatives yields the best performance, both in absolute and risk-adjusted terms. But first, we consider the performance of the three methods to construct a portfolio on the training data.

For MPT, we use the first 120 monthly observations of the asset prices, i.e., 10 years of data from 1969 to 1979, to construct the weights of the three asset classes. Thereafter, we do not update the distribution of asset prices but keep it and hence the efficient frontier constant. Therefore, the weights of the asset classes change only when the risk-free rate and thus the tangency point on the efficient frontier changes.

For TS, we start with the distribution of asset prices for the MPT but we use Thompson Sampling to update the posterior distribution with yearly frequency. Thus, the algorithm has the opportunity to learn on a yearly basis from the latest asset prices as well as contextual variables to update the posterior distribution and hence to adjust the weights of the three asset classes accordingly.

Lastly, the RL algorithm is actually trained on synthetic data sets generated by a vector auto-regressive (VAR) model that is calibrated on the first 10 years of the training data (identical to the data used for the MPT). Since the data generating process involves random sampling of data from the VAR model and as the neural network itself starts with random weights, the resulting portfolio and hence its performance varies slightly with each run. This is not the case for MPT and TS as the calculation of the mean and covariance matrix is based on historical data and a static updating rule. Therefore, there is no uncertainty in the portfolio construction of these two approaches. However, the neural network is trained only on the data of the first ten years and on the corresponding synthetic data. It is not retrained on later data, but it ingested the data from 1979 onward to produce weights of the asset classes. In Figure 20 we plot the performance of the three different approaches[34].

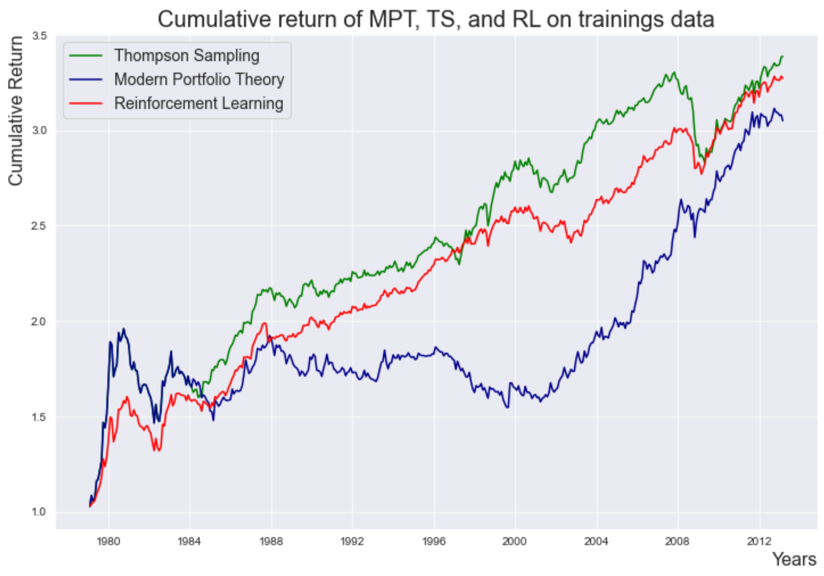


Figure 20: Performance of MPT, TS, and RL on the training data

Figure 20 shows the performance in terms of absolute return for the time

[34]For RL we plot a representative realization with average return of 5.20% p.a. and risk-return ratio of 0.20 (see table 1 for comparison).

period from January 1979 to February 2013. As stated earlier, note that we used the first 120 months of the data to estimate the distribution of asset prices. Hence, the analysis starts after 10 years and ends somewhat arbitrary in 2013 as we divided the entire data set in 80% training and 20% test data.

The performance of the "blue" MPT portfolio coincides with the "green" TS portfolio up until 1984. During that year, we see a difference in the performance of the MPT and the TS portfolio appearing. The reason for this is that it takes a while not only for the updating of the distribution during the TS algorithm to be measurable in the mean and covariance of the posterior distribution but also that these changes yield different portfolio weights during the process of determining the tangency point on the efficient frontier.

Note that we fixed the weight of the money market account at 25% in all portfolios to compare their performance. Adjusting the parameter of risk aversion (γ) so that the results between MPT and TS on the one hand and RL on the other are comparable turned out to be not only very difficult but unstable over time. For the RL algorithm, the weight of the money market account is found directly and not by a subsequent optimization procedure, as is the case for the MPT and TS approach. Hence, we decided that it is prudent to leave the risk-aversion parameter out of the simulation and rather fix the weight of the risk-free asset so that the choice and therefore the performance of the remaining risky assets are the decisive factors. Thus, instead of setting a value for gamma, the investor's risk aversion, and finding the corresponding portfolio on the CAPM-line, we set the weight of the money market account to 25% and choose gamma accordingly[35] This allows for the comparison of the results among all three approaches, as the weights of the risky assets are the driving force behind the performance of the resulting portfolio.

Surprisingly, the "blue" MPT portfolio shows the most volatile performance. It exhibits strong gains in early 1980 only to lose some of these gains again quickly and exhibits a phase of weak performance in the time from 1988 to 2004, when it starts to make up some of the lost ground to the other two portfolios. As we see in Figure 21, due to its small exposure to the S&P500, it is less affected by the Lehman crisis in 2008 and benefits more from the rally of the gold price as a result of that crisis.

[35]This is achieved by changing the weight of the money market account to 0.25 and distributing the difference of the previous value of the weight and 0.25 to the two weights of the risky assets evenly so that their ratio is unchanged.

The "green" TS portfolio outperforms the related MPT portfolio from 1995 to 2008. The reason for this, visible as well in Figure 21, is that the Thompson Sampling algorithm increases the equity weight by reducing the weight of gold. This leads to an outperformance over the MPT portfolio during 1988 to the beginning of the 2000s as the dot-com phase led to a strong rally in equity prices. However, with the Lehman-crisis in 2008, the outperformance over the "blue" MPT portfolio disappears again. Interestingly, though, the TS portfolio is less volatile than the MPT portfolio.

The absolute winner of the competition in risk-adjusted terms, at least on the training data (see also table 1), is the "red" RL portfolio, as it less volatile than TS. This is somewhat surprising, as the goal of the algorithm is to maximize the absolute return and the risk of the portfolio is not considered at all.

In Figure 21 we show the composition of the three resulting portfolios. Here we can see that the MPT portfolio invests only in money market and gold for the first twenty years. Only in 2002 the weight of S&P500 in blue becomes positive for the first time. This is the biggest difference to the TS portfolio that started investing in equities already in 1987, thereby separating itself from the MPT portfolio. Furthermore, it raises the weight for investments into the S&P500 of up to 60% in 2007.

In contrast to these two portfolios, the RL algorithm invests from the very beginning into equities. It is also noteworthy that in the RL the weights of the three asset classes change in a smaller magnitude than in the TS algorithm, which, for example, raised the weight of its equity investment from 1988 to 1989 by roughly 20%. The RL algorithm changes the weights at most by around 5% in an asset class from one year to the next.

The last results we want to show based on the training data, are the performance measures of the three different approaches in table 1, where we show ranges[36] for RL as the performance slightly changes from one run of the experiment to the next. As already observed from the performance graphs in Figure 20, we notice that the TS algorithm yields the best return in absolute terms, while RL yields the best risk-adjusted returns. The outperformance of RL can also be seen by the smallest value in the column "max. draw down", which shows the largest negative monthly performance.

As expected, the performance of MPT is lower than the performance of TS

[36]Note that we did not run enough experiments so that we could plot a distribution of these values as each run takes more than 20 minutes to complete.

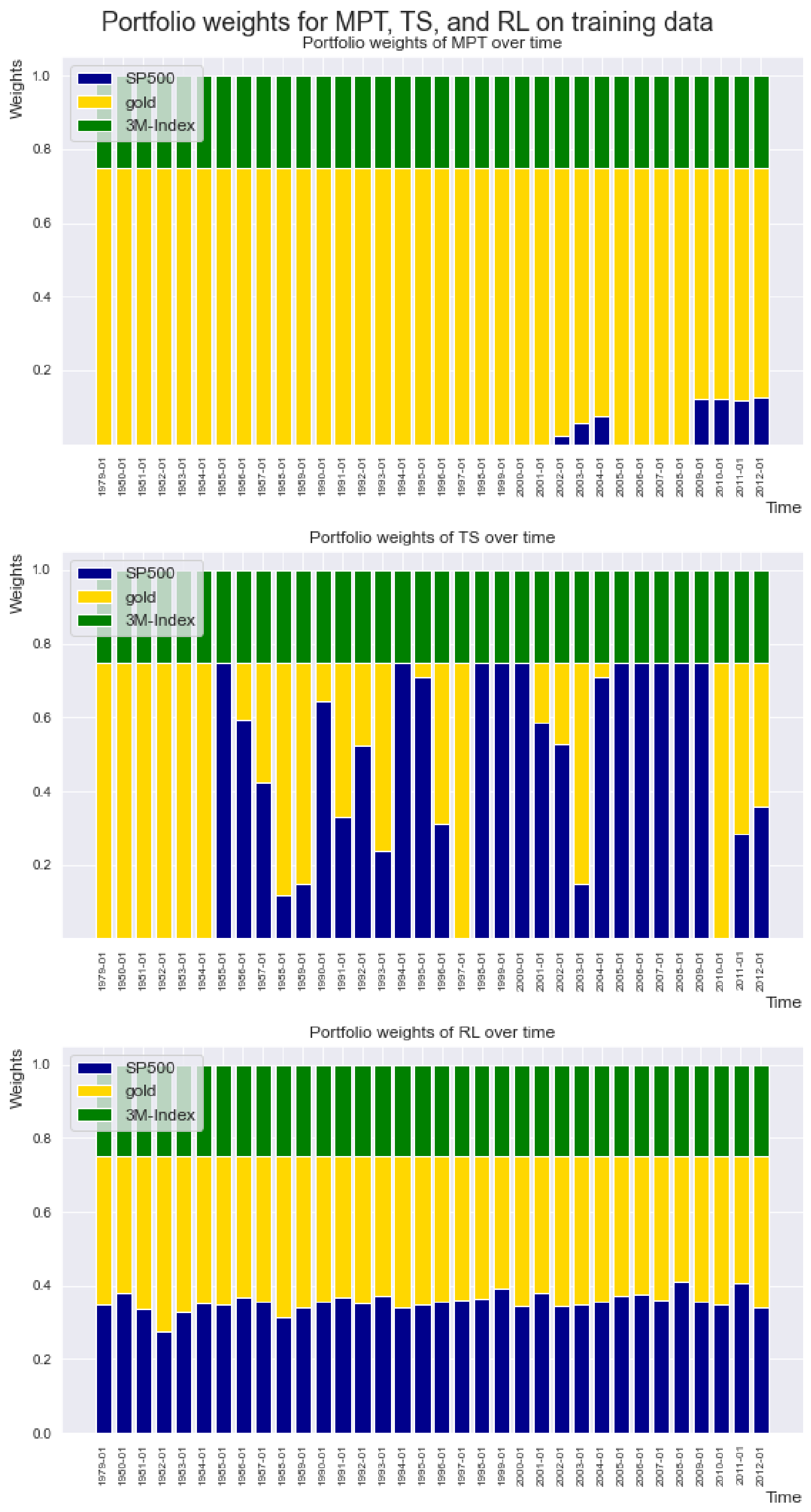


Figure 21: Portfolio composition of MPT, TS, and RL on the training data

Table 1: Performance measures for MPT, TS, and RL on training data

Algorithm	**Average Return in % p.a.**	**Max. draw down in %**	**Risk-return-ratio (average / std.)**
MPT	4.55	-16.62	0.12
TS	5.31	-16.62	0.15
RL	5.10 - 5.18	-12.68	0.18 - 0.21

and RL in absolute terms as well as in risk-adjusted terms. The fact that TS continuously incorporates new information about asset prices and adjusts their distributions (the posterior distributions) accordingly is a clear advantage. Note that this is not the case for the RL. We only trained RL on synthetic data generated from the first 10 years (120 observations) and did not retrain it later on. Hence, RL is quite successful in terms of relative return, but in the end an investor cares about the absolute return of her investment, and here the TS algorithm appears to be better.

7.2 Results from the test data set

Finally, the really important test is to compare the three approaches on the test data set. We separated 20% of the data at the beginning of the analysis to be able to run an unbiased test, as neither model has "seen" the data. The test data set covers the time from June 2013 to the end of the data in July 2024.

In contrast to previous experiments, we did not train the RL algorithm on synthetic data again[37], but we used the entire training data set. Since the training data is of reasonable size, we deemed it sufficient to train all the models on this data set before comparing their performances on the test data. This ensures that all three approaches have identical amounts of information at the beginning. Only the TS algorithm learns again with a yearly frequency about the asset prices and the contextual variables of the last 12 months and updates the posterior distribution accordingly. Hence, the TS algorithm has an information advantage over the other two approaches. The MPT algorithm is based on the historic mean and standard deviation of the asset prices from the whole training data set and the portfolio weights are only adjusted when changes in the risk-free rate changes

[37]Repeated testing showed that the usage of synthetic data did not yield more stable results but rather the opposite. The usage of synthetic data seems to amplify catastrophic forgetting (see below).

the tangency point on the efficient frontier. The RL algorithm learns every month the current asset prices and the values of the context and determines the portfolio weights based on that. However, the parameters of the neural network remain constant for the entire period as no retraining takes place.

Figure 22 shows the performance of the three approaches on the test data.

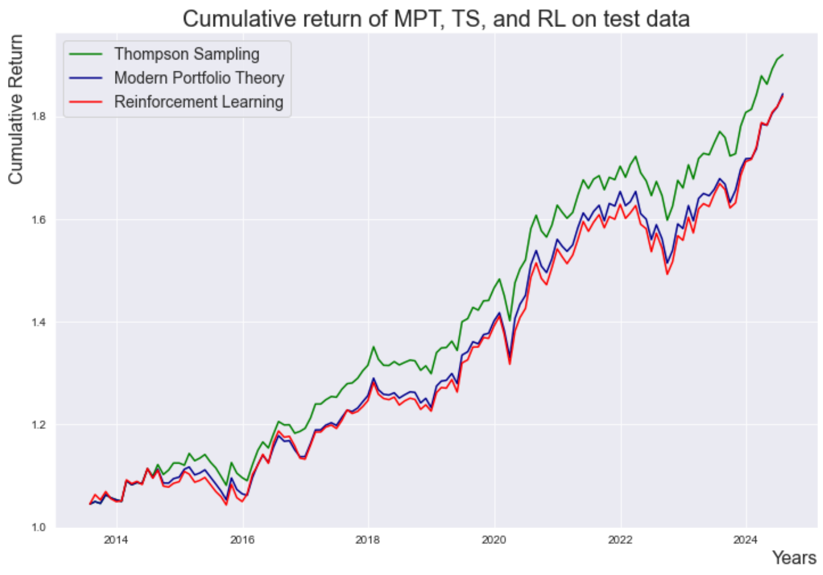


Figure 22: Performance of MPT, TS, and RL on the test data

In Figure 22 we can see that all three approaches show very attractive returns for an investor with a performance of between 6.57% and 7.50% per year. For this particular time period, the TS approach (green line) shows the best performance in absolute terms as well as in risk-adjusted terms. As the composition of the three portfolios shows, see Figure 31 in the appendix, this is due to the large weight of equities in the TS portfolio relative to the other portfolios. In addition, TS increases the weight of gold in 2021 and 2022 which performed very well during that period due to an increase in gold purchases from a number of central banks, who are rumored to reduce their dependency on the US-dollar. It is noteworthy that the TS approach, which is, as discussed earlier, to some extent a momentum strategy, increases the weight of gold so distinctively in 2021 (see the size of the yellow bars in Figure 31 in this year). Here, the information advantage of the TS approach becomes visible as gold performed very well in the years before 2021. Since the result of the RL varies with each run of the experiment, we again show ranges in table 2. Note that the variation in the test data increased in relation to the training data.

Furthermore, it should be noted that RL is outperformed by both approaches to construct an investment portfolio. This is also observable in the risk-return measures, which are shown in table 2.

Table 2: Performance measures for MPT, TS, and RL on the test data

Algorithm	**Average Return in % p.a.**	**Max. draw down in %**	**Risk-return-ratio (average / std.)**
MPT	7.19	-5.23	0.26
TS	7.88	-4.89	0.29
RL	6.57 - 7.50	-4.58 - 5.61	0.24 - 0.26

Table 2 displays TS as the overall winner with the highest annual performance and the best risk-adjusted return. MPT takes second place in terms of risk-adjusted return. The result of the RL performance is quite disappointing. It hardly beats the MPT when comparing the average annual return, despite having an information advantage, and clearly loses when comparing the risk-return metric. Furthermore, it also loses to TS, here even in both dimensions.

The outperformance of the TS approach should not be surprising, since it learns on a yearly basis about the latest asset prices and models the investor to update her posterior distribution of the asset prices accordingly. Although the RL-approach is not retrained during our experiment on the test data set, it gets the information about the context variables as well as the asset prices to calculate the weights for the next period. Hence, one could argue that it is in a better position to adjust for changes in the economic environment than the MPT-approach and that it should exhibit better return data than MPT.

Due of the drop in performance from training to test data, we conclude that the model has been overfitted to the training data (see Figure 17 for an illustration) and therefore the results of the training data probably overstated the performance of the RL algorithm.

Furthermore, we have noticed that the variability (i.e., the absolute range of the results) in the results of the RL-approach became even bigger on the test data. The fact that the results are not identical in each execution of the code is to be expected as the initial setting of the parameters is purely random and the results of a stochastic gradient descent are therefore stochastic as well. However, the results change more from one run to the next than can be explained by the inherent randomness of the neural network. Thus, we suspect that the learning itself is not stable.

As described earlier, we might encounter an often seen problem in online learning called *catastrophic forgetting.* Catastrophic forgetting refers to a scenario in which the optimal action in the current situation is completely reversed in the next situation. Hence, the algorithm "forgets" the conclusion from the first situation as it is overwritten by the optimal action in the subsequent situation. Note, however, that we are not in an online setting but we are applying Monte Carlo Control by considering averages of size "batchsize" when training the model. But still, the algorithm is not able to identify a unique minimum of the loss function or in more technical terms, the direction of the gradient descent, i.e., in which direction the weights should be adjusted, might flip from a given direction in one situation to almost opposite direction in the next while the parameters describing the state of the environment are still fairly similar. Thus, the learning process seems to be unstable. To test this hypothesis, we modify the RL-approach so that it should become more stable and, therefore, in a better position to learn from the context variables.

7.3 DQN, target network and experience reply

Since we suspect that the RL algorithm suffers from learning instability, we want to implement a modified Reinforcement Learning algorithm that uses two concepts that can potentially stabilize the learning process. Both concepts became popular after the publication of the article [Mni+15], in which data scientists from DeepMinds wrote an algorithm that could learn to play seven Atari games by itself using Reinforcement Learning. The approach is also known as Deep Q-Learning or DQN for short.

The first idea is *experience replay.* Here we store the main characteristics of the environment in a list[38] from which elements are randomly selected. The number of elements that are randomly chosen is again determined by the parameter "batchsize" during the training of the model. This way the order of the training examples is broken and the model is forced to learn the underlying relationship between the variables and the rewards. In our set-up we will store the current value of the state (the q-value), the values of the following state, the action currently chosen, and the reward following this action in the list.

[38]Here one often uses a particular data structure called *deque.* This list can store a certain number of elements simply by appending them. If more elements than its capacity are appended, the first elements are dropped from the list.

The second concept is called *target network*. Here, the idea is to separate the training of the neural network, which again is used to approximate the q-function, and the generation of the experience. To achieve this, we use two neural networks. The first one, called simply q-network or policy network, is used to generate q-values based on which the algorithm chooses the next action to take and which are stored in a list for experience replay. A second network, called target network, is used to approximate the value of moving to the next state. Put differently, q-values of the state s' are estimated by the target network while the q-values of state s are estimated using the policy network.

The difference between the two networks is how they are trained. The policy network is trained once a fixed number of epochs have passed. We use the same parameter "batchsize" for the training frequency. The target network is not trained at all, but it receives a copy of the parameters of the policy network once another fixed number of epochs has passed (we use the parameter "target_update_freq" to define this frequency). As the parameters of the target network are constant for a certain time and the target network provides the estimates of the q-values of the next states, the learning stabilizes as the estimates of the q-values for the next state are not influenced by the current experiences. To make this point clearer, we revisit equation (19) with α equal to one:

$$q(s_{t+1}, a) = q(s_t, a) + r_{t+1} + \delta \max_{a' \in \mathcal{A}} q(s'_{t+1}, a') - q(s_t, a). \tag{23}$$

If we manage to find the optimum of the q-function, denoted by q^*, then equation (23) becomes

$$q^*(s, a) = r_{t+1} + \delta \max_{a' \in \mathcal{A}} q^*(s', a'). \tag{24}$$

That is, the Bellman optimality condition implies that the value of being in state s is equal to the reward the agent receives when choosing action a plus the discounted value of the state s', in which the optimal action a' is being played. When applying the concept of a target network, the estimate of the q-value of state s' (right side of equation (24)) is calculated by the target network while the q-value of state s (left side of equation (24)) is calculated by the policy network. Since the loss-function takes as input the two sides of equation (24), we trying to minimize the difference (with the chosen loss function we minimize the squared difference), which is the same as adjusting the weights of the policy network so that equation (24) is getting closer to being satisfied. Since the q-value on the right side of the equation is approximated by the target network, only the left side of the equation is changed when the policy network is being optimized, which stabilizes the learning process. Figure 23, which is similar to Figure 3.15 from [ZB20], illustrates the relationships.

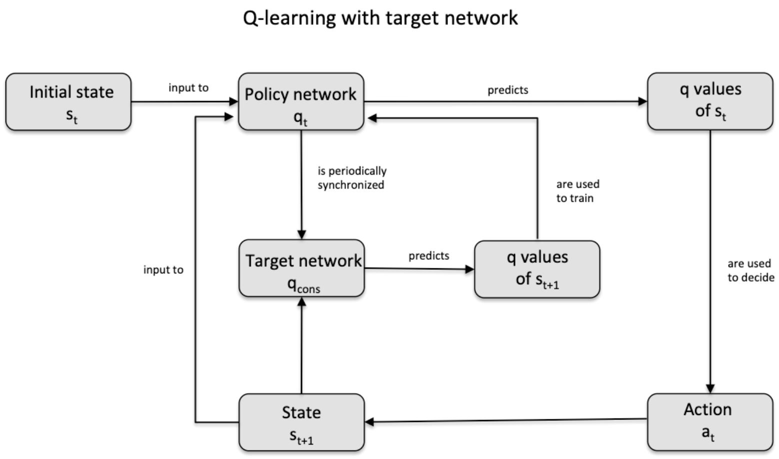


Figure 23: Visualization of the relation between policy and target network

Before we show the Python code to demonstrate how the two concepts are implemented, we want to highlight an additional change to the previous implementation of the Monte Carlo Control.

Following Mnih et al., we are using *epsilon-greedy* to steer exploration and exploitation. We start with a large value of epsilon to ensure exploration at the beginning of the experiment, and we reduce epsilon with each epoch until it reaches the value of $\epsilon = 0.01$ so that at later stage of the experiment the exploitation of the best strategies takes over.

The code snippet of listing 11 in the appendix shows how we draw a random number out of the interval from zero to one. If this number is smaller than epsilon, we draw a random vector (in Pytorch called torch) and use these weights for the three asset. This branch represents the exploration phase. If the random number is larger than epsilon, we are in the exploitation branch. Here, we receive the q-values from the policy network. These can be interpreted as probabilities, as the output layer of the neural network applies the softmax function. Hence, we receive three values that add up to one.

In listing 8, we show how to implement equation (24). The explanation of the code is as follows. We have randomly chosen a fixed number of complete descriptions of the environment from the deque data structure. The values characterizing the state of environment are stored in the list "state_batch". Using the policy network, we calculate the corresponding q-values. These are weighted with the respective probabilities of each action (”action_batch”) and aggregated to get one numerical value. This value is compared with the sum of the rewards and estimates of the q-values of the following states s' (”next_state_batch”). We determine the optimal strategies by applying the max(·)-function and calculate the respective q-value of the subsequent state s'. Note that these estimates are determined by the target network, not the policy network. This step is also exempt from the backpropagation step by using ”torch.no_grad”. That way, only the policy network is optimized when calling the train function while the parameters of the target network remain constant. This implementation serves the goal of learning stability. Although the target network is never explicitly trained, its estimation of the q-values of state s' should improve once its parameters are updated to the parameters of the policy network.

```
# Compute q-values with policy network
q_values = policy_net(state_batch)
prob = torch.cat(action_batch).view(batchsize,-1)
result = torch.sum(q_values * prob , dim=1).view(-1, 1).squeeze()
with torch.no_grad():
    max_next_q_values = target_net(next_state_batch).max(1)[0]
    target_q_values = reward_batch + delta * max_next_q_values
# train policy network
policy_net.train(result, target_q_values)
```

Listing 8: Implementation of experience replay

Finally, we show how to synchronize the target network with the policy network in listing 12 in the appendix. Every ”target_update_freq” epoch, we simply load the parameters of the policy network and store these as the new parameters of the target network.

7.4 Results of DQN on the test data set

Similarly to our previous analysis, we train the new RL algorithm, which we denote by RL+, on the entire training data set and evaluate its performance on the test data (no synthetic data is used). During the optimization phase on the training data we search for the optimal settings of the various hyperparameters. We find by increasing the number of nodes in the two hidden layers to 75 each, setting the discount factor δ equal to 0.9, and reducing the parameter ”batch_size” to 40, that the new algorithm RL+ shows the best results and easily beats the performance of TS and MPT on the training data set.

As before, the litmus test of the model is the test data, since this is new to the algorithm. Figure 24 reproduces the performance of MPT and TS on the test data and shows the cumulative return of RL+ in dark red.

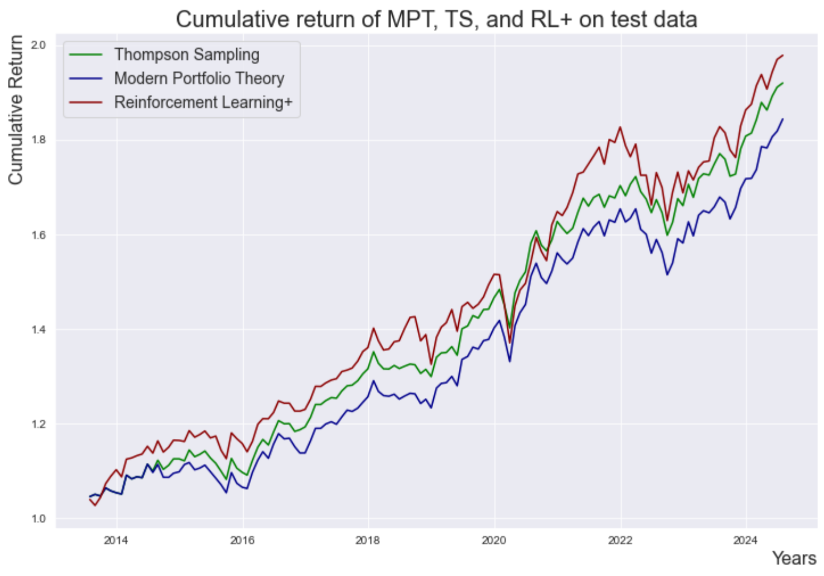


Figure 24: Performance of MPT, TS and Rl+ on the test data

Figure 24 shows a significant improvement of RL+ over RL in Figure 22. RL+ shows the highest absolute return almost over the entire time period. Only the COVID-19 selloff in early 2020 has a larger impact on RL+ than on the other two approaches, which is due to its bigger investment in equities, as we can see in Figure 25. For the remainder of the data set, the dark red line of RL+ is always above the lines representing MPT and TS.

Figure 25 shows the portfolio composition of RL+ on the test data set. Here we notice that the equity investment is larger compared to RL over the same period. At the beginning of 2022, the investment in gold is reduced to zero in favor of the investment into the S&P500. Note that the weight of the money market account is again fixed at 25%. Furthermore, we see that RL+ exhibits greater flexibility in moving funds in and out of the two risky assets compared to RL (compare the blue and yellow bars). In November 2013, for example, RL+ reduced the equity weight by approximately 10% from one period to the next and increased it by roughly the same magnitude in May 2014.

The improvement of RL+ becomes even more visible when considering table 3, which again reproduces the values for MPT and TS from table 2. Here we can

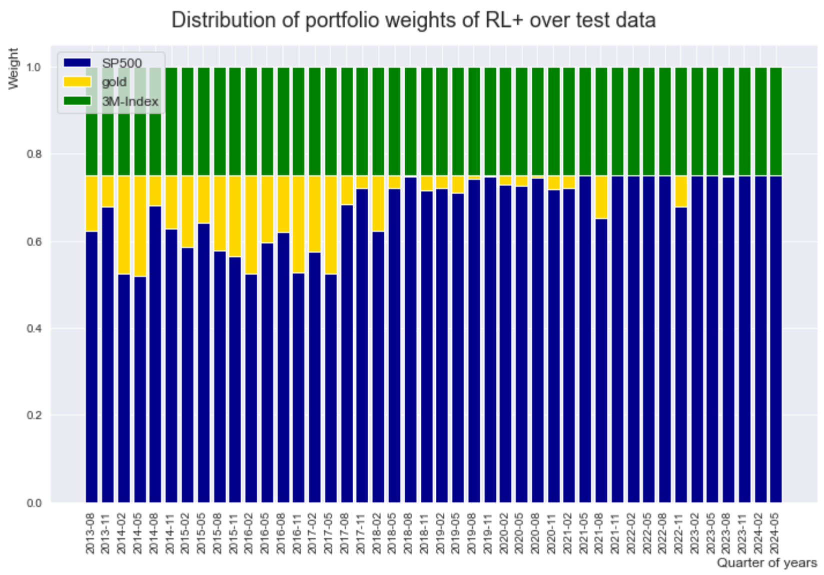


Figure 25: Performance of MPT, TS and Rl+ on the test data

clearly see that the RL+ shows significantly better performance than RL over the same time period. However, we also note that RL+ is more risky. Although the risk-adjusted return of RL+ is also higher than RL's, it cannot beat MPT or TS in that measure and the max_draw_down with -8.28% is significantly larger in absolute terms. The explanation for this is that RL+ puts significantly more weight on the asset class 'equity', which exhibits during this time period not only the highest average return but also the highest amount of risk measured by its standard deviation.

Note that in the way the q-function was defined, the algorithm aims to maximize the absolute return of the portfolio. In contrast to MPT and TS, the risk of the constructed portfolio does not enter the learning process. Hence, we can conclude with at least 1.31% higher absolute return compared to RL and outperforms MPT and TS, RL+ clearly achieves this goal.

Table 3: Performance measures for MPT, TS, and RL+ on test data

Algorithm	**Average Return in % p.a.**	**Max. draw down in %**	**Risk-return-ratio (average / std.)**
MPT	7.19	-5.23	0.26
TS	7.88	-4.89	0.29
RL+	8.81	-8.28	0.24

To receive more comfort that the RL+ approach is able to retrieve information from the context variables, we look at the correlations between the weights of the respective asset classes and the context variables. Here, we hope to see that the correlations point in directions that match our economic intuition.

The heatmap in Figure 26 indicates the correlations between the context variables and the weights of the asset classes. We show the context variable on the vertical and the asset prices on the horizontal axis.

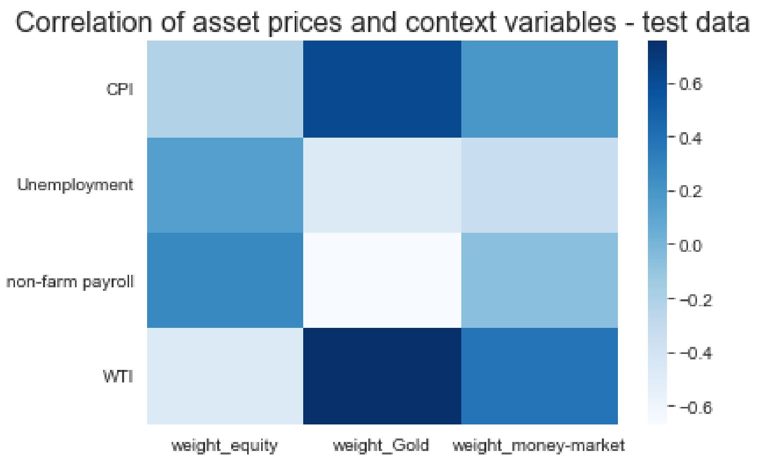


Figure 26: Correlations of asset prices and context variables on the test data

Figure 26 indicates the correlation using the darkness of the color in the respective area. That is, the darker the blue in the area of the intersection between context variable and the weight of the asset class the higher the correlation. The actual values are given in table 4 in the appendix.

Generally, the results are according to economic textbook. The algorithm tends to increase the weight of gold when the general price level goes up, that is when CPI increases or when the price of oil (WTI) goes up. So, the algorithm seems to learn that gold can serve as a hedge against inflationary pressure since it increases the weight of gold and decreases the weight of the equity investment. The same holds for an investment into the money market account, but to a smaller degree compared to gold. If the context variables CPI or WTI indicate inflationary pressure, the weight of the money market account tends to increase.

Similarly, when an increase in Non-Farm Payrolls indicates a stronger economy, the algorithm tends to reduce the weight of gold, as we see a strong negative correlation between gold and Non-Farm Payrolls, and to increase the weight of equities, since we find a strong positive correlation between the latter and Non-Farm Payrolls. So for all three asset classes we find indications that the algorithm adjusts the weights following our economic intuition.

To get an idea of how each variable impact the three investment weights, we consider *SHAP values*. SHAP stands for SHapley Additive exPlanations and is a method to explain the output of a black-box model by attributing the model's predictions to individual input features. The concept is named in honor of Lloyd Shapely, who introduced already in 1951 (see [Sha51]) an idea of how to evaluate the individual contribution of a player in a cooperative game (multiple player play together instead of against each other). Shapley's goal was to quantify how much each player contributes to the overall outcome of the game. This is achieved by measuring each player's contribution in each possible coalition with other players and then taking the average of the player's input. The contribution, and hence the SHAP values, are additive. That is, the sum of each players contribution results in the overall outcome.

In Machine Learning this concept is very useful, as many models are a black-box in terms of how and which feature is being used. Thus, SHAP values are one of the concepts in the developing field of *explainable AI* (XAI) that is used to measure and explain the impact of the various features on the output of a non-linear model. The model is interpreted as the game and the different features as the cooperative players. Therefore, SHAP values offer a solid method for measuring the impact of each feature on the prediction of the model. By checking the impact of the different features against our economic intuition, we can gain more comfort that the model is not only finding some spurious correlations between the features and the underlying goal but that the algorithm is really able to retrieve valuable information from the data that adds value to the optimization problem.

Listing 9 shows how to invoke the relevant Python package and how to generate SHAP values.

```python
import shap # import the respective package
# Calculate SHAP values
explainer = shap.DeepExplainer(model=policy_net, data=torch.
    from_numpy(data_training).float())
shap_values = explainer.shap_values(torch.from_numpy(
    data_training).float())
```

Listing 9: Generating SHAP values

The SHAP values can be plotted using the "shap.plots.waterfall" command. Figure 27 visualizes for a certain observation the impact each variable has on

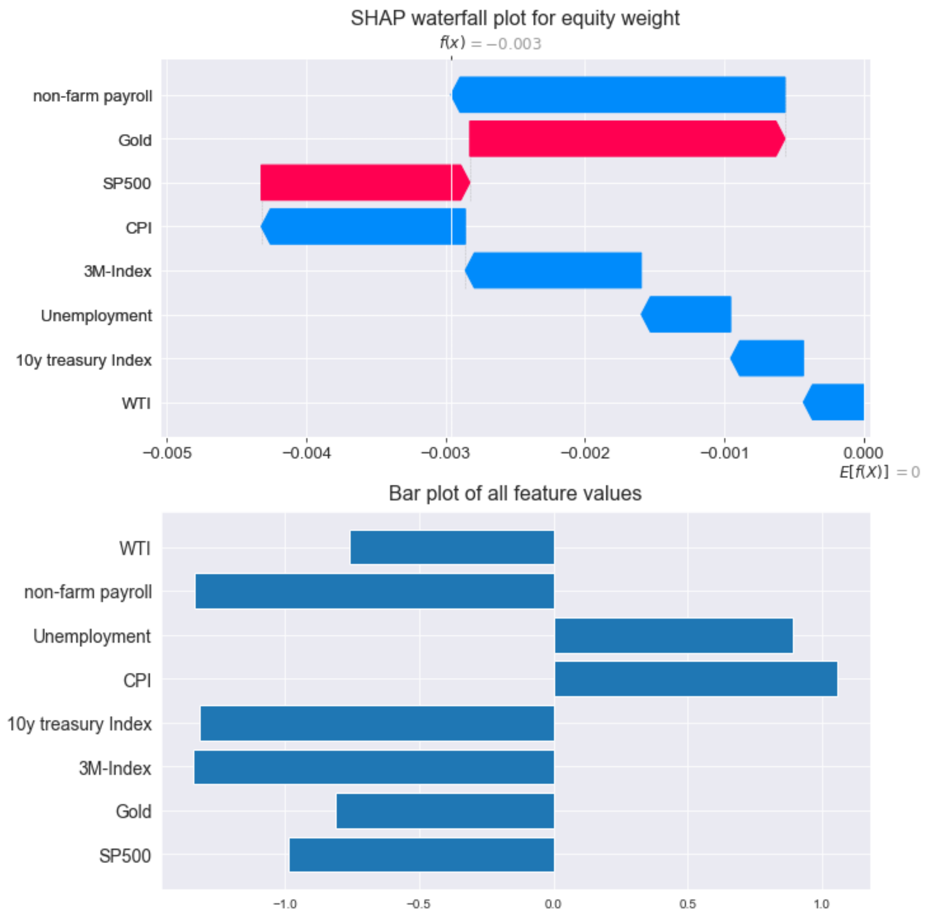


Figure 27: SHAP values offer insights on the impact of different features

the weight of the equity position. At the bottom of the graphic we start with the unconditional expected value of the equity weight. We normalize this value to zero to see the net change of all features on the weight, since the value itself is normalized thereby making it difficult to interpret. Below the waterfall plot we show a barplot with the normalized values of each feature for this particular observation. Here we are again only interested in the sign of the value to understand if the variable increased at a rate above the average or below in this month (remember that we are using the standard scaler and monthly data). With these two pieces of information, we can infer the impact of each feature on the equity weight for this month.

The starting point of the waterfall is the expected value of the equity weight at the bottom of the graph. The impact of each feature can be added to lead to the net change in the equity weight in this particular observation. The features are ordered by the absolute size of their impact. WTI causes the smallest absolute (negative as blue arrow points left) change, while Non-Farm Payroll has the largest absolute impact (but also negative).

A negative SHAP value of WTI means that the change in the price of oil in this observation, which was negative as well as we can see in bar chart below, slightly reduces the equity weight. The same is true for the next feature, the 10y treasury index. Unemployment also causes the equity weight to decrease (blue arrow pointing left). But in contrast to the two variables before, unemployment increased above average in this month, as we can see from the bar chart below. Hence, the algorithm changes the equity weight according to economic intuition: an increase in unemployment indicates slower economic growth, which makes an investment into the S&P500 less attractive. Also the reaction of the algorithm to reduction in the price of oil is according to economic textbook: a reduction in the price of WTI indicates a contraction of the economy which makes an investment in equity less attractive. This also causes a reduction of the equity weight.

Following this line of reasoning, we can check if the change in each variable in this particular month causes the algorithm to change the equity weight according to economic intuition, thus providing valuable insight into the model. We observe that the algorithm reacts in line with economic intuition to an above average fall of the Non-Farm Payroll number in that month by reducing the equity weight. Even the reduction in the price of the S&P itself follows textbook reasoning. An above average reduction in the S&P makes an investment in this asset class more attractive, as prices just went down and the algorithm increases the weight as expected. The opposite is true for gold (red arrow to the right). As the price of gold also decreased at an above-average rate in this month, an investment in gold becomes more attractive. Since we analyze the impact on the weight of the equity investment and gold is a substitute for that, the change in the gold price tends to decrease the equity weight (red arrow to the right)[39]. We show similar plots for the weights of gold and money market in the appendix.

Not only are the signs of the SHAP values of the features as we would have expected, we also notice that the largest impact on the equity weight has the Non-Farm-Payroll feature. This is a very interesting fact, as the Non-Farm Payroll number is for the financial markets one of the most anticipated data publications. As stated earlier, it is considered to be one of the best indicators of how the "real"

[39]In Figure 32 in the appendix we observe that the same decrease in the price of gold in this month has a positive impact on the weight of gold itself. With the decrease in the price of gold, an investment in gold becomes more attractive. Again, in line with our economic intuition.

economy is doing, and markets tend to react very strongly if the number differs from the consensus estimates. Interestingly, the algorithm gives the same result.

7.5 Investment with a finite horizon

The last check of the algorithm we want to perform is a test of how the model would invest if faced with a finite-horizon problem. That is, many private individuals invest their money with the goal of increasing their wealth only until a certain point in time. Motivation for such behavior is often the purchase of large-ticket items, such as a house or sending a child off to college. Thus, when this kind of large expenditure is due, the investment is withdrawn from the capital markets. This investment motivation also has implications for how the person invests. As the time of the large purchase comes closer, the investor is more likely to invest in safer assets (with smaller standard deviation), as there is less time to make up for a potential sell-off. The same behavior can be observed from professional investors when they are approaching retirement age. They tend to significantly reduce the risk of their portfolios as they shift their focus from the return of investment to preserving the value of their investment.

To test our algorithm in this regard, we only have to make a small adjustment. If the investment process is due to end at a certain time T in the future, then equation (24) changes to

$$q^*(s,a) = \begin{cases} r_{t+1} + \gamma \max_{a' \in \mathcal{A}} q^*(s',a') & \text{for } 0 \leq t < T \\ r_T & \text{for } t \geq T \end{cases} \tag{25}$$

The idea behind this modification is that there is no next state s' after time T to which the environment could move. Without making any other changes to the algorithm, we set $T = 120$ to model a time horizon of ten years as the investor's horizon. That is, we assume that the agent invests her capital for only ten years with the goal of achieving as much capital as possible to make a large-ticket purchase.

With these parameters, the RL+ algorithm returns the investment weights on the test data set as shown in Figure 28.

In Figure 28 we see that the magnitude of the equity investment compared to Figure 25 is much smaller to begin with (ca. 30% compared to 60% in Figure 25). Furthermore, the weight of the equity investment clearly decreases further as

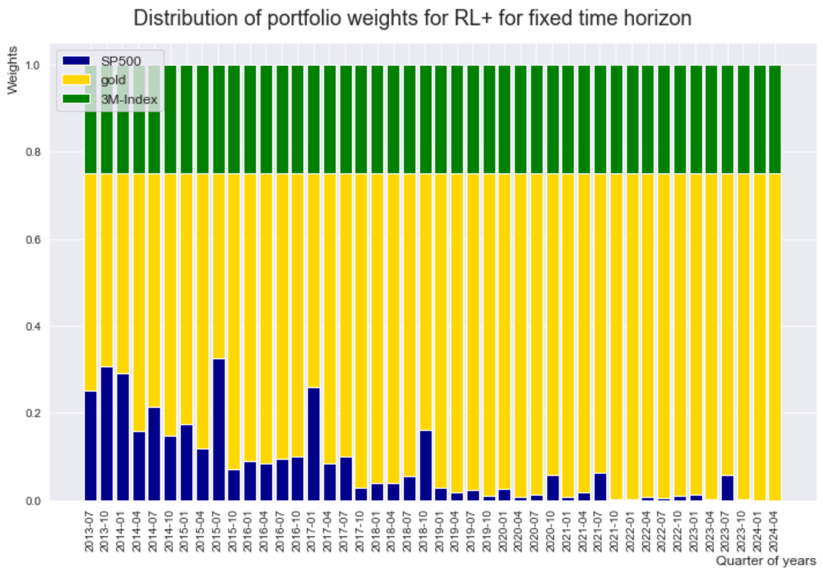


Figure 28: Portfolio composition for a finite time horizon

time passes[40]. Consequently, the weight of the less volatile investment gold goes up. Hence, the algorithm reduces the weight of the equity investment down to zero toward the end of the ten years, which is sensible considering that the chance of a sell-off in equities is bigger than in the other two asset classes (i.e., it has the highest standard deviation of all three investments). Thus, the algorithm's actions again match our expectations.

8 Summary

In the previous study we investigated whether we can use an Reinforcement Learning algorithm to successfully construct a portfolio of equities, gold, and a money market account. The performance of the resulting portfolio is compared against portfolios constructed by Modern Portfolio Theory and a variation thereof, which incorporates the latest financial data using Thompson Sampling. The RL algorithm is able to learn by itself from historical asset prices and contextual variables to build an investment portfolio that exhibits higher absolute returns than the two benchmark portfolios.

To have a benchmark against which we were able to compare the performance of the portfolio, we first developed the MPT-framework by deriving the

[40]Except for the situation in Feb. 2023 in which the chance of equity must have outweighed the risk. However, thereafter the equity weight drops to zero

efficient frontier, a set of portfolios with minimal risk for a given level of return. We found the CAPM-line through a combination of the risk-free rate and the tangency point on the efficient frontier. The CAPM-line represents the set of combinations between the risk-free rate and the tangency point on the efficient frontier, from which a risk-averse investor selects her utility-maximizing portfolio. Where exactly on the CAPM-line her particular optimal choice of that portfolio lies depends on her individual risk preferences.

Since the MPT-approach uses only historic prices, we applied Thompson Sampling to efficiently incorporate the latest data into the investor's decision process. In this methodology, the investor learns the latest prices of the assets as well as the values of contextual variables and updates her belief about future price developments, the so-called posterior distribution, accordingly. This update process can be shown to be an efficient incorporation of new information while still applying the utility-maximization approach of the MPT when finding the optimal portfolio weights.

As the third method of portfolio construction, we applied Reinforcement Learning to the investment problem. To derive the theoretical framework, we first introduced model-based learning using the game Gridworld as an example. In this game, a player's goal is to reach a certain tile on a board as quickly as possible, which earns her a certain amount of reward points. More generally, the goal of the player is to make moves that maximize her future rewards. This game served as an example to introduce value- and policy-functions to formulate the optimal behavior of the player. We demonstrated how this game can be solved using the policy iteration algorithm. Further, we showed how these concepts can be extended to an environment with infinite number of states to make them applicable to our investment problem. Using temporal difference and the Bellman optimality condition, we showed how to develop conditions for the so-called q-function, which described the solutions to our investment problem needed to satisfy.

Next, we introduced deep neural networks to approximate the q-function. Again, we first applied neural networks to Gridworld and demonstrated that a neural network is indeed capable of solving this game. Thereafter, we built a neural network to approximate the q-function of the investment problem by describing the environment and formulating the respective loss function to be minimized. The introduction of neural networks completed our toolbox of theoretical concepts needed to address the investment problem, and we moved on to the practical part of the thesis.

We used data from two different sources. We used Bloomberg as the source for price data of the S&P500, gold, 10-year US government bonds, and the 3-month US-Libor starting in January 1969. Four macroeconomic time series, which we used as context variables for the TS and RL algorithms, were taken from the FRED database. These variables describe the US consumer price data, the level of unemployment, the price of oil and the number of newly created jobs in the non-farm sector. In addition to the actual data, we developed a vector-autoregressive model that allowed us to generate synthetic data for the first 10 years. The goal was that the different time series of the synthetic data had the same statistical relationship among each other as the original data. We then used the synthetic data to repeatedly train the neural network.

Next, we used (only) historical asset prices to calculate the efficient frontier. Since the 10-year government bond index behaved too similar to the money market account, we dropped it as an asset class and added it to the context variable. Using the time series of the S&P500 and gold, we estimated the efficient frontier from which we could derive the CAPM-line for a given level of the risk-free money market rate. We fixed the weight of the money market investment to 25% to be able to compare the performance between the three different approaches of portfolio construction.

By fixing the weight of the money market account, we implicitly assumed the investor's parameter of risk aversion to take on that level that would fix the weight of the money market part exactly at 25% (without explicitly solving for the value). Given this weight of the money market, we knew where on the CAPM-line the optimal portfolio was to be found.

Subsequently, we used Thompson Sampling to incorporate new information such as incoming asset prices and context variables to calculate the portfolio weights. We expected that this second approach would determine portfolio weights that yield better performance than MPT, since it had the advantage of adjusting the portfolio weights to more recent asset price data.

The third, and for this study relevant approach to construct an investment portfolio, was based on a Reinforcement Learning algorithm. It incorporated the previously discussed concepts of q-learning via a neural network. The advantage of this approach was for one, the possibility to incorporate context variables and, second, to explicitly model the investment into the money market account as an alternative to the other two investment classes. In the MPT and TS approach, the safe asset class was assumed to be uncorrelated to all other assets, and its investment weight depended only on the risk-aversion of the investor. Hence, the RL algorithm modeled the safe asset in a more realistic way and we expected it to be able to discover additional information between the money market account and the other assets, as well as the contextual variables.

We used four different macroeconomic variables for which we found long and reliable time series: CPI, unemployment data, Non-Farm Payroll data and the price of a barrel of oil (WTI). The novelty of this approach was that if the algorithm was able to detect valuable information within these context variables, it should be able to adjust the portfolio weights in response to certain changes in these variables. Another important difference from the first two approaches was that the RL algorithm maximizes the absolute return of the portfolio without taking its risk into account, measured as the standard deviation. In contrast, the MPT and TS portfolio construction explicitly tried to maximize profit for a given level of risk, or, equivalently, tried to minimize risk for a given level of profit.

We used the Pytorch framework to implement the so-called Monte Carlo Control algorithm first. For the convergence criteria, we implemented an early stopping class that monitored the loss function of the model and halted the algorithm if no improvement was detected for a predetermined number of iterations.

The first important finding of this study was that we discovered indications that the algorithm was indeed able to detect valuable information in the context variables and was able to construct an investment portfolio whose performance and risk-return profile was similar to the two benchmark approaches MPT and TS. The RL portfolio showed better absolute return figures than the MPT portfolio and was able to beat both portfolios in the risk-return measure on the training data. But we also noticed a variability in the results when repeating the experiment and we were surprised by the good risk-return ratios, as the RL method did not consider risk in the portfolio construction process.

However, the real test was the test data set. We trained the RL algorithm on the entire training data and refrained from using synthetic data to stabilize the results. We saw an underperformance of the RL approach versus the other two portfolio construction methods in terms of absolute as well as on risk-adjusted return. Only in the measure of the maximum draw-down, i.e., the largest negative return in any period during the observed timer period, was the RL algorithm able to beat the other approaches. The drop in performance from training to test data led us to the conclusion that the algorithm was overfitted to the training data and that the results for this data set were overstated. In addition, we noticed that the learning process was not robust.

In order to make the learning process more stable, we significantly adapted the model by introducing experience replay and dividing the neural network into a target and a policy network. The experience replay idea was implemented by storing a fixed number of complete states of the environment in a specific data structure. Out of this structure we randomly drew a number of states given by the parameter "batch_size". Next, we trained the algorithm only on this random set of states. This new approach had two effects. First, since the training was done on multiple (i.e., "batchsize" many) states of the environment, parts of the noise in these inputs were averaged out. Secondly, since the model was trained on a random set of states, the sequential order of the input data was broken, and the algorithm was forced to learn the underlying relationship of the input data and the resulting rewards.

These modifications led to an improved algorithm we called RL+. The portfolio constructed by RL+ showed a significantly higher absolute return than the two portfolios constructed by the two benchmark approaches. Due to the larger exposure to the asset class 'equity', the RL+ portfolio was inherently more risky, which was observable in a higher maximal draw down and lower risk-adjusted return. However, as we formulated the problem so that the reward of the investor was the absolute return rather than the a risk-return-ratio, we concluded that the algorithm accomplished the formulated goal.

To get more confirmation that the algorithm was able to retrieve valuable information from the data and successfully incorporate these into the portfolio construction process, we analyzed the correlations between the portfolio weights and the context variables. The results showed nicely that the algorithm was correctly increasing the weights of gold and, to a lesser extent, the money market account if the consumer price index (CPI) or the price of oil (WTI) went up. Hence, one could argue that the algorithm correctly identified gold as a hedge against inflationary pressure. Similarly, the algorithm increased the weight of the equity investment if there were signs that the economic activity was picking up indicated by a rising figure of the Non-Farm Payrolls.

To improve the understanding of how the model learned and how the context variables influenced the portfolio weights, we calculated the corresponding SHAP values. SHAP values are used in explainable AI (XAI) to enhance the interpretability of black-box models, such as neural networks, by quantifying the average marginal contribution of each feature. When we analyzed the SHAP values, we saw that the model changed the weights of the asset classes according to our economic intuition. If, for example, in one month the prices of equity went down, the algorithm increased the weight of the equity part of the portfolio (all else equal) since this investment became more attractive. A reduction in the price of oil or the unemployment rate and an increase in the Non-Farm Payroll figure had the same impact on the equity weight. If the price of gold decreased in a given month, then the algorithm reacted by decreasing the equity weight (all else equal) since the investment in gold, a substitute for equity, became more attractive. The analysis of the SHAP values revealed that not only was the qualitative impact of the variables showing the expected direction, but also that the change of the Non-Farm Payrolls had the largest impact of the context variables on the portfolio weights. This result matched our expectations as the Non-Farm Payroll is the indicator of changes in the economic activity in the USA with the shortest time lag. The publication of the figure on the first Friday of the month is one of the closest followed events by market participants, and capital markets worldwide react strongly if it changes differently to the consensus forecast.

Lastly, we investigated how the portfolio composition changed over time if we modified the challenge from an unconstrained maximization of the investment return to a maximization of the return until a given point in time in the future. As we would have expected from a human portfolio manager, the algorithm continuously reduced the weight of the most risky asset class, 'equity', as the cut-off date approached.

In summary, the reinforcement algorithm produced results that not only matched our economic intuition, it based its decision on the historically most relevant macroeconomic variables and was able to construct an investment portfolio with annual return figures that were not only able to compete with the figures from the benchmark approaches, but exceeded them in some metrics (absolute return). Thus, one can conclude that Reinforcement Learning can be successfully used in the investment process.

The study could be expanded in a number of ways. First, the logical extension would be to not only consider a larger number of asset classes to invest in but also feed the algorithm a larger number of context variables and observe if the results improve further.

Second, it would be interesting to include a penalty for risk of the resulting portfolio in the reward function. Since the portfolio of the RL+ algorithm is inferior to the MPT and TS portfolio in terms of risk-adjusted return, this modification could be a way to improve the results in that dimension. A factor in the reward function to balance the positive reward of a higher return and a negative reward of higher risk could potentially be used to steer this trade-off.

Third, it would be interesting to extend the neural network to a long-short-term memory (LSTM) or a gated recurrent unit (GRU) as both these models allow us to identify long-term relationships among features. In financial markets, we often find correlations among variables of different time lags, which these models could in theory be able to pick up. However, we could no longer use experience replay to stabilize the learning process as here we purposely distort the order of inputs to force the model to learn the underling relationship.

Finally, there are multiple ways to potentially improve the learning process further. In an actor-critic framework, for example, two separate neural networks are used to stabilize the learning process. The actor network predicts the best action to take, whereas the critic estimates the value of getting into the new state when playing the proposed action. Hence, the critic network's output is used to reinforce which action the actor network proposes. Another alternative on how to modify the learning process is to move away from a single agent being modeled to multiple agents whose actions can influence each other. Hence, if one agent learns something new, the other agents can also benefit from that, and the learning process may become faster and more robust.

References

[Bel52] Richard Bellman. “On the theory of dynamic programming”. In: *Proceedings of the national Academy of Sciences* 38.8 (1952), pp. 716–719.

[BO94] Steve Beveridge and Cyril Oickle. “A Comparison of Box—Jenkins and objective methods for determining the order of a non-seasonal ARMA Model”. In: *Journal of Forecasting* 13.5 (1994), pp. 419–434.

[Bru24] Markus K. Brunnermeier. *Lecture notes Princeton University.* 2024. URL: `https://swh.princeton.edu/~markus/teaching/Fin501/07Lecture.pdf` (visited on 06/29/2024).

[Bue+19] Hans Buehler, Lukas Gonon, Josef Teichmann, and Ben Wood. “Deep hedging”. In: *Quantitative Finance* 19.8 (2019), pp. 1271–1291.

[Dev22] TensorFlow Developers. “TensorFlow”. In: *Zenodo* (2022).

[DF79] David A Dickey and Wayne A Fuller. “Distribution of the estimators for autoregressive time series with a unit root”. In: *Journal of the American statistical association* 74.366a (1979), pp. 427–431.

[DT20] Xin Du and Kumiko Tanaka-Ishii. “Stock embeddings acquired from news articles and price history, and an application to portfolio optimization”. In: *Proceedings of the 58th annual meeting of the association for computational linguistics.* 2020, pp. 3353–3363.

[Du+20] Jiayi Du, Muyang Jin, Petter N Kolm, Gordon Ritter, Yixuan Wang, and Bofei Zhang. “Deep reinforcement learning for option replication and hedging”. In: *The Journal of Financial Data Science* 2.4 (2020), pp. 44–57.

[Eng82] Robert F Engle. “Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation”. In: *Econometrica: Journal of the econometric society* (1982), pp. 987–1007.

[Fam17] Eugene F. Fama. *Efficient Capital Markets A Review of Theory and Empirical Work.* Ed. by John H. Cochrane and Tobias J. Moskowitz. The Fama Portfolio: Selected Papers of Eugene F. Fama. University of Chicago Press, 2017.

[Gér22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* ” O’Reilly Media, Inc.”, 2022.

[IPK21] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. “PyTorch”. In: *Programming with TensorFlow: solution for edge computing applications* (2021), pp. 87–104.

[Lab] U.S. Bureau of Labor Statistics. *U.S. Bureau of Labor Statistics.* URL: `https://www.bls.gov/cpi/` (visited on 01/27/2025).

[Lim+21] Francisco Caio Lima Paiva, Leonardo Kanashiro Felizardo, Reinaldo Augusto da Costa Bianchi, and Anna Helena Reali Costa. “Intelligent trading systems: A sentiment-aware reinforcement learning approach”. In: *Proceedings of the second ACM international conference on AI in finance.* 2021, pp. 1–9.

[Lop16] Marcos Lopez de Prado. “Building diversified portfolios that outperform out-of-sample”. In: *Journal of Portfolio Management* (2016).

[Mar52] Harry M. Markowitz. “Portfolio Selection”. In: *Journal of Finance* 7.1 (1952), pp. 77–91.

[mat25] mathworks. *Überanpassung.* 2025. URL: `https://de.mathworks.com/discovery/overfitting.html` (visited on 03/30/2025).

[Mni+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[MP21] Diego Mendez-Carbajo and Genevieve M Podleski. “Federal Reserve Economic Data: A History”. In: *The American Economist* 66.1 (2021), pp. 61–73.

[MP43] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[Pas+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic differentiation in pytorch”. In: *working paper* (2017).

[Pet21] Todd E Petzel. *Modern portfolio management: moving beyond modern portfolio theory.* John Wiley & Sons, 2021.

[Rao20] Ashwin Rao. *Understanding Risk-Aversion through Utility Theory.* 2020. URL: https://web.stanford.edu/class/cme241/lecture_slides/UtilityTheoryForRisk.pdf (visited on 07/06/2024).

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[Rit24] Ritvikmath. *Example Thompson sampling.* 2024. URL: https://github.com/ritvikmath/YouTubeVideoCode/blob/main/Thompson%20Sampling.ipynb (visited on 08/03/2024).

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[Sha51] Lloyd S Shapley. *Notes on the N-person Game.* Rand Corporation, 1951.

[Sha64] William F Sharpe. "Capital asset prices: A theory of market equilibrium under conditions of risk". In: *The journal of finance* 19.3 (1964), pp. 425–442.

[Sti03] Joseph E Stiglitz. *Economics for an imperfect world: Essays in honor of Joseph E. Stiglitz.* MIT Press, 2003.

[Str00] Malcolm Strens. "A Bayesian framework for reinforcement learning". In: *ICML*. Vol. 2000. 2000, pp. 943–950.

[TE21] Thibaut Théate and Damien Ernst. "An application of deep reinforcement learning to algorithmic trading". In: *Expert Systems with Applications* 173 (2021), p. 114632.

[Tho33] William R Thompson. "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples". In: *Biometrika* 25.3-4 (1933), pp. 285–294.

[Wan+19] Jingyuan Wang, Yang Zhang, Ke Tang, Junjie Wu, and Zhang Xiong. "Alphastock: A buying-winners-and-selling-losers investment strategy using interpretable deep reinforcement attention networks". In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 1900–1908.

[Wan+21] Zhicheng Wang, Biwei Huang, Shikui Tu, Kun Zhang, and Lei Xu. "DeepTrader: a deep reinforcement learning approach for risk-return balanced portfolio management with market conditions Embedding". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 35. 1. 2021, pp. 643–650.

[Wika] Wikipedia. *Conjugate Distributions*. URL: `https://en.wikipedia.org/wiki/Conjugate_prior` (visited on 08/03/2024).

[Wikb] Wikipedia. *Riskaversion*. URL: `https://de.wikipedia.org/wiki/Risikoaversion` (visited on 08/03/2024).

[Win25a] Johannes Winkler. *Efficient frontier*. 2025. URL: `https://github.com/JohannesWinkler74/MSc_Thesis/tree/main/efficient_frontier` (visited on 03/30/2025).

[Win25b] Johannes Winkler. *Generate data*. 2025. URL: `https://github.com/JohannesWinkler74/MSc_Thesis/tree/main/generate_data` (visited on 03/30/2025).

[Win25c] Johannes Winkler. *Gridworld Bellmann equation*. 2025. URL: `https://github.com/JohannesWinkler74/https://github.com/JohannesWinkler74/MSc_Thesis/tree/main/Gridworld` (visited on 03/30/2025).

[Win25d] Johannes Winkler. *Monte Carlo Control*. 2025. URL: `https://github.com/JohannesWinkler74/MSc_Thesis/tree/main/Monte%20Carlo%20Control` (visited on 03/30/2025).

[Win25e] Johannes Winkler. *Thompson Sampling*. 2025. URL: `https://github.com/JohannesWinkler74/MSc_Thesis/tree/main/Thompson%20Sampling` (visited on 03/30/2025).

[ZB20] Alexander Zai and Brandon Brown. *Deep reinforcement learning in action*. Manning Publications, 2020.

[ZZR19] Zihao Zhang, Stefan Zohren, and Stephen Roberts. "Deeplob: Deep convolutional neural networks for limit order books". In: *IEEE Transactions on Signal Processing* 67.11 (2019), pp. 3001–3012.

```
# each weight needs to be between 0 and 1
bounds = tuple((0, 1) for w in equal_weights)

# two constraints
constraints = (
  {'type': 'eq', 'fun': lambda x:  target - portfolio_returns(x,
   mu) }, # portfolio return should equal return_target
  {'type': 'eq', 'fun': lambda x: np.sum(x) - 1} )# weights must
   sum to one

# Initialize an array of target returns
return_target = np.linspace(start = 0.00, stop = 0.1, num = 100)
# empty list to store the results
std = []
# For loop to minimize objective function
for target in return_target:
  opt_result = sci.minimize(
    # Objective function
    fun = portfolio_sd,
    # start with equal weights
    x0 = equal_weights,
    args = sigma, # covariance_matrix as argument
    method = 'SLSQP',
    bounds = bounds,
    constraints = constraints )
```

Listing 10: Calculation of minimal risk portfolio

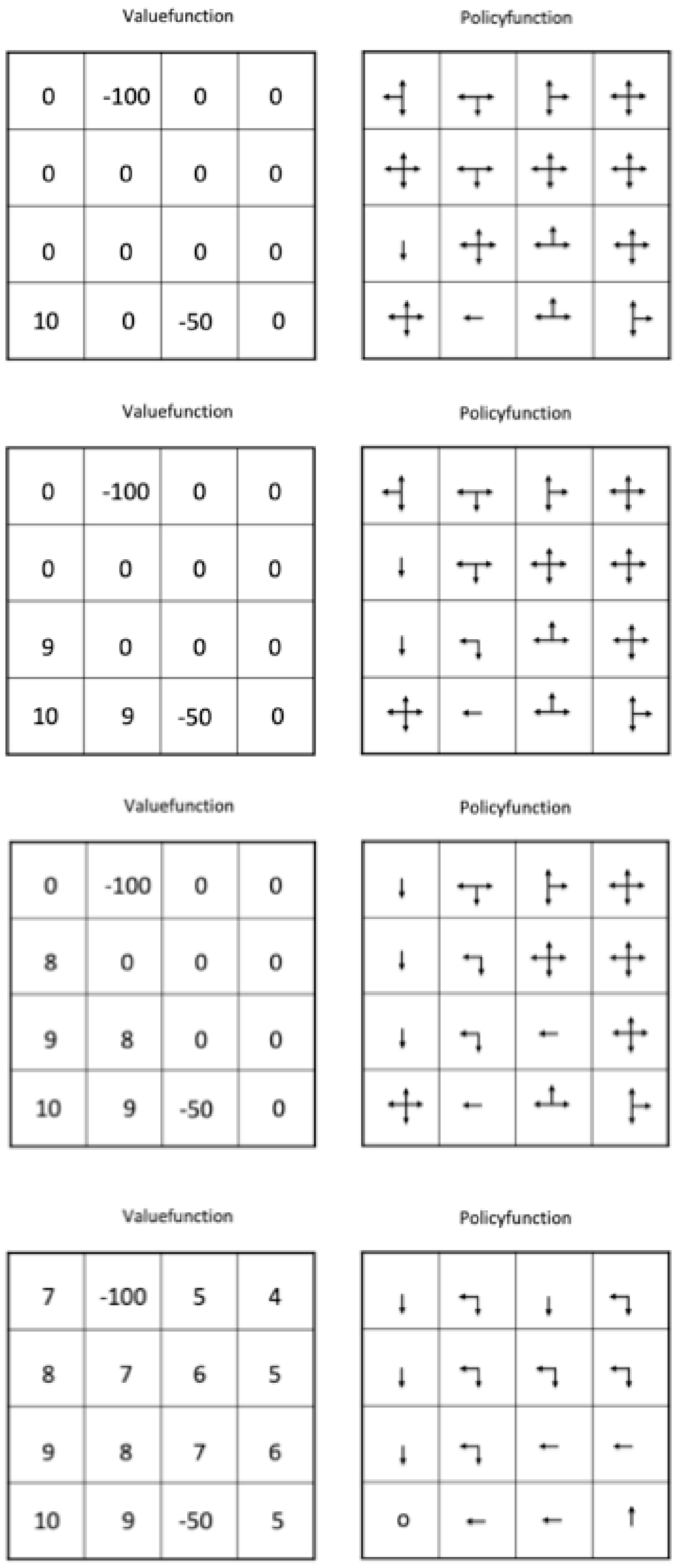
```
# epsilon-greedy
if random.random() < epsilon: # Explore
    prob = torch.rand(3, requires_grad=False)
    prob /= torch.sum(prob) # normalize to sum to 1
    action = prob

else: # Exploit
    q_values_policy = policy_net.forward(torch.from_numpy(state).
   float())
    action = q_values_policy
```

Listing 11: Implementation of epsilon-greedy

```python
# Update target network periodically
if steps % target_update_freq == 0:
    target_net.load_state_dict(policy_net.state_dict())
```

Listing 12: Synchronize the target network



0	-100	0	0
0	0	0	0
0	0	0	0
10	0	-50	0

0	-100	0	0
0	0	0	0
9	0	0	0
10	9	-50	0

0	-100	0	0
8	0	0	0
9	8	0	0
10	9	-50	0

7	-100	5	4
8	7	6	5
9	8	7	6
10	9	-50	5

Figure 29: Various stages of the iteration process between value and policy function

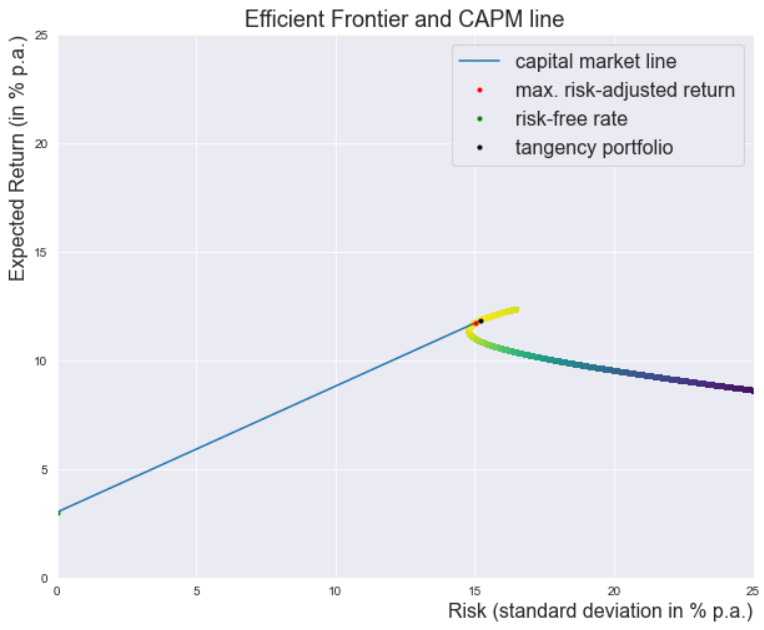


Figure 30: The CAPM line from the risk-free rate to the tangency portfolio

Table 4: Correlations of asset prices and context variables on test data

	weight equity	**weight gold**	**weight mm**
CPI	-0.223	0.617	0.191
Unemployment	0.140	-0.478	-0.335
Non-farm payroll	0.266	-0.668	-0.066
WTI	-0.476	0.751	0.378

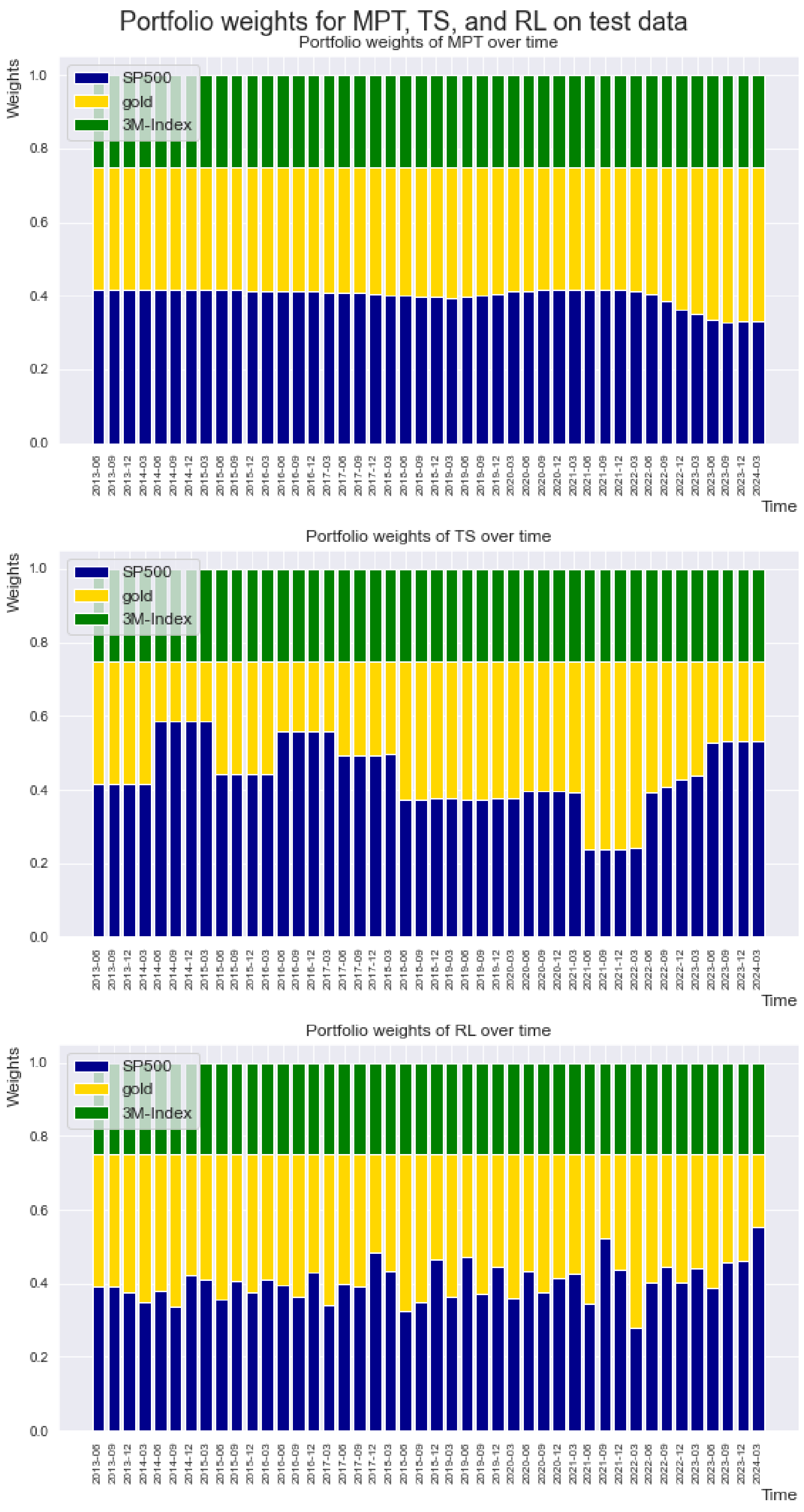


Figure 31: Portfolio composition of MPT, TS and Rl on the test data

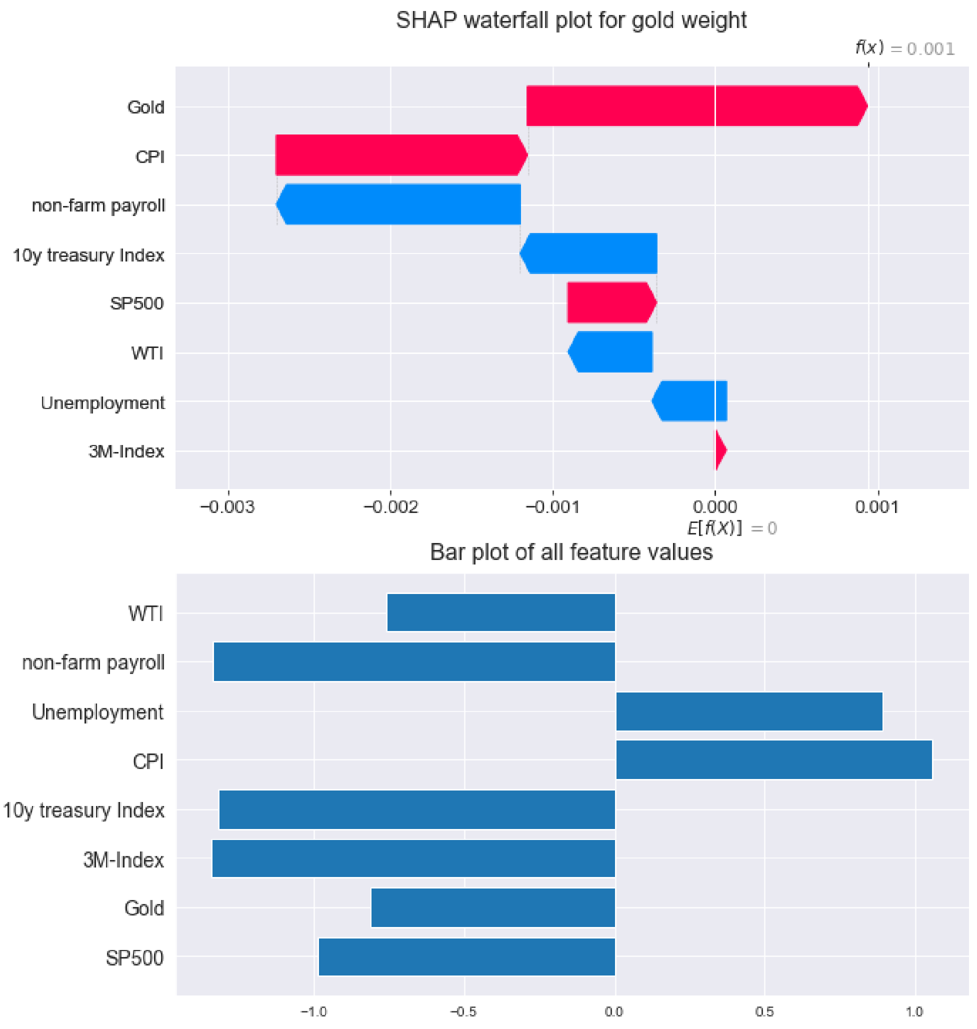


Figure 32: SHAP values for weight of gold investment

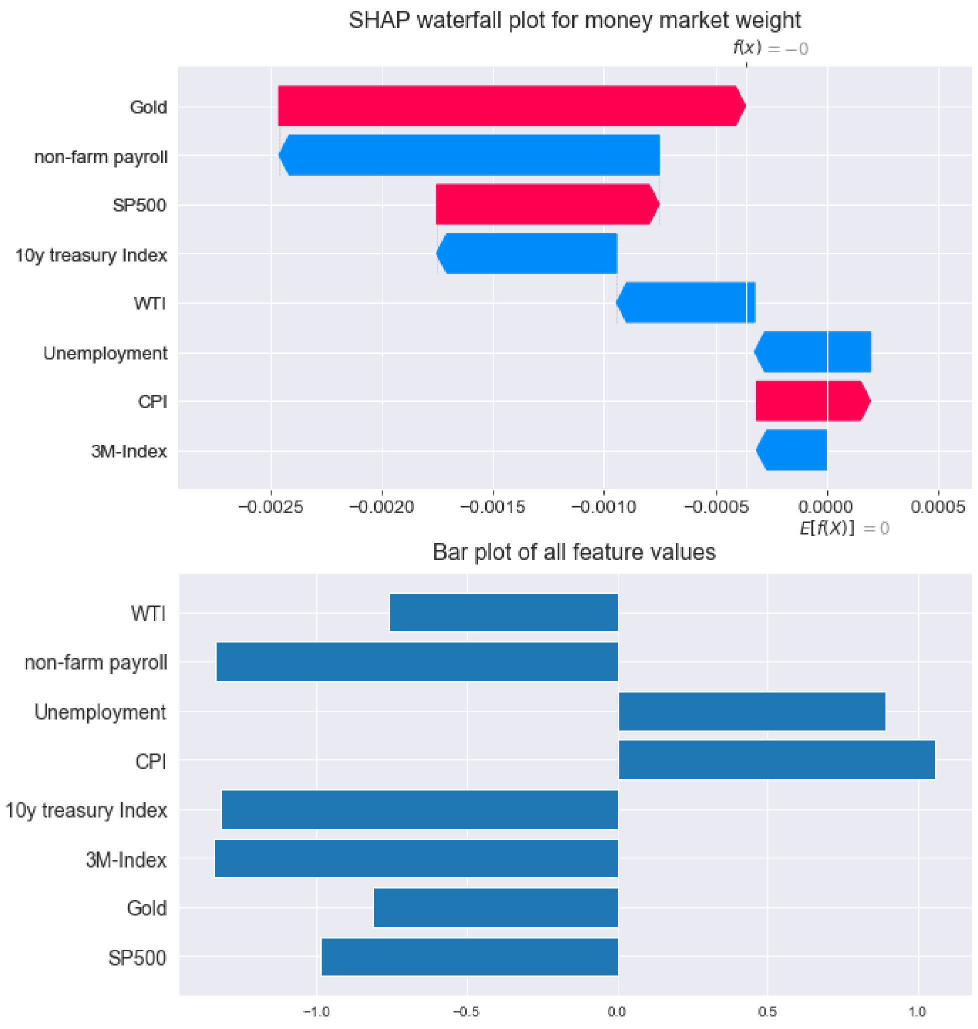


Figure 33: SHAP values for money market investment