POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Deep Reinforcement Learning for Portfolio Optimization

Supervisors

Prof. Luca CAGLIERO

Prof. Jacopo FIOR

Candidate

GIOELE SCALETTA

April 2024

Abstract

Portfolio management involves asset selection, allocation, and monitoring and aims to maximize returns while mitigating risks, considering the specific financial goals, risk appetite, and investing time horizon preferences of each individual or institution. Indeed, the risk-return trade-off, central to portfolio optimization, hinges on investor's preferences. Risk includes systematic and unsystematic risks, with diversification mitigating the latter. In imperfect markets, investors hope to exploit inefficiencies such as information asymmetries, frictions and their own psychological biases. Therefore, crafting custom and dynamic portfolio management strategies based on future prices predictions is a major challenge.

This thesis focuses on automating portfolio management using Deep Reinforcement Learning (DRL) which is based on an agent's interaction with the environment and optimizes decisions through feedback. DRL's suitability lies in its ability to directly output investment actions without having the unrealistic presumption of predicting future prices thus partly overcoming the complexity associated with modelling the market underlying functioning. The base DRL algorithm used is Proximal Policy Optimization (PPO) which improves previous constrained policy gradient optimization algorithms, retaining stability with a simpler implementation and better sample efficiency.

The additional contributions of this project are three implementation variants exploiting sequence models architectures and imitation learning to address the limitations of DRL applied to portfolio optimization. The Gated Transformer-XL (GTrXL) model is a transformer-based architecture designed to process long horizons of sequential information in RL and it was used in the first variant to improve the embedding of past prices and indicators. In the second variant, by predicting in hindsight the optimal actions, an expert actions dataset was created, and this allowed to pre-train PPO with the expert actions to improve data efficiency of limited financial data. Moreover, the expert actions were used in a third variant, which employs TD3+BC, an offline reinforcement learning model introducing Behavioural Cloning into Twin Delayed deep deterministic policy gradient. Furthermore, to diversify more and reduce bias and overfitting, this last variant also enlarges the training set using more stocks than the ones used for trading.

Overall, plain and volatility-adjusted returns results were positive with the third variant always overperforming the baseline in backtesting. However, stability was still not satisfying leading to the conclusion that, despite the model making effective decisions that persistently beat the market, technical indicators are not informative enough for the model to show stable and consistent decision-making.

Index

1 Introduction

In the financial domain, managing assets against uncertainty while trying to exploit market inefficiencies is a task of crucial importance and with big implications for both individual and institutional investors. Portfolio management consists of selecting, allocating, and monitoring a set of assets to maximize return while mitigating risks. It is a complete strategy that extends beyond mere investment choices by constantly redistributing wealth among different assets. Moreover, this process must be aligned with the individual's or institution's financial goals, risk tolerance, and investment time horizon.

The risk-return trade-off is one of the pillar concepts behind portfolio optimization and it is one of the most determinant factors when shaping a portfolio. As a matter of fact, more risk implies more return, and the balance between the two must be decided based on the investor's own preferences and goals. Risk in the investment domain can be quantitatively identified as volatility that indicates how much and how quickly the asset value fluctuates over time. Indeed, the more the assets value is unstable, the more uncertainty there is about the future value, the more insurance is needed, and therefore the more risk is involved in the investment.

Total risk is usually considered as incorporating two different risks: the systematic risk and the firm-specific or unsystematic risk. Where the first one is common to the whole market and depends on macro events and the second is the risk related to events affecting only a single firm. The unsystematic risk can be reduced or even eliminated by using diversification. As a matter of fact, by spreading the investment across different firms, sectors, industries and geographical regions, the specific risks are eliminated and only the common risk remains. Losses due to risks specific to a sector are in this way offset by gains in another one.

In a perfect market, where all investors take decisions rationally based on the same set of information, arbitrage, and therefore gaining an advantage with respect to other investors, would become almost impossible. However, information asymmetries and frictions such as transaction costs together with other factors allow some investors to exploit market inefficiencies and to make a profit out of it.

Concerning this, it is relevant to mention behavioural finance and the huge impact that psychology has on financial decision-making. Investors' behaviour is highly influenced by emotions and cognitive biases, determining to a big extent the sentiment and the flows of the market and consequently its inefficiencies that are the source of profit as explained in the paragraph before.

Moreover, the dream of every investor would be to take decisions based on predictions on the future price of an asset. The market price of an asset is determined by supply and demand, more specifically, it is the price at which quantity supplied equals quantity demanded. Predicting the price would therefore mean predicting how many investors would buy and sell at a specific price and this is a very challenging task.

In other words, a successful portfolio management strategy needs to be customized and to stem from a profound understanding of the dynamic nature of market forces. Automating this process is not an easy task, due to the variety of the forces driving the market and therefore the diversity and amount of information that needs to be abstracted and efficiently put together to shape an effective strategy.

Also from a mathematical perspective, modeling the behaviour of financial markets is very difficult because of challenges like nonstationarity, poor prediction ability, and weak historical coupling, and this has recently caught the attention of the research community. In the past, this problem has been analyzed using a more static approach, leading to the Modern Portfolio Theory developed by H. Markowitz in 1952 (1) or mathematical formulations in the context of Signal Processing that have shaped today's quantitative finance. However, recent developments in the field of optimization of dynamical systems have created a growing interest in applying new techniques to the portfolio optimization task.

The focus of the thesis will be to automatically create portfolio management actions with an algorithm based on Deep Reinforcement Learning (DRL). In Reinforcement Learning (RL), (2) an agent interacts with the environment at each time-step by executing an action and receiving a reward consequent to the action as well as the new state of the environment after the action. The reward is a feedback relative to the action and to the effect the action had on the environment. The goal is to use this reward signal to optimize the policy used by this agent to make decisions. The term "Deep" refers to the fact that neural networks are employed to build and progressively optimize the decision-making policy used by the algorithm to take decisions. Indeed, the ascent of Deep Learning (3) led to the rediscovery of RL in 2014, since the the power of neural networks enabled RL to obtain impressive results in some domains.

The main reasons why RL has been chosen are the similarity between its paradigm definition and the way the stock market works as well as the fact that this technique bypasses the pretentious step of predicting the future price of the assets and directly outputs investment actions. In other words, using Deep Reinforcement Learning, we don't have to get a full understanding of how the stock market works which would be an almost impossible task as described before, while in order to predict the exact future price that would be necessary. Moreover, DRL is created for decision-making and this makes it suitable for this task. In addition to that, currently, there are very few analytical techniques for the portfolio optimization problem and they tend to be very basic and simple.

The thesis is divided in four chapters after the introduction. Chapter 2 and 3 are dedicated to presenting the necessary background and reviewing the most relevant papers respectively in the portfolio management domain and in DRL, sequence modelling and imitation learning. Chapter 4 discusses the algorithm implementation, the libraries used and the contributions made to obtain the resulting final models. A DRL algorithm was first implemented and then enhanced with the use of the GTrXL architecture and then two different forms of imitation learning in order to address its main inherent limitations. Eventually, in Chapter 5, the results and the conclusions are presented.

1.1 Portfolio optimization problem definition in the Deep Reinforcement Learning Paradigm

As shown in Figure 1, the agent interacts with the environment at each timestep by executing an action *At* and receiving an observation and a reward consequent to the action. The observation *Ot* describes the next conditions of the environment after the action has been taken and the reward *Rt* is a feedback signal relative to the action *At* in the environment. The goal is to maximise the total cumulative reward over all timesteps. The agent decides which action to take based on the state and the rewards he has received in the past. The agent state is the agent's internal representation of the history of all environment observations so it refers to all previous timesteps. This can be contextualized for the portfolio optimization case considering the observations as the stock prices, the indicators and the balance or investable amount; the reward as the investment return at a specific timestep and the action as the portfolio weights decided by the agent given the state. The trading environment is a simulation of a discrete-time stochastic dynamical system. The deep reinforcement learning agent learns a decision-policy and therefore a trading strategy that optimizes the reward through experience.

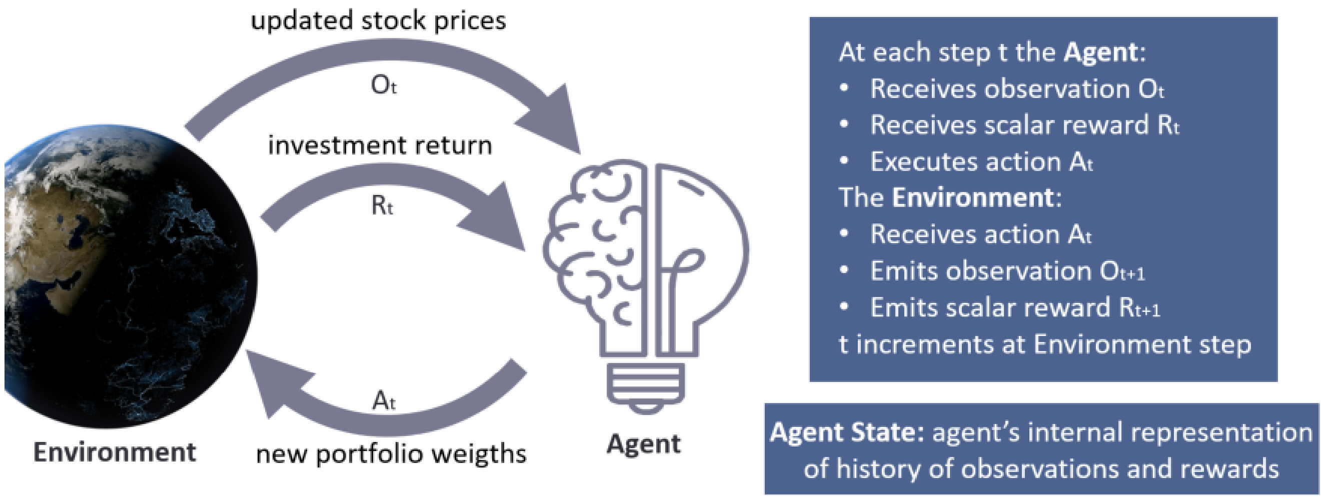

updated stock prices
O_t
investment return
R_t
A_t
Environment
new portfolio weigths
Agent
At each step t the **Agent**:
• Receives observation O_t
• Receives scalar reward R_t
• Executes action A_t
The **Environment**:
• Receives action A_t
• Emits observation O_{t+1}
• Emits scalar reward R_{t+1}
t increments at Environment step
Agent State: agent's internal representation of history of observations and rewards

Figure 1: Reinforcement Learning schema for the Portfolio Management Problem

For this thesis work, the algorithm has been tested on a portfolio composed of the Dow Jones Industrial Average index stocks. For most of the work, excluded the different time granularities section 5.3, the considered timesteps are days, therefore the agent code receives input observations and rewards every day and outputs a continuous action between -1 (sell as much as possible) and 1 (buy as much as possible) for each stock. Transaction costs are considered and amount to 0.1% for both buying and selling. The environment code receives actions from the agent, performs the buy and sell operations and returns the next timestep observations (updated balance, stock prices and indicators values, portfolio weights) as well as the reward (portfolio return calculated from the previous timestep)

1.2 Research Question

As already briefly mentioned, the main reason for using Deep Reinforcement Learning (DRL) for Portfolio Optimization is that, with respect to other techniques, its formulation allows to directly output a decision without trying to predict the future stock price's trend that is a much more difficult task. And, in addition to that, there are very limited mathematical models for Portfolio Optimization and they are usually derived from very basic rule-based inference due to the inherent complexity of the problem. Consequently, the contributions of this master thesis consist in exploring this type of Deep Learning approach to Portfolio Optimization and in trying to address the main limitations that have been identified in using RL for this specific task:

Firstly, for the algorithm to be effective, the representation the model has of the state of the environment at a specific timestep must contain all useful information from the history. The challenge, in this case, is to create a tensor as a state where we include as much useful information from the past as possible with limited noise. Moreover, Reinforcement Learning algorithms are prone to be unstable, especially when trained with not enough data. Therefore, the state representations have been built by using sequence models so as to efficiently capture more information about the past and in particular the GTrXL (4) model has been chosen. This because transformer-based models are state-of-the-art in sequence modeling and because this model has been designed specifically to give more stability to RL algorithms.

Secondly, RL has given great results in the game-playing domain, where data can be collected with no limits by just playing the game, but it has been less effective in other domains with less input data available. The stock market offers public easy-accessible data, but the amount is limited and often not enough for DRL requirements. Therefore, expert actions obtained using future returns have been used to pre-train the model in a supervised manner before using RL. In this way, the limited data have been used twice, the first time using imitation learning pre-training so that the second time Reinforcement Learning doesn’t have to start from scratch. In addition, the model has been tested with different time granularities so as to find the best trade-off between data volume and noise.

Finally, a third variant has been studied starting from the results and limitations of the first two variants explained above. Again exploiting imitation learning and the expert actions, an offline Reinforcement Learning algorithm was trained over a set of randomly sampled (from S&P500) portfolios to have a bigger and more diverse dataset.

2 Financial Background and literature review

The aim of this Chapter is to introduce all the theoretical concepts that are necessary to grasp a thorough and complete understanding of the portfolio management problem from a financial point of view contextualized to our research question and goal.

2.1 Stocks

The term "asset" is a broad definition referring to any item with an economical value. For the purpose of this work, we will focus on stocks to avoid having an excessively wide research scope. Stocks are equities, other asset classes include fixed income such as bonds, cash and cash equivalents, real estate, commodities, currencies, cryptocurrencies and structured products. The most relevant features that distinguish stocks from other asset classes are the following:

- High volatility and consequently higher return with respect to most mentioned asset classes. Only cryptocurrencies, some commodities and some structured products can be more risky.
- Ownership rights in a company: in this work a simplistic view of stocks is considered and therefore ownership rights are excluded from the picture. Consequently, stocks are merely considered as an investment asset implying a financial position in which an investor seeks to profit from changes in the price without acquiring the traditional privileges associated with holding the underlying asset. This would be possible in the real market with a derivative contract known as a Contract for Difference (CFD)
- High liquidity: this allows us to avoid making liquidity considerations that would be necessary otherwise.
- Public market asset: this affects stocks behaviour in multiples ways such as regulations, general market influence including financial statements publications, economic indicators, geopolitical events and so on.

2.1.1 Common Measures of Stock Prices Risk and Return

2.1.1.1 Return

The concept of expected return refers to the weighted average of the possible returns, weighted on their probabilities. Expected return can also be approximated with the average over a period of time of the historical realized return that is instead the return that occurred over a particular time period in the past. The error of this approximation can also be estimated as the standard deviation divided by the square of the number of observations (5)

$$\text{Gross Return: } R_t = \frac{St}{St-1}$$

$$\text{Simple Return: } R_t = \frac{St}{St-1} - 1$$

$$\text{Expected return: } E[R] = \sum_R P_R \times R$$

$$\text{Average historical realized return: } \bar{R} = \frac{1}{T}\sum_{t=1}^{T} R_t$$

$$\text{Standard error} = \frac{\text{SD (Individual Risk)}}{\sqrt{\text{Number of Observations}}}$$

2.1.1.2 Variance, Standard Deviation and Volatility

Variance is the expected squared deviation from the mean and Standard Deviation is the square root of the variance. Both are measures of the risk of a probability distribution. In finance, the standard deviation of a return is also referred to as its volatility. The standard deviation is easier to interpret because it is in the same units as the returns themselves. (5)

$$Var\,(R) = E[(R - E[R])^2] = \sum_R P_R \times (R - E[R])^2 = \frac{1}{T-1}\sum_{t=1}^{T} (R_t - \bar{R})^2$$

$$SD(R) = \sqrt{Var(R)} = \sqrt{\frac{1}{T-1}\sum_{t=1}^{T} (R_t - \bar{R})^2}$$

2.1.1.3 Risk

The total risk is usually considered as incorporating two different risks: the systematic or common risk and the independent or unsystematic risk. Common risk shows perfect correlation, impacting the entire market while independent risk is uncorrelated and affects a specific security. As multiple stocks are combined in a big portfolio, the independent risk component for each stock tends to average out and be diversified, however, the systematic risk will impact all firms and will not be diversified.

Real-world firms deal with both market-wide risks and firm-specific risks therefore only the unsystematic risk will be diversified when many stocks are put together into a portfolio. Thus volatility will decline until only common risk remains. Uncorrelated risk can be diversified, hence investors are not compensated for holding firm-specific risk and the risk premium is zero. If the diversifiable risk was rewarded with a supplementary premium, then investors could exploit this by buying the stocks, earning an extra premium, and, at the same time, eliminating the risk by diversifying.

The risk premium depends on systematic risk and does not depend on diversifiable risk. This implies that a stock's volatility or standard deviation, representing total risk, is not particularly informative when determining the risk premium investors will receive. Therefore, there should be no direct relationship between volatility and average returns for individual securities. Consequently, in order to estimate a security's expected return, we need to find a measure of its systematic risk. Measuring the systematic risk of a stock involves determining the extent to which the variability of its return is due to systematic risk versus unsystematic risk. (5)

2.1.1.3 Beta

To understand how responsive a stock is to systematic risk, it is necessary to look at the average change in the return for every 1% change in the return of a portfolio that is only affected by systematic risk. This sensitivity to systematic risk is called Beta (β). Beta is exclusively a measure of systematic risk. A security's beta is determined by how much its underlying revenues and cash flows are influenced by general economic conditions. Stocks within cyclical industries are likely to display greater sensitivity to systematic risk and have higher betas with respect to stocks in less sensitive industries. (5)

2.1.2 Stock Prices Distribution

2.1.2.1 Markov Process

It is relevant to introduce the concept of Markov Process since Stock prices are commonly assumed to follow this kind of process. A Markov process is a specific form of stochastic process in which a future forecast is only based on the current value of a variable. The history of the variable and the way that the present evolved from the past are not considered relevant.

Future predictions involve uncertainty and have to be modelled as probability distributions. The Markov property implies that the probability distribution of a future price value is independent of the specific trajectory the price has taken in the past. This property aligns with the weak form of market efficiency, meaning that the current value of the stock price incorporates all the information from historical prices. If this was not true, technical analysts could make greater profits than the average just by using charts of historical stock prices, however, there is very limited evidence that this is the case.

Actually, weak-form market efficiency and the Markov property hold because of market competition. Investors follow the stock market closely and also base their decisions on past trends resulting in current stock prices reflecting past prices. For instance, if multiple investors foresee a price hike due to the observation of a particular condition, they will immediately buy the stock, making demand and therefore prices rise before the effect can be observed and therefore eliminating it. (6)

2.1.2.2 Wiener process

A Wiener process is a particular case of a stochastic Markov process with a mean change of 0 and with a variance rate of 1 per year. It was born in physics, where it is used to study Brownian motion, the diffusion of minute particles suspended in fluid, as well as other types of diffusion. More formally, for $z(t)$ to be defined as a Wiener process, it has to respect the following properties:

- $z(t)$ is a continuous random variable for t >= 0.
- Initial condition: $z(0) = 0$.
- Increments: $z(t) - z(s)$ follows a normal distribution $N(0, \sqrt{(t-s)})$ for 0 <= s <= t.
- Independence: $z(t) - z(s)$ is independent of $z(u)$ for 0 <= u <= s if 0 <= s <= t.

From this, those other properties can be implied:

- $E[z(T) - z(0)] = 0$. This means that the drift rate (mean change per unit time) is equal to zero
- Standard deviation of $[z(T) - z(0)]$ is $\sqrt{T}$
- Variance of $[z(T) - z(0)]$ is T.This means that the variance rate (variance per unit time) is equal to one.
- $z(t) - z(s) = \sqrt{(t\ -\ s\ )}\ \varepsilon$ where $\varepsilon \sim N(0,1)$

By generalizing the above defined Wiener process, we can extend the definition to the case where the drift rate is not necessarily equal to zero and the variance rate is not necessarily equal to one. This generalized Wiener process (dz) for a variable x can be defined with two constants a and b as:

$$dx = a * dt + b * dz$$

It can be shown that such a process has an expected drift rate of a and a variance rate of b^2 (6)

2.1.2.3 The process of a stock price

Markov and Wiener processes formulation allow us to define the stochastic process usually assumed for the price of a non-dividend-paying stock. With respect to a generalized Wiener process, a key difference can be outlined: while a Wiener process has a constant expected drift rate and a constant variance rate, investors' required return does not depend on the value of the stock's price, and the percentage required return expectation is what remains constant. In other words, the constant drift rate assumption is inappropriate and it has to be replaced by the assumption that the percentage expected return is constant. This can be achieved by dividing the change in the stock price by its initial value. From the expected return formula, using the drift rate μ, we can say that the change in price is equal to the value of the price multiplied by the drift rate and the time delta. By making the time delta go to zero and moving the value of the price to the left side of the equation, we can obtain the following:

$$dS = \mu\ * S * dt \text{ that becomes } dS/S =\ \mu\ *\ dt$$

In this formula, since the coefficient of dz is zero, the assumption is that there is no uncertainty, that is certainly not the case. Taking into account uncertainty, it becomes:

$$dS = \mu * S * dt + \sigma * S * dz \text{ or } dS/S = \mu * dt + \sigma * dz$$

The final equation shows the most commonly employed model for describing stock price dynamics. Where:

- μ denotes the expected return of the stock
- σ represents the volatility of the stock price
- σ^2 is considered as its variance rate.

In a risk-neutral environment, μ would be equivalent to the risk-free rate, denoted as r. (6)

2.1.2.4 Lognormal distributions considerations

Ito's lemma provides a method to calculate the stochastic process of a function based on the process of its underlying variable. A fundamental point is that the Wiener process dz underlying the process for the variable is identical to the Wiener process underlying the process for the variable's function. Both are influenced by the same source of uncertainty. When Ito's lemma is used on the stock price equation, we can derive the stochastic process of $ln(S)$. The outcome demonstrates that $G = ln(S)$ follows a generalized Wiener process with a constant drift rate of $\left(\mu - \frac{\sigma 2}{2}\right)$ and constant variance rate of σ^2.

The change in $ln\ (S)$ over a period going from time 0 to a future time T is normally distributed, with mean $\left(\mu - \frac{\sigma 2}{2}\right) * T$ and variance σ^2*T. Consequently, it can be proved that ln (S_T) is normally distributed with the same variance and mean $\ln(S_0) + \left(\mu - \frac{\sigma 2}{2}\right) * T$. A variable is lognormally distributed if its natural logarithm follows a normal distribution. Therefore, the stock's price at time T, given its price today, is lognormally distributed and the standard deviation of the logarithm of the stock price is equal to $\sigma * \sqrt{T}$ that is proportional to the square root of the time horizon. (6)

2.1.2.5 Log Returns

$$R_t = ln(S_t) - ln(S_{t-1})$$

As we saw in the last paragraph, we assume the asset price S to be lognormally distributed, then log-returns show a normal distribution and are additive, while simple returns are shifted log-normally distributed which is less practical. The arithmetic mean of the daily log-returns serves as an unbiased estimator for the drift term, offering practical advantages over the geometric mean needed with simple returns, which tends to introduce bias. The arithmetic mean of the simple returns does not have the same practical meaning.

Consequently, log-returns contribute to data stationary by stabilizing the variance and linearizing the relationships over time. Stationarity is desirable when dealing with time series because it simplifies modeling and makes it easier to identify patterns over time. Moreover, they show statistical properties such as additivity for consecutive periods and a more symmetric distribution.

In other words, multiplicative simple returns can be transformed into additive with the log operation and this makes it easier to analyze over multiple periods. For instance, when stock prices jump and then return to their original value, simple returns have an average different from zero, while log-returns sums up to 0. Therefore, log-returns offer an advantage as they cancel out during price fluctuations, unlike percentage changes and this is especially true for buy-and-hold strategies.

On the other side, it must be taken into account that the average log-return over a period of time provides limited information on price's past dynamics, since it is equal to the log of the final price minus the log of the initial price and therefore it just depends on those two prices. In addition, it is relevant to point out that log-returns are not additive across trades. In summary, when aggregating returns across trades keeping time fixed, simple returns linearly aggregate, while log-returns don't. Conversely, when aggregating across time keeping the trade fixed, simple returns don't aggregate linearly, and log-returns do. (6)

2.2 Portfolio Optimization

A portfolio refers to a collection of financial assets and, it can be represented as a vector of weights. Each component of this vector shows the amount invested in a specific asset at a particular time. In other words, portfolio weights represent the fraction of the total portfolio value of a stock.

2.2.1 Portfolio Risk and Return

2.2.1.1 Portfolio Expected Return

The expected return of a portfolio is the weighted average of the expected returns of the investments within it. (5)

$$E[R] = \sum_{i=1}^{n} w_i \times E[R_i]$$

2.2.1.2 Portfolio Risk, Covariance and Correlation

When creating a portfolio with a set of stocks, the total risk of the Portfolio is reduced with respect to the sum of the single stock's risk due to diversification. Risk is diversified according to how the stock prices move together. To calculate it, it is necessary to understand the degree to which the stocks' returns move together.

Covariance represents the expected product of the deviations of two returns from their means. A positive covariance implies that the two returns tend to move in the same direction, while a negative covariance indicates that the two returns move in opposite directions.

Correlation measures the common risk among stocks, independent of their individual volatility. The correlation between two stocks always falls within the range –1 to +1. The risk of a Portfolio is calculating using variances and standard deviations of individual assets, but also taking into account how the different assets' movements influence each other. The variance of a portfolio is the weighted average covariance of each stock with the portfolio. The Portfolio's risk will be lower than the weighted average volatility of the individual stocks apart from the case in which all the stocks show a perfect positive correlation of +1 with one another. (5)

Covariance of two stocks i and j:

$$Cov(R_i, R_j) = E[(R_i - E[R_i])(R_j - E[R_j])] = \frac{1}{T-1}\sum_t \left(R_{i,t} - \bar{R}_i\right)\left(R_{j,t} - \bar{R}_j\right)$$

Correlation of two stocks i and j:

$$Corr(R_i, R_j) = \frac{Cov(R_i, R_j)}{SD(R_i)SD(R_j)}$$

Portfolio Variance:

$$Var(R_P) = Cov(R_P, R_P) = Cov\left(\sum_i x_i R_i, R_P\right) =$$
$$= \sum_i x_i\, Cov(R_i, R_P) = \sum_i x_i Cov\left(R_i, \Sigma_j x_j R_j\right) = \sum_i \sum_j x_i x_j Cov\left(R_i, R_j\right)$$

2.2.1.3 Sharpe Ratio

Sharpe ratio is a measure of a portfolio risk-adjusted performance that is to say the ratio of reward-to-volatility provided by a portfolio. The calculation may and is usually based on historical returns. A higher Sharpe ratio is better when comparing similar portfolios because it means that the ratio between return and volatility is higher and therefore that the return for a determined level of volatility is higher. The Sharpe ratio may still be a simplistic measure of a portfolio performance since it just keeps into account return and volatility. (7) (5)

$$Sharpe\ Ratio = \frac{Portfolio\ Excess\ Return}{Portfolio\ Volatility} = \frac{E\,[R_P] - r_f}{SD\,(R_P)}$$

2.2.2 Markovitz Portfolio Theory

The concept of Efficient Portfolio has been introduced by Markovitz with its Portfolio theory in 1952 and it describes a Portfolio where there is no way to reduce the volatility of the portfolio without lowering its expected return or no way to increase its expected return without increasing its volatility.

As you can see in Figure 2 shown below all the efficient portfolios can be shown as a curve in the return-volatility graph created by changing the weights of a portfolio with two stocks (AAPL and BRK.B). All the blue points are efficient portfolios and the difference between them can be considered the risk-appetite of the portfolio. While by lowering the weight of APPL to less than 15.48% we go to the set of inefficient portfolios.

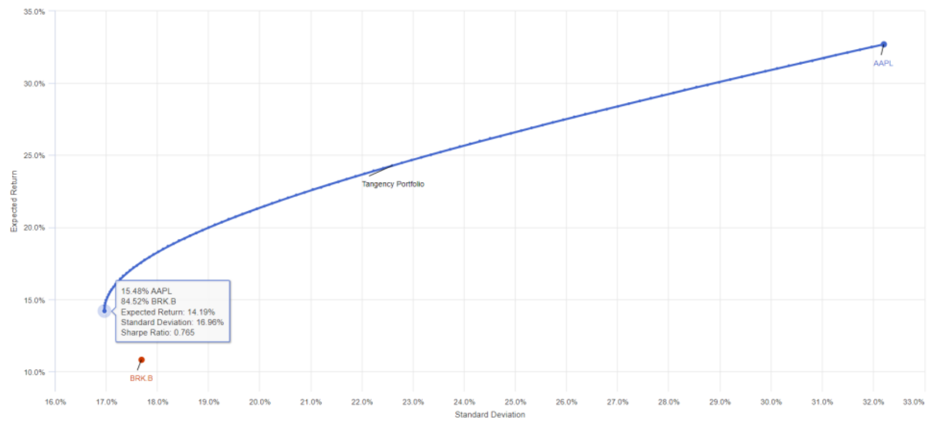


Figure 2: Efficient Frontier Portfolio of APPL and BRK.B obtained on portfoliovisualizer website with calculation period 2005-2022

Correlation does not affect the expected return of a portfolio. However, it will influence the volatility of the portfolio. As a matter of fact, with a lower correlation, we will have a lower volatility. The following pictures obtained by varying the weights of a portfolio composed of stocks (E(R)=10%, vol=13%) and bonds (E(R)=3,5%, vol=6%) show that the curve will bend to the left when correlation decreases.

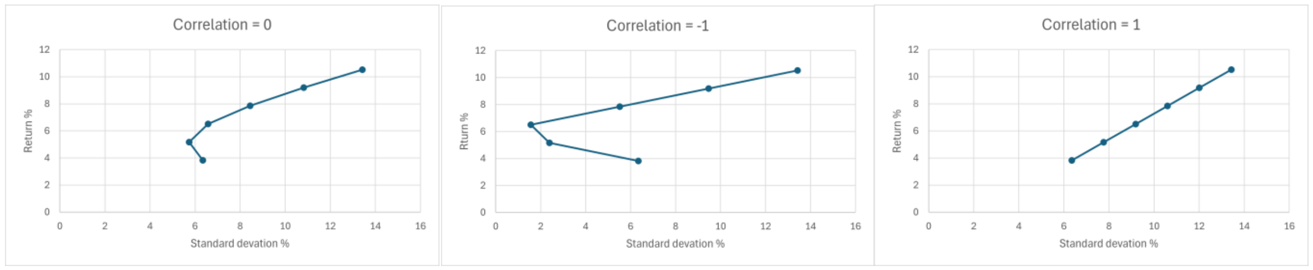


Figure 3: Effect on Returns of reducing correlation

Moreover, it is also possible to notice that the more stocks we add to the portfolio, the more the efficient frontier moves more and more to the left, implying that with more stocks we can obtain the same return with less volatility because we are diversifying more and more and therefore we are eliminating more and more firm-specific risk.

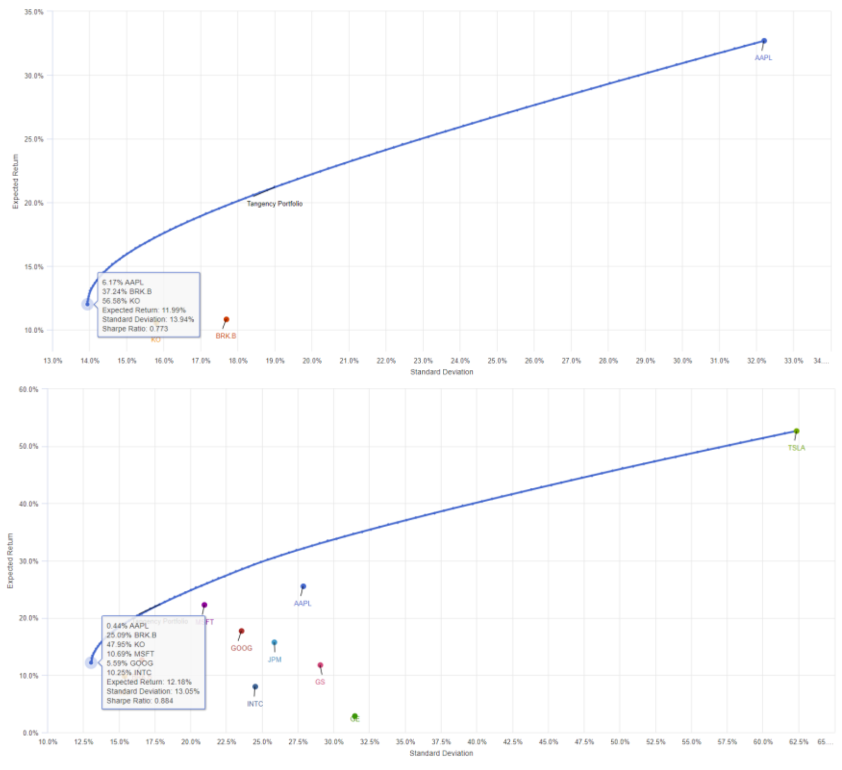


Figure 4: Effect of adding more stocks (3 and 10) to Efficient Frontier Portfolio calculation

Moreover, by adding to the picture an investment in the risk-free rate, it is possible to identify the tangent portfolio. This is the portfolio where the line with the risk-free investment is tangent to the efficient frontier of risky investments. It can be obtained on the graph by tracing a tangent line to the efficient frontier curve line starting from the risk-free investment. This portfolio will also be the one with the greatest Sharpe ratio. Therefore, mixing the risk-free asset with the tangent portfolio will give the best possible compromise in terms of risk and return. From this, it can be inferred that the tangent portfolio is efficient and that all the efficient portfolios can be obtained with a mixture the tangent portfolio and the risk-free. All investors should invest in the tangent portfolio without considering their risk appetite. Their preference will only influence how much funds to allocate to the tangent portfolio with respect to the risk-free investment. Prudent investors will invest a limited amount in the tangent portfolio while aggressive investors will allocate more to the tangent portfolio. (1) (5)

2.2.3 CAPM, Alpha, biases and market inefficiencies

2.2.3.1 Capital Asset Pricing Model

$$E[R] = \textit{Risk-Free Interest Rate} + \textit{Risk Premium} = r_f + \beta \times \left(E[R_{Mkt}] - r_f\right)$$

The equation reported above is often called the Capital Asset Pricing Model (CAPM) where the market risk premium or expected return of the market is the investors expected reward when holding a portfolio with a beta equal to one. The CAPM is a fundamental method for the estimation of the cost of capital utilized concretely. Using it, it is possible to find the efficient portfolio of risky assets without having any knowledge on each security's expected return. (5) (8)

Three assumptions are necessary for the model to be effective:

- Investors are able to buy and sell assets at fair market prices assuming therefore no taxes or transactions costs and can borrow and lend at the risk-free interest rate.
- Investors only hold efficient portfolios meaning that they only own portfolios giving the maximum expected return for a particular level of volatility.
- Investors envisage the same volatilities, correlations, and expected returns of assets and therefore they will look forward to the same portfolio of risky assets.

2.2.3.2 Security Market Line and Capital Market Line

When the previously mentioned assumptions are true, the optimal portfolio is constituted by a blend of risk-free assets and the market portfolio. If the tangent line intersects the market portfolio, it takes the name capital market line (CML). Moreover, a direct correlation exists between a stock's beta and its expected return. Concerning this, the security market line (SML) is depicted as a line connecting the risk-free investment with the market. In addition to that, the CAPM implies that if the expected return and beta for individual assets are depicted on a graph, they should align along the SML. The CML instead illustrates portfolios merging the risk-free asset and the efficient portfolio thus showing the highest expected return achievable for a specific volatility level. Therefore, the market portfolio lies on the CML and all other assets contain risk that can be diversified and are on the right side of the CML. The SML plots the expected return for each assets with respect to its beta with the market. Eventually the CAPM states that the market portfolio is efficient, so all single securities and portfolios should be on the SML. In Figure 5, the CML and of the SML are shown. (5)

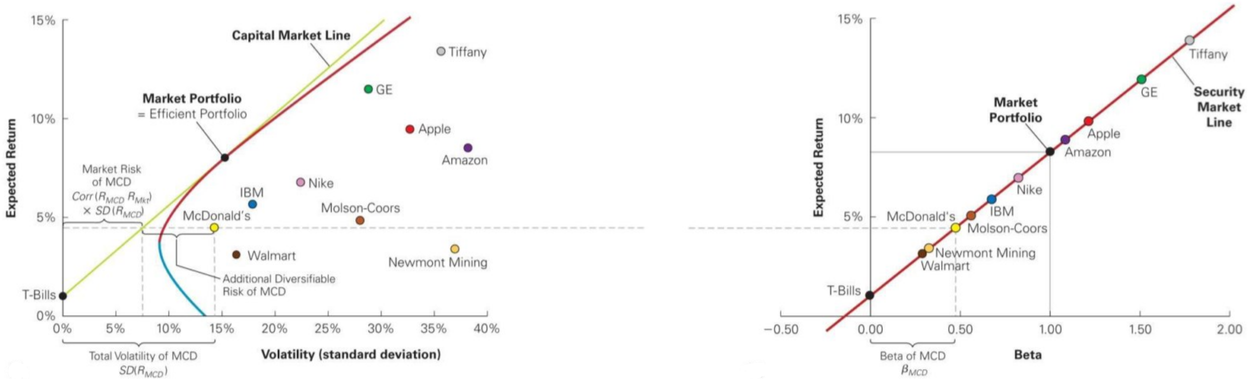


Figure 5: Capital Market Line and Security Market Line from (5)

2.2.3.3 Alpha

Keeping in mind those definitions, it can be assumed that investors will look forward to achieving better portfolios performances by evaluating the expected return of an asset with respect to its required return from the Security Market Line. The disparity between a stock's expected return and its SML required return is referred to as stock's alpha. In an efficient market portfolio, all stocks lie on the SML and present an alpha value equal to zero. Investors can optimize the performance of their portfolios by owning more stocks with alphas greater than zero and less stocks with alphas lower than zero. (5)

2.2.3.4 Market inefficiencies and investor biases

Within the CAPM, investors are expected to own the market portfolio and risk-free investments, however, taking this for granted overestimates the quality of an investor's knowledge or trading abilities. Moreover, all investors would have to accurately exploit the information they have, as well as all the information that can be inferred from market prices or other trades. Even if an investor can access extensive information, owning the market portfolio ensures an alpha of zero. Since the average portfolio of all investors constitutes the market portfolio, the average alpha equals zero. Only if some investors obtain negative alphas, some other investors will be able to earn positive alphas, assuming the market portfolio is efficient. The market portfolio is not efficient when either multiple investors give a wrong interpretation to information and think they are gaining a positive alpha when their alpha is instead negative or when they prioritize aspects of their portfolios other than expected return and volatility, and therefore accept to own inefficient portfolios of assets. (5)

Historical data suggests that investors do not properly diversify their portfolios or are making biased decisions, some examples are the following:

- *Familiarity bias*: preferring to own familiar assets. It has been observed that investors appear to give a lot of importance to their own experience with respect to historical data. For instance, growing up in a bullish period results in higher tendency to invest with respect to someone who was born in a bearish period.
- *Overconfidence bias*: when investors are too confident on their guess and end up with excessive trading.
- *Disposition effect*: selling winner stocks and buying looser stocks.
- *Informational cascade effects*: when investors use information of other traders that has some inaccuracies rather than basing decisions solely on their own information. This is related to trading based on recommendation and it has been proved that investors are more likely to invest in stocks that have recently gained more mediatic attention.
- Being more interested in *performance with respect to other investors* rather than absolute performance.
- *Curiosity* to invest in new or different assets.
- Even the *weather* may influence investors mood and decisions.

2.2.4 Performance indicators

The simplest and most used portfolio performance indicators such as expected return, volatility and Sharpe ratio have already been introduced in section 2.2.1, however, there are some other ratios that is useful to introduce:

Sortino Ratio

$$\frac{E[R_P] - r_f}{\sigma d}$$

Where σd is the standard devation of the downside. Indeed the Sortino ratio is different from Sharpe because it uses the standard deviation of the downside risk instead of the normal standard deviation that includes both upside and downside risk. This since by focusing only on the negative oscillations from the mean, we exclude positive volatility that is beneficial and therefore it is thought to give a better view of a portfolio's risk-adjusted. To sum up, the Sortino ratio measures the return weighted on the amount of bad risk incurred. (9)

Max Drawdown Ratio

The maximum drawdown represents the most substantial observed decline in a portfolio from its highest point to the subsequent lowest point before a new peak is reached. It serves as an indicator of the downside risk incurred over a particular timeframe. In other words, Maximum drawdown is a measure of an asset's largest price drop from a peak to a trough.

$$Maximum\ Drawdown\ (MDD) = ((Peak\ Value\ -\ Trough\ Value))/(Peak\ Value)$$

Where Peak Value is the highest point of value before a drop and Trough value represents the lowest point reached after the peak. It is important to note that while MDD gauges the extent of the most substantial loss, it does not take into account the frequency of losses or the magnitude of any gains. (10)

Value at Risk (VaR)

Portfolio Value at Risk (VaR) serves as a risk metric assessing the potential loss in a portfolio's value over a specific timeframe for a certain confidence level. It gives an approximation of the maximum loss a portfolio could experience within a defined probability. This metric estimation can be carried out with three different modalities: the historical, variance-covariance, and Monte Carlo methods. The formula showed below is with the variance-covariance method and a normal distribution for the return of the stocks. It's worth pointing out that VaR gives an estimation of possible losses, but it does not inform about exceptional events that go over the confidence level. Furthermore, the accuracy of the estimate is influenced by the assumptions made during calculation. When managing risk, usually VaR is used together other measures so as to get a full understanding of portfolio risk.

$$VaR = Portfolio\,Value \times (\mu - Z \times \sigma)$$

Where Portfolio Value indicates the current value of the portfolio, μ is the expected portfolio return, Z is the critical value corresponding to the desired confidence level and σ is the standard deviation of the portfolio return. (11)

2.3 Technical Analysis

Technical analysis is a methodology providing investors with valuable tools to predict and study demand and supply and its effect on prices, volume and implied volatility. This is achieved by interpreting market trends, price patterns, and trading signals often with charts and tools to visually analyze historical price movements. Trends are identified using indicators and trendlines; supports and resistance levels are researched to identify points where securities commonly change direction. Unlike fundamental analysis, which focuses on a company’s financial statements, technical analysis is based solely on historical prices and volume data.

Technical analysis comes with a lot of objections and critics, and its efficacy is debated due mainly to the fact that theoretically, in the financial market, the past is not informative and neither a proxy for the future and future prices do not depend on past prices as also better explained in the previous chapters. However, in practice, the assumption that past trading activity and price changes of a security can be useful indicators of the security's future price movements has some credibility.

As a matter of fact, history still tends to repeat itself and this repetitive nature of price trends is often attributed to market psychology, which is usually predictable based on emotions like fear or enthusiasm. Patterns identified in graphs and charts are used to study market emotions and their influence on market movements and trends. Moreover, analysts assume that all sorts of information from accounting to psychology is already embedded in the stock price and this would be consistent with the efficient market hypothesis described in the previous chapters. Eventually, another assumption many techniques are based on, is that prices will show trends independently of the range of time considered. That is to say, a stock price tends to keep the trend rather than moving randomly.

2.3.1 Main indicators

Below the technical indicators used in this work and their derivations will be explained. All technical indicators values are calculated using the Stockstats library (12) therefore the formulas reported below are directly derived from the library's code.

2.3.1.1 Simple Moving Average:

Simple moving average calculates the rolling arithmetic average of prices over a specified range. This is a very simple technical indicator that can help to estimate whether an asset price will keep or reverse a bull or bear trend.

$$SMA = \frac{(Sum\ of\ values\ in\ the\ rolling\ window)}{(Number\ of\ values\ in\ the\ rolling\ window)}$$

2.3.1.2 Exponential Moving Average:

The Exponential Moving Average (EMA) is a rolling average that weights more and gives more importance to most recent data points. EMA can be used to create signals when the current average diverges or intersects the historical average. The length of EMA is an informative parameter that is often fine-tuned or changed since different lengths provide different signals and information. (13)

Stockstats library calculation:

$$EMA\_t = \alpha * value\_t + (1 - \alpha) * EMA\_\{t-1\}$$

- ***EMA_t*** is the EMA at time t,
- ***value_t*** is the value of the time series at time t,
- $\boldsymbol{\alpha}$ is the smoothing factor, which can be calculated as $\alpha = 2/((span + 1))$ or provided directly using the alpha parameter. Where span is equal to the window.

2.3.1.3 Moving Average Convergence Divergence:

Moving average convergence divergence is a momentum indicator used to identify the interrelationships between two moving averages of a stock's price. It is obtained by subtracting the 26-day and 12-day exponential moving average of the closing price. Moreover, also a nine-period EMA of the MACD itself called signal line is used. This signal line is considered the slower line because the points plotted move less and it is the line determining buy and sell decisions. The MACD gives indications on whether the current momentum is bullish or bearish and helps to find entry and exit points for trades. Multiple strategies based on EMA can be used by traders, some are the histogram, the crossover, the zero-cross, the money flow index, and the relative vigor index. MACD can be risky in some situations where a reversal signal can give misleading indications. (14)

Stockstats library calculation:

- Calculate 12-day Exponential Moving Average and 26-day Exponential Moving Average (EMA):

$$ema\_short = ema(close, short\_w)$$

$$ema\_long = ema(close, long\_w)$$

- Calculate MACD Line and 9-day Exponential Moving Average (EMA) of MACD Line (Signal Line):

$$macd = ema\_short - ema\_long$$

$$macds = ema(macd, signal_w)$$

2.3.1.4 Bollinger Bands:

Bollinger Bands are used to create oversold or overbought signals. They are made of three lines including a simple moving average, plus an upper and lower band. Those last two bands are typically two standard deviations away in negative and positive from the simple moving average that is 20 days in our case, but it can also be of different length. If the price repeatedly crosses the upper Bollinger Band, it indicates an overbought signal. While when the price touches the lower band it suggests an oversold signal. (15)

Stockstats library calculation:

- The input parameters are the period for the moving average (M) and the multiplier for the standard deviation (K)
- The moving average (MA) is calculated using the close prices over the M period.
- ***boll***: The middle band, representing the moving average (MA) of the closing prices.
- ***boll_ub***: The upper band, calculated as the moving average plus K times the standard deviation (σ).
- ***boll_lb***: The lower band, calculated as the moving average minus K times the standard deviation (σ).

$$boll = MA$$

$$boll\_ub = MA + K * sigma$$

$$boll\_lb = MA - K * sigma$$

2.3.1.5 Relative Strength index:

The relative strength index (RSI) is a widely used momentum oscillator. It allows to identify signals about bullish and bearish price momentum, and it is usually plotted below the chart of the stock's price. When the RSI is higher than 70 it indicates an overbought condition, while, when below 30, it suggest and oversold condition. The RSI intersecting the overbought line from below or the oversold line from above is usually interpreted as a buy or sell signal. The RSI can assume values in between the range zero and 100. (16)

Stockstats library calculation:

- ***Price changes***: calculated as the difference between consecutive closing prices.
- ***close_pm***: Average gain, considering only positive changes.
- ***close_nm***: Average loss, considering only negative changes.
- ***p_ema***: Smoothed moving average of positive changes. Calculated as simple moving average of *close_pm*
- ***n_ema***: Smoothed moving average of negative changes. Calculated as simple moving average of *close_nm*
- ***RS***: Relative strength, calculated as the ratio of p_ema to n_ema.

$$RS = p\_ema / n\_ema$$

$$RSI = 100 - (100 / (1 + RS))$$

2.3.1.6 Commodity Channel Index:

The Commodity Channel Index (CCI) indicator calculates the difference between the current price and the historical average price. If the CCI is positive, it means that the price is higher than the historic average. On the other hand, when the CCI is negative, the price is lower than the average. The CCI is unbounded, so it can oscillate higher or lower indefinitely. As a matter of fact, overbought and oversold signals are usually identified for every stock by using historical extreme CCI levels where the price reversed. (17)

Stockstats library calculation:

- Typical Price (TP):

$$TP = ((High + Low + Close))/3$$

- 20-period Simple Moving Average of TP (tp_sma):

$$tp\_sma = sma(TP, 20)$$

- Mean Absolute Deviation of TP (mad):

$$mad = mad(TP, 20)$$

  Where mad value is calculated by applying the function $f(x) = |x - \bar{x}|$ for each specified rolling window

- Commodity Channel Index (CCI):

$$CCI = ((TP - tp\_sma))/((0.015 * mad))$$

2.3.1.7 Directional Movement Index:

The directional movement index (DMI) indicates both the strength and direction of a price movement aiming to minimize false signals. It employs two standard indicators, one negative (-DM) and one positive (+DN), along with a third, the average directional index (ADX), which, despite being non-directional, represents momentum. A wider spread between the two primary lines, indicates a more robust price trend. When +DI significantly goes above -DI, the price trend is upward. If -DI is higher than +DI, then the price trend is downward. (18)

Stockstats library calculates it with the following steps:

- Calculate High and Low Price Differences:

$$hd = col\_diff('high^\wedge')$$

$$ld = -col\_diff('low^\wedge')$$

- Calculate Plus Directional Movement (PDM) and Minus Directional Movement (NDM):

$$p = hd\, if\ (hd > 0) and\ (hd > ld) else\ 0$$

$$n = ld\, if\ (ld > 0) and\ (ld > hd) else\ 0$$

- Smooth PDM and NDM (if window > 1):

$$pdm = smma(p, window)$$

$$ndm = smma(n, window)$$

- Calculate Plus Directional Index (PDI) and Minus Directional Index (MDI)

$$pdi = pdm/atr * 100$$

$$ndi = ndm/atr * 100$$

Where *atr* is Average True Range: a 14-day smoothed moving average of the true range values. Where True Range is a measure of volatility of a High-Low-Close series calculated as:

$$tr = \max\left[(high - low), abs(high - close\_prev), abs(low - close\_prev)\right]$$

- Calculated Directional Movement Index

$$dx = abs(pdi - mdi)/((pdi + mdi)) * 100$$

3 Deep learning Background and Literature Review

The aim of this Chapter is to introduce Deep Reinforcement Learning and its fundamentals as well as presenting the main research works that are the basis of this thesis contributions such as state-of the-art Deep RL algorithms, advanced time series embedding and imitation learning.

3.1 Reinforcement Learning

Reinforcement Learning was initially introduced by Sutton and Barto in (2) and lies at the intersection of many different fields such as mathematics, computer science, engineering, economics and even psychology spanning over optimal control, machine learning, reward systems and so on. Machine Learning can be considered as divided in three main branches that are supervised, unsupervised and reinforcement learning. While supervised receives input data with labels and unsupervised aims at discovering patterns in raw data without labels, the Reinforcement Learning paradigm is instead quite different, it is indeed designed to optimize decision making. It does so through trial-and-error learning. It takes an action and receives a reward or punishment when transitioning to the next state. In Reinforcement Learning there is no supervision, but only a reward signal that is the enabler for the learning process. This feedback is received with a delay, therefore a lot importance is given to time consistently with any decision-making process that is sequential and where the data received in the future also depends on the agent's past decisions.

The addition of Deep Learning (3) to Reinforcement Learning started from an experiment in 2014 when, given the recent success of deep learning, a group of RL researchers at Google Deepmind decided to try merge this two branches of machine learning to design an algorithm to play Atari (19) and surprisingly for them as well it gave great results. The added value of deep learning becomes clear when we think of it as a very powerful tool that can learn and approximate any kind of function. In our case, it can be both the value function or the policy. So this means that instead of calculating for every new possible state-action pair the value of the value function or of the policy, we approximate those functions with neural networks exploiting the data by progressively adding them to the algorithm. This will be explained in section 3.1.6. (20)

The ideas behind reinforcement learning are derived from behavioural psychology and this makes RL really catchy and interesting, but, concerning the current applications of Reinforcement Learning, it has been less effective in business with respect to the other machine learning branches and this is due to massive data requirements as well as its even greater technical complexity. Although its paradigm can theoretically adapt well to any decision-making field, the main practical achievements are recent and have been achieved mostly in the game-playing domain. Atari was already mentioned as the main first success. Regarding it, it's worth to notice that the model takes as input the frames of the videogame and just from that using first Convolutional Neural Networks and then Reinforcement learning, it takes decision for the game. Later in 2016, again google Deepmind was able to win four games to one against the Go world champion and the game was watched by 200M people. (21) Go is a very popular game in Asia and this was a big achievement because it's a game where a sense and intuition is fundamental and it's not possible to just calculate all combinations as it is possible in chess for example. Finally, another notable achievement is Alphafold whose paper was published in 2021. (22) This application is is not anymore in the gameplaying domain since the model is able to predict with way higher accuracy than before the way aminoacids spontaneously fold to form 3D proteins structure. The 3d structures of proteins is fundamental to their biological functioning and this is why the protein folding problem is important.

3.1.1 Main concepts in Reinforcement Learning

Please refer to section 1.1 in the introduction for a basic definition of the Reinforcement Learning paradigm. It was in fact deemed important for the reader to have an idea of the mechanics of Reinforcement Learning from the beginning in order effectively grasp the research question of this work. In this section, before digging deeper into more details, we add to that initial introduction by explaining the pillar concepts of RL:

- **Agent**: It represents the entity that interacts with the environment to learn and improve a policy based on observations and rewards.
- **Environment**: The external context in which the agent operates. It receives actions and outputs observations and rewards based on the effect of those action on the external environment.
- **Reward**: the reward feedback signal is scalar, and gives an indication of the agents current performance at that time step. The agent's aim is to maximize its total future cumulative value. The reward function is part of the environment and its design is fundamental since it has to efficiently include and represent all the objectives all the algorithms eliminating noise as much as possible.

- **Observations**: It differentiates from the reward because it is not a specific feedback on the previous action, but it represents the environment at the current timestep.
- **State**: The concept of state is crucial, it indicates the whole set of information used to determine the next action and it is a function of the history. Where the term history means the sequence of observations, actions and rewards.
- **Value Function**: It evaluates how good a certain state-action pair is that is to say it predicts the total future expected reward starting from state s and performing a specific action a. For instance, if the current state is characterized by the weights of a portfolio (containing for instance 60% of Apple and 40% of Microsoft stocks) and the current and past prices of Microsoft and Apple, the value function tells us the total expected future return of that portfolio when taking a specific action among the possible ones, for instance selling 10% Microsoft to buy 10% of Apple. Value-based agents use the value function to determine the next action to take.
- **Policy**: It corresponds to the agent's behaviour function. It doesn't output the expected return associated with a state-action pair, but directly the best action to take given a state, so in our example it would output directly whether we should invest more in Apple or Microsoft. Policy-based agents used the policy to determine the next action to take.

So in value-based agents the policy is implicit since we use the value of the value function to determine the best action to take while in policy-based ones there is no value function and the policy directly tells us how to act. (20) (2)

3.1.2 Exploration-Exploitation dilemma

A fundamental aspect of RL is the Exploration vs Exploitation dilemma. When an algorithm is trained, it has to get maximize reward based on the current policy it built, but it is also important to try some random actions because this may lead to the discovery of some even better options since the environment is not fully known. Exploration refers to gaining new experience and finding more information about the environment, while exploitation refers to making the best decision to maximize reward given the current information. Finding the right balance between the two is very complex also because the best long-term strategy may have short-term side-effects. Among the simplest exploration techniques, there are ε-greedy strategies, where the agent chooses a random action (exploration) with probability ε or selects the action with the highest estimated value (exploitation) with probability 1-ε.

Other more advanced algorithms that will be further described later introduce instead entropy regularization to the policy loss function encouraging more stochasticity. (20)

3.1.3 Markov Decision Process, Value Function, Policy, Bellman Equation

Markov processes and the Markov property have already been introduced in section 2.1.2.1. Contextualizing it in RL, we can say that, for the algorithm to be effective and the RL formulation to be valid, the state must be an information state or Markov state that implies it has to embed all useful information from the history. Once we obtained the state, we don't need the history anymore. In other words, the current state has to fully characterise the current situation, and then the future is independent on the past given the present.

$$\text{Markov property: } \mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \dots, S_t]$$

A Markov process or chain can be then defined as a random process without memory that is composed by a sequence of random states with the Markov property. It is relevant to also introduce the concept of transition matrix that is a matrix defining transition probabilities from one state to another, each row of such a matrix has the sum equal to one.

A Markov reward process is a Markov chain with reward values. Therefore, a new concept of return needs to be introduced as the total discounted reward from the current timestep to the end. The discount is the present value of future rewards and it is an important concept because it represents uncertainty about the future and avoids infinite returns when processes are cyclic. From the concept of return, we can better understand the state value function. The state value function of a Markov Process indeed calculates the expected return starting from a specific state s.

$$\text{Return: } G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\text{State value function: } v(s) = \mathbb{E}[G_t \mid S_t = s]$$

Where γ is the discount factor and R is the reward.

The state value function can be divided into two terms that are the immediate reward and the discounted future rewards and this leads to the Bellman Equation that is probably the pillar equation of Reinforcement Learning. With this separation, it is easier to assess the impact of the present reward and the future separately since higher present reward may not necessarily lead to the best expected return. The equation can be thought of as a one-step look ahead. That is, the value of state s is obtained by averaging over all possible states transitions one step ahead. Basically, $v(s')$ is the value function of all possible transitions from *s* to *s'*.

Bellman Equation: $v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] = \mathcal{R}_s + \gamma \sum_{s' \in S} \mathcal{P}_{ss'} v(s')$

Where $\mathcal{P}$ is the state transition probability matrix of S_{t+1} being equal to *s'* conditioned to the fact that S_t is equal to *s*.

A Markov decision process is a Markov reward process with decisions (actions). Hence, a Markov Decision Process will include actions in its tuple (S, A, $\mathcal{P}$, $\mathcal{R}$, γ). So, with this addition, the transition probability matrix will also be dependent on the action taken.

It is now useful to better define a policy. A policy π is a probability distribution over actions given states.

Policy: $\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$

The action-value function *qπ(s, a)* is the expected return starting from state *s*, taking action *a*, and then following policy *π*

Action-Value Function: $q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$

And this allows us to define the Bellman Optimality equation where the optimal action-value function q_* is the maximum action-value function over all policies and the superscript a refers to taking action a. This equation is non-linear and it has no closed from solution so it is used iteratively to update the values of the state-action pairs, ultimately converging to the optimal values that maximize the expected cumulative reward for each state-action pair. It forms the basis for several reinforcement learning algorithms, including value iteration, policy iteration and Q-learning. (23) (2)

Bellman Optimality Equation: $q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$

3.1.4 Dynamic Programming, Policy iteration and Value iteration

Dynamic Programming (24) can be defined as a technique to solve complex problems by dividing them into simpler subproblems. For a problem to be a dynamic programming problem, it needs to have an optimal substructure and overlapping subproblems; two conditions that a Markov decision process has. Dynamic programming can be used both to predict or control, but it is needed to have full knowledge of the Markov Decision Process. In this case, predicting means obtaining the value function and control obtaining the optimal policy.

A policy can be improved using dynamic programming by policy or value iteration. The main idea of policy iteration is to iteratively improve the policy based on the learned value function until an optimal policy is obtained. This process combines both policy estimation and policy improvement (updating the policy based on value estimates) in a cyclic manner. The convergence of this iterative process is guaranteed. Value iteration is another method to find the optimal policy, but here the policy is implicit. It consists in progressively updating the value function using the bellman equation for all states at each iteration. (25) (2)

3.1.5 Model-Free prediction and control, Monte-Carlo and Temporal Difference Learning

Model free is when instead the Markov Decision Process is not known. In other words, it is learning without building a model of the world or fully knowing the dynamics of the environment. So the learner has no explicit knowledge of state transitions, and only learns from its own experience. While, if there is enough knowledge of the environment's dynamics, first a model of the environment can be built and then planning can be used as described before in value iteration and policy iteration.

Monte Carlo methods learn directly from complete episodes of experience and it uses the very simple idea that the expected value is equal to the mean return. To evaluate the policy, we first find the value function by using the empirical mean instead of the expected return and then we estimate the value as the mean return. The convergence is ensured by the law of large numbers.

Temporal Difference learning is different because it learns from incomplete episodes instead. It basically estimates the value based on estimated returns, therefore updates are a guess towards a guess. This methods learns progressively online while the episode is still one step after another step.

To sum up, the main differences in results between those two methods are that Monte Carlo has high variance, no bias, very good convergence and it is very simple while Temporal Difference tends to be more efficient, has low variance and some bias.

Regarding model-free control, it can be on-policy if the policy is learned while the algorithm is sampling experience from the policy itself or off-line if experience is experience from another policy. Concerning online Monte Carlo Control, there is again a policy iteration cyclically followed by a policy improvement that is done with a e-greedy method maximizing the q value. Indeed, while policy improvement over the value function requires knowledge of the model of the Markov decision process, the improvement over *Q(s,a)* is model-free. The idea in epsilon greedy exploration is to choose greedily with a high probability but to still explore (pick a random action) with a low probability. Q-learning is instead when learning is done off-policy with action-values *Q*. Moreover, there are two policies, one, the behavior policy, is used to pick the first action that leads to state *s'* and it is e-greedy, while, the second one is called target policy and is greedy and it is used to take the second action from state *s'*. The *Q(s,a)* is updated towards the value of the second action. Basically, Q-Learning uses an exploratory policy to find the optimal policy. (26) (2)

3.1.6 Value function Approximation, Experience Replay, policy gradient

In the previous parts, we considered the value function as a lookup table having one entry for each state or for each action-pair. However, for many problems, Markov decision processes have many states and this is not possible. A possible solution that turned out to be very powerful is the use of function approximation. In particular, neural networks trained with stochastic gradient descent showed great performance at this task, and, as explained before, this enables to exploit the power of deep neural network and create Deep Reinforcement Learning. One of the first example of this was the DQN (19) algorithm that exploits this function approximation using the Q-learning logic described before together with other improvements such as experience replay. (29)

Experience replay refers to the way experience is fed to the algorithms, in fact with the experience replay buffer, experience is not collected sequentially anymore, but it is first collected and then sampled stochastically from the buffer and used by the algorithm in a second moment. Rather than executing Q-learning on state/action pairs as they happen in simulation or real-world experience, the discovered data are conserved in the replay memory independently of the episode in the form of *(state, action, reward, next_state).* It's essential to note that it contains raw data intended for action-value calculations and not associated values. The learning phase is then logically distinct from the phase of gaining experience, relying on randomly sampling from this table. It is crucial to separate the two processes of acting and learning as enhancing the policy will result in different behavior that should explore actions closer to optimal ones, and those are the ones learning should be derived from. Random minibatches of experience are sampled for learning, and the learning process is conducted off-policy. The advantages can be summed up as:

- Reduction of autocorrelation, which can result in unstable training, by transforming the problem into one that resembles a supervised learning scenario.
- Better converge properties since the data will be more independently and identically distributed
- Higher data efficiency since we can reuse the sample stored in the replay buffer multiple times extracting it with a different order

Coming back to DQN, its high level logic is to gather a random batch training sample from the experience replay buffer, that then flows both in the Q network used to predict Q values and in the target network that predicts the target Q values that are then optimized with a Loss function and backpropagation. However, here there is a fundamental detail, indeed, the training Q network is optimized at each iteration while the Target Network remains fixed and is optimized only once every T steps by copying the train Q networks weights. This process ensures a balance between stability and adaptability during training. As a matter of fact, this process helps to stabilize the learning process and to improve the training efficiency also reducing correlation between consecutive samples consequently reducing overfitting. (29) The DQN pseudocode taken from (19) is reported in the annex.

The fact that the buffer is sampled randomly is important, indeed, learning from consecutive samples is inefficient due to the correlations between them, while by retrieving randomly, variance is reduced. Moreover, a sample can be reused improving data efficiency. (27)

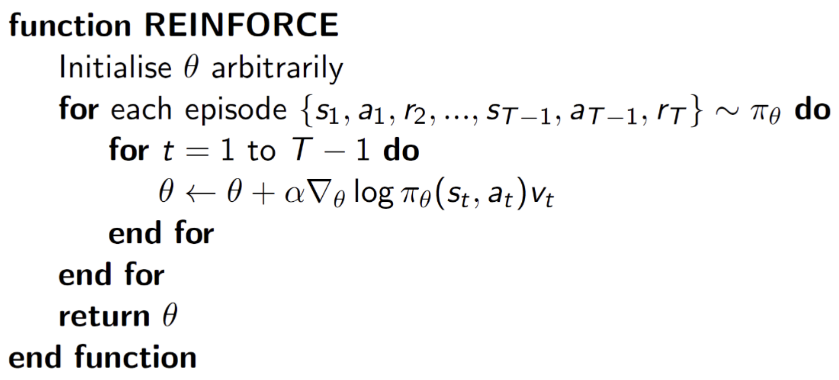
As we approximated the value function, also the policy can be obtained using function approximation with neural networks for model-free reinforcement learning. Policy gradient algorithms ensure better converge since, if the action space has a lot of dimensions, updates of the value function can be slow. The policy is updated using gradient descent on the policy objective function.

The policy gradient theorem reported below asserts it is possible to obtain optimal updates to the policy gradient, utilizing the long-term Q value rather than relying solely on the one-step instantaneous reward.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)Q^{\pi_\theta}(s,a)]$$

Where $\pi_\theta(s,a)$ is any differentiable policy and J is the policy objective function

This allows us to introduce one of the most famous policy gradient algorithms Reinforce, below the pseudocode is reported:



```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

Figure 6: REINFORCE pseudocode taken from slide 21 of (28)

Where v is used as an unbiased sample of the long-term Q value. The policy parameter is updated through Monte Carlo method or, in other words, by using random samples.

In a policy gradient approach, the batch is composed of trajectories where actions are randomly sampled at each time step. Trajectories are rolled out until the desired number of time-steps is reached, and value functions are computed to estimate the gradient using the log-probability score function. The policy gradient is then determined as the average of these gradients. This differs from the previously described experience replay methods, where transitions are stored and randomly sampled from a buffer. In this case, there is no empirical estimate for the value of the policy since it only samples transitions. Therefore, learning involves bootstrapping to estimate the value function and using a derivative with the chain rule to maximize the value estimate. (28) (2)

3.1.7 Actor-Critics Algorithms

As already mentioned before, we can categorize Reinforcement Learning algorithms into two main types: value-based (learn a value function and the policy is only implicit such as e-greedy) and policy-based (there is no value function and the policy is learnt directly). However, the current state-of-the-art algorithms employs a third technique that is based on both methods: actor-critics algorithms. (2)

Indeed, the fact that updates in Reinforce are done through random samples, as explained before, adds high variability in log probabilities and cumulative reward values thus introducing more noise in gradients leading to unstable learning and potential skewing of the policy distribution in a suboptimal direction. Alongside the issue of high gradient variance, another challenge with policy gradients arises when the total reward is zero. The core concept of policy gradients involves increasing the probabilities of positive actions and decreasing those for negative actions in the policy distribution. However, when trajectories result in a null cumulative reward, both positive and negative actions are not efficiently exploited. (30)

In order to address this issues, Actor-Critics methods add a critic components that has the task of estimating the action-value function. Therefore, two different sets of parameters are used for the actor and the critic, where the actor updates the policy parameters in the direction indicated by the crititc. In other words, the critic address the problem of policy evaluation for the current parameters. The pseudocode of Actor-Critics methods in general taken from (28) is available in the annex

Moreover, another improvement introduced that also reduced variance is the addition of a baseline function. This function is subtracted from the policy gradient without changing expectation. A suitable baseline could be the state value function. Then the concept of advantage function is introduced with the following formulation:

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

This allows us to reformulate the policy gradient theorem as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a) A^{\pi_\theta}(s,a)]$$

To estimate the advantage we can have the critic estimate both $VV^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$. Otherwise, The advantage function can also be represented using the TD error and in this case only one set of critic parameters would be required. The two formulas above would respectively become:

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

and:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)\delta^{\pi_\theta}]$$

(28)

A visual schema of the Actor-critics algorithm is proposed below:

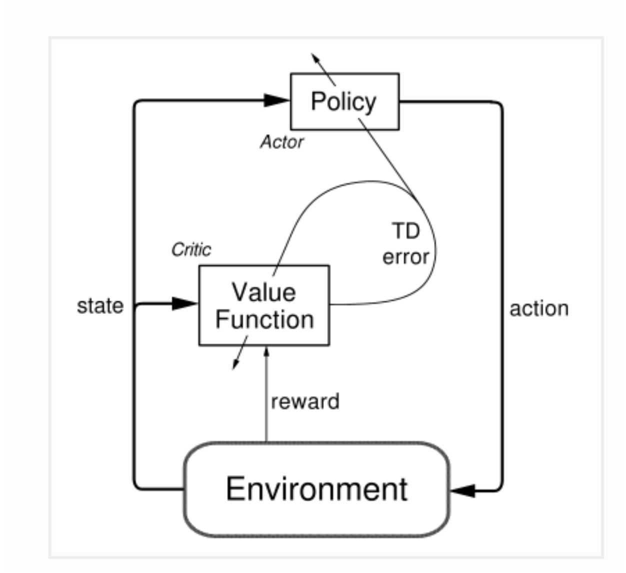


Figure 7: Actor-Critic schema taken from (2)

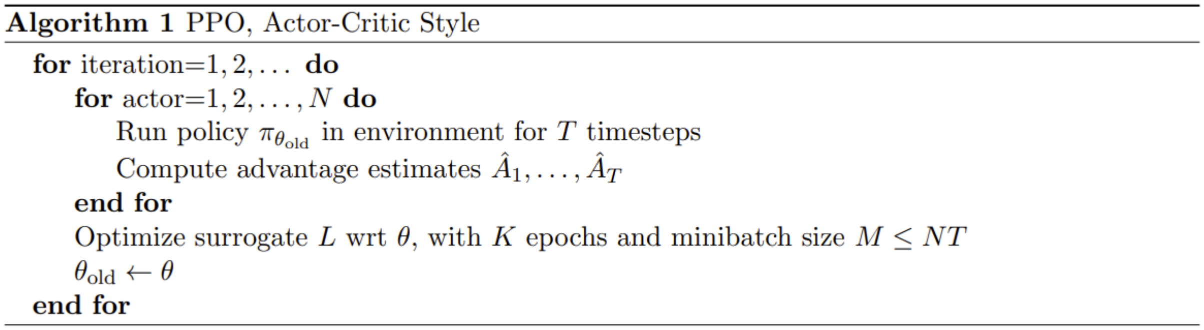
3.1.8 State-of-the-Art RL Algorithms

In this section, the most successful and still used Deep Reinforcement Learning algorithms are listed and explained.

- **Advantage Actor Critic (A2C)** and **Asynchronous Advantage Actor Critic (A3C)** were both released in 2016 by Google DeepMind in the paper (31)**.** Those are the first Deep Learning successful algorithms implementing the Actor-Critic method described in the previous section 3.1.7. This with the addition of a Critic component that has the task of estimating the Advantage values instead of the Q values or action-value function. This is possible by introducing a baseline function and by and estimating the advantage with respect to it reducing the high variance typical of policy networks. When A3C became public, it had a significant effect on the research community because of it simplicity, stability and performance. The “Asynchronous” in A3C refers to the fact that it consists in many agents with their own network interacting in parallel with copies of the environment all updating a global network at regular intervals, but not at the same time. After experimenting with A3C, the authors understood that also by just having different versions of the environment running in parallel synchronously they could get the same results with a much simpler model and this led to A2C. (31) (32)

- **Deep Deterministic Policy Gradient (DDPG)** (33) was developed by Google in 2016, it is also an Actor-critic algorithm, but focused on operating in continuous action spaces domains based on deterministic policy gradient method. When dealing with continuous action spaces, it is more difficult to exhaustively evaluate the space and the optimization is therefore more complex than with discrete actions. The networks are similar to the already described Advantage Actor-Critic method (section 3.1.7), however, the actor maps states to actions straight skipping the first usual policy output that is a probability distribution over a discrete action space. This algorithm exploits a replay buffer and experience replay. (refer to section 3.1.6). Therefore, the policy is updated in an off-policy manner with batches of samples. Exploration is also different in the context of continuous action spaces, indeed, instead of selecting a random action from time to time, some noise is added to the action output. In DDPG, the authors use Ornstein-Uhlenbeck Process (34) to add noise. For more details on the DDPG functioning, the pseudocode is provided in the annex. (33) (35)

- **Proximal Policy Optimization (PPO)** (36) was published by OpenAI in 2017 and builds on a previous algorithm called Trust Region Policy Optimization (37) keeping its main benefits with a much simpler implementation while showing better sample complexity and data efficiency. It belongs to the policy gradient methods that, as explained in section 3.1.6, cyclically interacts with the environment to sample data and optimizes a surrogate objective function through stochastic gradient ascent. The goal of this two algorithms is to take the biggest possible improvement step on a policy by using the available data, without stepping too far causing convergence issues and performance instability. To achieve this, TRPO adds a KL divergence constraint to enable a trust-region for the optimization process thus making sure the optimization is not too unstable and successive updates are not too far from the previous ones. Consequently, the new updated policy doesn't deviate too much from the previous one. PPO achieves the same results with a much simpler technique by basically adding a penalty to the objective function. This is obtained calculating a clipped value by truncating the policy ratio in a range and then making the objective function select the minimum between the actual value and the clipped value. Moreover, also the objective function is different since it performs multiple epochs of minibatch updates while standard policy gradient updates the policy once per sample. As already mentioned, those two enhancements lead to better stability and data efficiency. (36) (38) Below, the pseudocode is reported:



Algorithm 1 PPO, Actor-Critic Style

```
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 8: Proximal policy Optimization pseudocode taken from (36)

- **Twin Delayed Deterministic Policy Gradient (TD3)** (39) paper was published in 2018. TD3 is an Actor-Critic algorithm that addresses the main issues of DDPG such as unpredictable behavior and difficult hyperparameters tuning. This model utilizes two critic networks instead of only one (twin) and it performs delayed updates of the actor (delayed) and lastly it adds action noise regularization. The two critics are used by making the algorithm pick the lower of the two Q values produced by the critics creating an underestimation bias resulting in a more stable approximation and therefore in a more stable algorithm. Similarly to the considerations made for delayed target network weights updates in sections 3.1.5 and 3.1.6, in this case, the delayed update is applied to the actor network again for a more stable and efficient training. Noise regularization refers instead to the addition of clipped noise to actions when calculating the targets giving more importance to more robust actions. For more datils the pseudocode is reported in the annex. (39) (40)

- **Soft Actor critic (SAC)** (41) was published in 2018 by researchers at Berkeley University. It lays in between stochastic policy optimization (on-policy) and DDPG-based approaches (off-policy) and aims at improving sample efficiency and convergence. Indeed, SAC is off-policy and Actor-Critic, but it shows much better stability compared to the other previously-mentioned off-policy algorithms. The main feature of SAC is entropy regularization where the policy is trained to maximize a trade-off between expected return and entropy, that is closely related with the exploration-exploitation trade-off that was previously explained in section 3.1.2. In other words, the aim is not only to maximize the reward, but also to increase as much as possible the entropy of the policy. The agent has to succeed at the task while acting as randomly as possible. For more details, the pseudocode is provided in the annex. (41) (42)

3.2 Time Series data modelling

In order to model sequences effectively, we require a dedicated learning framework that can:

- Handle variable-length sequences.
- Preserve sequence order.
- Capture long-term dependencies instead of truncating input data prematurely.
- Enable comprehension of each word based on the context of previous words.
- Provide the network with the capability to search for a specific feature throughout the entire sequence, not limiting it to a particular region.
- Share parameters across the sequence to avoid redundance when learning.

The most suitable and extensively studied domain for introducing sequence modeling is natural language processing and this is the reason why some references to this domain may be found. However, to give a straightforward general definition of a sequence modelling network, such a network aims at predicting an output sequence from an input sequence, using an appropriate loss function.

3.2.1 Main sequence modelling architectures

In this first section, the main sequence modelling architectures that anticipated and therefore led to the creation of the transformer are introduced and briefly explained.

3.2.1.1 Recurrent Neural Networks

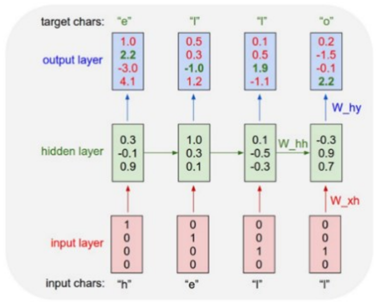


Figure 9: RNN schema taken from (43)

Unlike traditional neural networks, Recurrent Neural Networks present link forming directed cycles, enabling them to create and maintain a hidden state that embeds information about previous inputs of the sequence. The hidden state makes it possible to model temporal dynamics and process sequential data, that explains their suitability for tasks like Natural Language Processing (NLP), speech-to-text, and time series analysis.

The fundamental concept of RNN is its recursive structure, which entails several key aspects:

- The size of inputs and outputs are variable.
- There exists a hidden state that represents past knowledge. The previous hidden state, along with the current step's input, is utilized to calculate the current hidden state, which is then used to determine the output for the current step.
- The same activation function is repeated all over the architecture with the same configuration.

It's worth pointing out that the fact that all the parameters are shared, especially the ones of the hidden state, can become an issue as the network length increases. As a matter of fact, having just one memory element across the entire network can lead to information overload and difficulty to remember distant past events. Consequently, RNNs show reduced effectiveness when the distance between relevant information and the point where it is needed becomes substantial. This inefficiency arises because information is transmitted at each step, and, as the chain lengthens, there is a higher chance of information loss along the sequence. This is known as the vanishing gradient problem of RNNs. (44) (43)

3.2.1.2 Long Short Term Memory

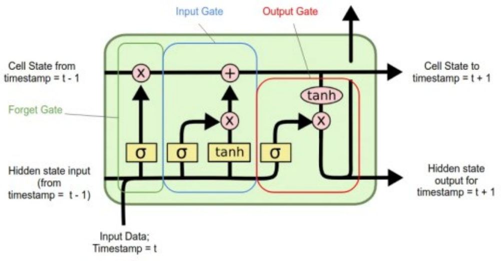

Input Gate
Output Gate
Cell State from timestamp = t - 1
Forget Gate
Hidden state input (from timestamp = t - 1)
Input Data; Timestamp = t
Cell State to timestamp = t + 1
Hidden state output for timestamp = t + 1
σ
σ
tanh
σ
tanh

Figure 10: LSTM schema taken from (45)

Long Short Term Memory networks (46) improved the performance of RNNs by addressing their main problems, and still keeps its popularity twenty years after its invention. The main issue with RNN, as already mentioned, is that they struggles to effectively retain past information due to a limited set of parameters. This issue causes information overload.

With sequential data, such as chats or conversations, multiple dimensions need to be considered, including what to emphasize, output, and forget. LSTM addresses these challenges by incorporating Input, Output, and Forget Gates. These gates combined effect, along with the Sigmoid function, filter and memorize relevant incoming information. The different gates' function can be summarized as following:

- Input Gate: Determines what information to remember, excluding less useful elements like common words.
- Output Gate: Specifies what information should be stored in the hidden state but omitted from the output.
- Forget Gate: Manages what information to forget, especially when the context changes.

The Sigmoid function, with its output ranging between 0 and 1, efficiently represents the percentage of information for input, output, and forget.

LSTMs present issues similar to RNNs. Indeed, keeping the context for the words positioned far from the current word being processed is problematic also for this kind of network. We can therefore point out again that, when dealing with long sequences, distant information is not well kept and exploited. Moreover, it would be more effective to have separate modelling mechanisms for short and long dependencies that in this case are not present. Eventually, parallel processing is not possible due to the inherent dependent nature of sequential data. (45) (46)

3.2.1.3 Convolutional neural networks

Although Convlution Neural Networks have been mainly studied for computer vision, they are relevant since some of their features are in between RNN's and LTSM's weaknesses and the self-attention mechanism that addressed those weaknesses explained later. As a matter of fact, they are more easily parallelizable since each entry of the input can be processed at simultaneously and does not depend on the previous entries. Moreover convolution allows to detect local dependencies, but this is not enough for many of the dependencies that we can find in sequential data. (47)

A quite successful model was built by exploiting the positive aspects of both CNNs and LSTMs called ConvLSTM. (48) The architecture of a ConvLSTM typically involves convolutional layers applied to the input data, followed by the integration of LSTM cells. This allows the model to capture both spatial features through convolutions and long-term dependencies through the LSTM memory cells. Therefore, ConvLSTM is a powerful architecture for tasks that involve sequences of data with spatial dependencies, offering the benefits of both convolutional and LSTM networks. However, the time dependencies are still handled by LSTM in this case so the same limitations remains on the time dependencies, indeed, in many fields the transformer substituted the previously unbeaten convLSTM.

3.2.2 Transformer-based architectures

In this section, the transformer architecture and its main components will be first explained to then introduce the transformerXL and the GTrXL architectures.

3.2.2.1 The transformer architecture

The Transformer architecture, (49) initially proposed in the natural language processing domain, has proven to be highly versatile and has become the foundation for various state-of-the-art models initially in NLP, but then also in other domains such as computer vision. The transformer addresses multiple limitations of previous models:

- Transformers allow for parallel processing of sequence elements, making them more efficient. The self-attention mechanism enables capturing relationships between all positions in a sequence simultaneously. There is a direct flow of information for each input and this is probably the key strength compared to CNNs, LSTMs and RNNs.

- Again thanks to self-attention, long-term dependencies are captured more efficiently than previous models.
- It is more parameter-efficient.
- Attention presents better explainability with respect to RNN and LSTMs.
- It has an explicit way to model hierarchy and therefore it is very scalable.

The Transformer architecture consists of an encoder, a decoder, and connections linking them. The encoder is a stack of identical encoders, and the decoder is a stack of an equal number of decoders. The decoding component is a stack of decoders of the same number. Each encoder comprises two sub-layers. The encoder's inputs initially passes through a self-attention layer, enabling it to consider other words in the input sentence while encoding a specific word. The outputs of this layer are then fed to a feed-forward neural network. The decoder includes both those layers, with an attention layer in between that helps the decoder focus on important parts of the input sentence. (50) (49)

In the next subchapters, some of the main concepts behind the transformer architecture will be explained.

3.2.2.2 Self-attention and Multi-head attention

To grasp the meaning of attention, it is useful to do a parallelism with retrieval systems and human attention. We will see that the attention mechanism is based on three vector or matrices K,Q,V. If we think of the attention operation as a retrieval process, we will have a query that we have to associate to some existing keys. Therefore, we will compute the matches and associate K and Q with the best matching ones to get a final value (result of the retrieval process). To make a parallelism with human attention, if the query corresponds to our will or voluntary signals, the key to sensory inputs of the environment and the value to the actual content or effect of what we are looking for. Indeed, queries and keys are multiplied to get an attention score that tells us if there is a good match and then this is combined with the value to get the final attention output. In the same way, the attention mechanism matches our voluntary signals with the sensory inputs of the environment to make a selection and get a final outcome. (that depends on the selected value)

For instance, consider an input sentence we need to translate: "The professor didn't receive the student because he was too busy." What is "he" referring to? The professor or the student? This might seem straightforward, but it is challenging for an algorithm. While the model encounters the word "he," self-attention comes into play, enabling it to correlate "he" with "professor". Throughout the processing of each word, self-attention empowers the model to focus on different positions in the input sequence, seeking contextual clues to improve and refine the encoding of a specific word.

Self-attention is calculated with the following steps:

- First of all, for each input element of the sequence, a Query vector, a Key vector, and a Value vector are created of the same dimension d_k. They're useful representation s that are also in line with the attention logic described before. These vectors are obtained multiplying the initial embedding by three matrices that are trained during the training process.
- The next step is to calculate a score for each entry of the input sequence against each other entry. The score represents the connection with or the focus on the other entries of the sequence with respect to that entry. This is obtained with the dot product of the query and key vectors.
- Then a division and softmax operation are done to the score for more stability and normalization. The score obtained at this point indicates how much each word will be expressed at this position.
- Finally each value vector is multiplied by this softmax score and then this weighted value vectors are summed and this is the output of the self-attention layer. This calculation is in practice done with matrices as showed in the formula below

$$Attention\ (Q,K,V) = softmax\ \left(\frac{QK^T}{\sqrt{d_k}}\right)$$

The transformer paper enhances the self-attention by creating a mechanism called "multi-headed" attention. Basically, it consists in doing the same procedure for the self-attention, but eight different times with different weight matrices with random initialization. Then to condense it back to one matrix at the end before the feed-forward layer, they are concatenated and multiplied by a fourth matrix Wo.

This improves the performance of the attention layer by helping the model to focus on different entries of the input vector and by allowing the attention layer to create different representations. Indeed, due to the random initialization, each set of vectors may focus on a different feature or representation. (50) (49)

$$MultiHead(Q,K,V) = Concat(head_i, \dots, head_h) * W^O$$

$$\text{where } head_i = Attention\ \left(QW_i^Q, KW_i^K, VW_i^V\right)$$

3.2.2.3 Positional encoding

The model as analyzed until now doesn't take into account the order of the words in the sequence. As explained before, this is very important in sequence modelling. Therefore, the authors of the transformer paper decided to add a vector to each input embedding to embed position information. These vectors adhere to a distinct pattern that the model learns, aiding it in identifying the position of each vector's entry, or the distance between distinct entries in the sequence. The underlying concept is that by adding these values into the embeddings incorporates relevant information on distances between the embedding vectors, both in their projected Q,K,V vectors and during dot-product attention.

The formula for positional encoding is described in the paper and uses sine and cosine functions at different frequencies.

$$PE_{(pos,2i)} = sin\left(pos\ /10000^{2i/d_{model}}\right)$$
$$PE_{(pos\ ,2i+1)} = cos\left(\ pos\ /10000^{2i/d_{model}}\right)$$

Were p *pos* is the position and i is the dimension of the embedding.

There are alternative approaches for positional encoding. However, this method offers the advantage of adaptability when handling sequences of unknown lengths, such as when our trained model has to translate a sentence longer than any in the training set. (50) (49)

3.2.2.4 Encoder-decoder architecture

As already mentioned, the transformer is based on a encoder-decoder architecture. To recap, the encoder layers are composed of two sublayers: self-attention and position feed-forward layer. The decoder layers are instead composed of three sublayers: self-attention, encoder-decoder attention, and a position feed-forward layer. The parts described so far are used as they are in the encoder. As a matter of fact, the encoder first processes the input sequence and then its first output is transformed into the attention vectors K and V. These vectors are then fed to the decoder in the encoder-decoder attention layer which allows the decoder to focus on the right part of the input sequence. This step is repeated multiple times. When the output is completed, the output of every step is fed to the bottom decoder of the next time step. Also in this phase, position encoding is added to the decoder input to indicate the position of each entry of the sequence. The decoder's self-attention layers are slightly different with respect to the encoder. Indeed, they are restricted to attending only to preceding positions in the output sequence through a masking operation of future positions before the softmax step. The Encoder-

Decoder Attention layer is similar to multiheaded self-attention, however, it creates its Queries matrix from the layer below it and retrieves the Keys and Values matrix from the output of the encoder stack. (50) (49)

3.2.2.5 TransformerXL

The authors aim with the Transformer-XL (51) paper is to overcome the limitations of fixed-length contexts. The idea is to integrate the concept of recurrence into the deep self-attention network. Indeed, the hidden states obtained in previous segments are exploited instead of recalculating them from scratch for each new segment. Those hidden segments act as memory for the current segment thus establishing a recurrent connection among different segments. This allows to model longer-term dependencies since information can propagate through the connections. At the same time, it addresses the challenge of context fragmentation by transferring information from the previous segment

Moreover, the authors define a new form of positional encoding that is relative, that, with respect to absolute ones, facilitate to reuse the state without creating temporal confusion. Therefore, this formulation is straightforward yet more effective, extending its applicability to attention lengths beyond those encountered during training. These two techniques collectively provide a comprehensive solution, as either in isolation fails to address the constraints associated with fixed-length contexts. (51)

3.2.2.6 GTrXL

Gated Transformer XL (GTrXL) (4) is an innovative model designed for processing extensive time horizons of sequential information in reinforcement learning, starting from the advancements of the Transformer-XL architecture. Thus demonstrating that the standard transformer architecture is difficult to optimize as already noted in the supervised learning setting and becoming an even more pronounced issue when dealing with Reinforcement Learning. This architecture outperforms LSTMs in challenging memory environments, achieving state-of-the-art results.

The two main contributions are identity map reordering and gating layers. Identity map reordering involves relocating the layer normalization to the input stream of the submodules. Notably, this reordering enables an identity map from the input of the transformer at the first layer to the output of the transformer after the last layer. This is a substantial difference with respect to the canonical transformer where a series of layer normalization operations nonlinearly transform the state encoding. This modification facilitates faster and more effective convergence of the network to the final outcome The intuition behind it is that, in many environments, it is crucial to learn reactive behaviours before effectively utilizing memory-based ones. For instance, an agent needs to learn how to read before it can remember what it has read.

To improve performance and optimize stability, the authors also replace the additions operations of residual connections with a gating layer. The paper proposes various implementations of this layer, with the most successful one based on a GRU-type mechanism. Gated Identity Initialization is a concept where the authors argue that Identity Map Reordering aids policy optimization by initializing the agent close to a Markovian policy or value function. (4)

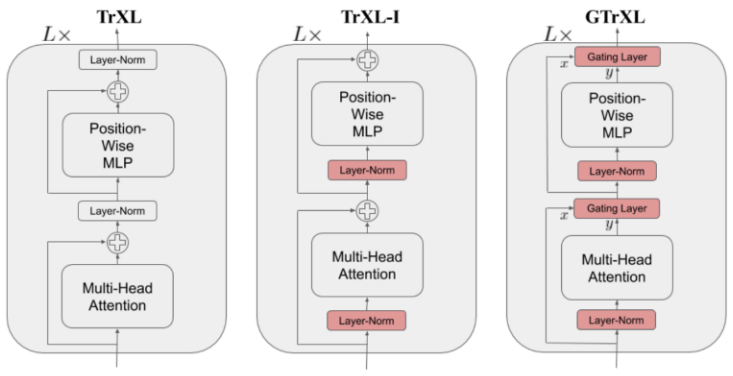


Figure11: GTrXL schema taken from (4)

3.3 Imitation Learning

Imitation Learning is a variant of Reinforcement Learning where the learning agent seeks to replicate human behavior. In contrast to conventional reinforcement learning methods, where an agent learns through trial and error within an environment (typically a Markov Decision Process) guided by a reward function, imitation learning involves the agent learning from a dataset of demonstrations performed by an expert. The objective is to mirror the expert's behavior in similar or identical situations. Imitation learning proves beneficial when it is more feasible for an expert to demonstrate the desired behavior than to specify a reward function that would produce the same behavior or to directly learn the policy. In some instances, direct access to the expert during training allows for additional demonstrations or evaluation queries. In order to provide a comprehensive understanding of Imitation Learning, we categorize it into two primary branches:

- Behavioral Cloning and its successive improvements, such as reward-weighted regression and advantage-weighted regression, where the agent learns a mapping from states to actions.
- Inverse Reinforcement Learning, where the agent deduces the underlying reward function from expert behavior.

(52) (53)

3.3.1 Behavioural Cloning

The most straightforward form of imitation learning is Behavioral Cloning, which allows to find the corresponding policy when it is not known by using expert demonstration in the form of state-action pairs in a supervised manner. One of the first examples of behavior cloning was ALVINN, (54) a vehicle that was able to map sensor inputs to driving decisions. This pioneering project, dates back to 1989, and it was probably also the first example of imitation learning in general.

It is relevant to point out that the resulting policy must be able to generalize to states that have never been encountered. Behavioral Cloning gives great results if the set of expert data that we have contains optimal high-quality solutions and the amount of data is sufficient to have a good amount of generalization and avoid overfitting and its simplicity and efficiency are a great advantage. Nonetheless, it is prone to failure in situations where the dataset includes a significant amount of random or task-irrelevant behavior, or when the acquired policy results in a trajectory distribution that diverges considerably from the dataset under examination. Moreover, another obstacle arises from the independent and identically distributed assumption, as supervised learning assumes for state-action pairs. Indeed, in a Markov Decision Process, an action in a given state triggers the next state, deviating from the distribution assumption. Consequently, errors in different states may accumulate, and a mistake by the agent can lead to an undefined behavior. (52) (53)

For simple behavioral cloning, we would be minimizing a Loss function like this one:

$$Loss\ (\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{t=1}^{H} log\ \pi_{\boldsymbol{\theta}}\left(\boldsymbol{a}_t^{(i)} \mid \boldsymbol{s}_t^{(i)}\right)$$

Where t is time, and i identifies the demonstration sample.

3.3.2 Offline or Batch Reinforcement Learning (Reward Weighted Regression)

With Behavioural cloning, the obtained policy does not perform better than observed demonstrations and for cases in which expert data samples led to different results, some better and some worst, we can have some improvements. As a matter of fact, an option is to weight samples based on the return or reward (by adding an environment) that each demonstration obtained. Then, the resulting policy should perform better than the observations. The updated loss function has just an additional demonstration reward term:

$$Loss\ (\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{t=1}^{H} log\ \pi_{\boldsymbol{\theta}}\left(\boldsymbol{a}_t^{(i)} \mid \boldsymbol{s}_t^{(i)}\right)R\left(\tau^{(i)}\right)$$

This formulation, since we introduced an environment and a reward, may seem similar to off-policy Reinforcement Learning. As explained in the image below, taken from (55)**,** the distinction lies in the fact that, in the traditional off-policy setting, the agent's experiences are added to a data buffer or replay buffer. Each new policy accumulates additional data, so this buffer comprises samples from all the different policies gradually constructed and improved over time. All this data is then utilized to train an updated new policy. In contrast, offline reinforcement learning relies on a dataset collected by some (potentially unknown) behavior policy. This dataset is collected once and remains unchanged during training, allowing the use of large previously collected datasets. The training process has no interaction with the Markov Decision Process (MDP), and the policy is only deployed after being fully trained.

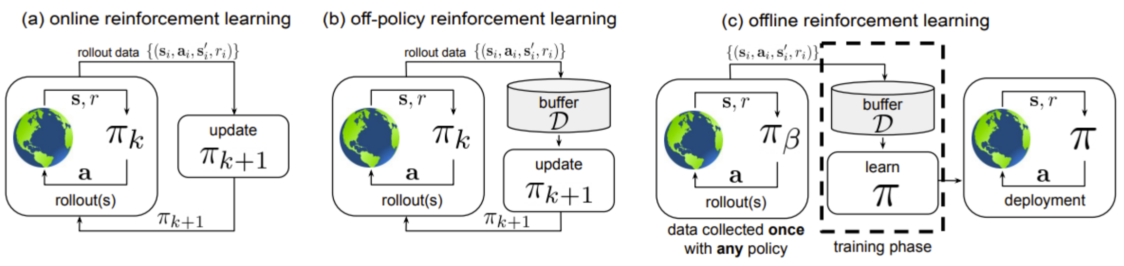

(a) online reinforcement learning
rollout data $\{(s_i, a_i, s'_i, r_i)\}$
s, r
π_k
a
rollout(s)
update
π_{k+1}
π_{k+1}
(b) off-policy reinforcement learning
rollout data $\{(s_i, a_i, s'_i, r_i)\}$
buffer $\mathcal{D}$
update π_{k+1}
(c) offline reinforcement learning
$\{(s_i, a_i, s'_i, r_i)\}$
π_β
learn π
deployment
data collected **once** with **any** policy
training phase

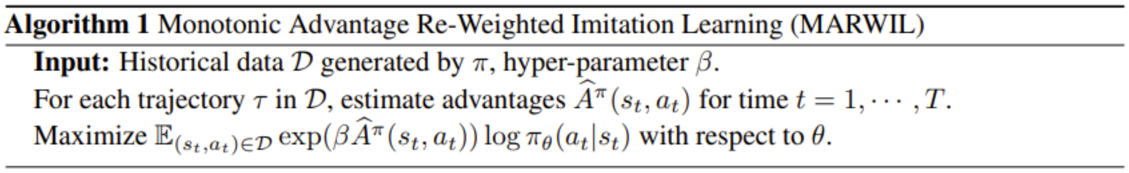
Figure 12: Comparison of online, off-policy and offline RL taken from (55)

The potential of a fully offline reinforcement learning framework is very interesting. Similarly to how supervised machine learning has led to the transformation of data into robust and generalizable pattern recognizers such as for image classification for instance; offline reinforcement learning, together with function approximation, has the potential to convert data into versatile decision-making engines. This implies that, with a sufficiently large datasets, this approach could be leveraged to develop successful decision-making policies. (52) (55)

3.3.3 Advantage Weighted Regression

In Marwil (Monotonic Advantage Re-Weighted Imitation Learning) (56) the advantage a specific action has is estimated from the cumulated discounted future reward and the value of the current state given by a value function estimated with a neural network. Then higher sample weight is given to the actions with a higher advantage value.

The pseudocode provided by the authors it's the following:



```
Algorithm 1 Monotonic Advantage Re-Weighted Imitation Learning (MARWIL)
  Input: Historical data $\mathcal{D}$ generated by $\pi$, hyper-parameter $\beta$.
  For each trajectory $\tau$ in $\mathcal{D}$, estimate advantages $\widehat{A}^{\pi}(s_t, a_t)$ for time $t = 1, \cdots, T$.
  Maximize $\mathbb{E}_{(s_t,a_t)\in\mathcal{D}} \exp(\beta \widehat{A}^{\pi}(s_t, a_t)) \log \pi_\theta(a_t|s_t)$ with respect to $\theta$.
```

Figure 13: MARWIL pseudocode taken from (56)

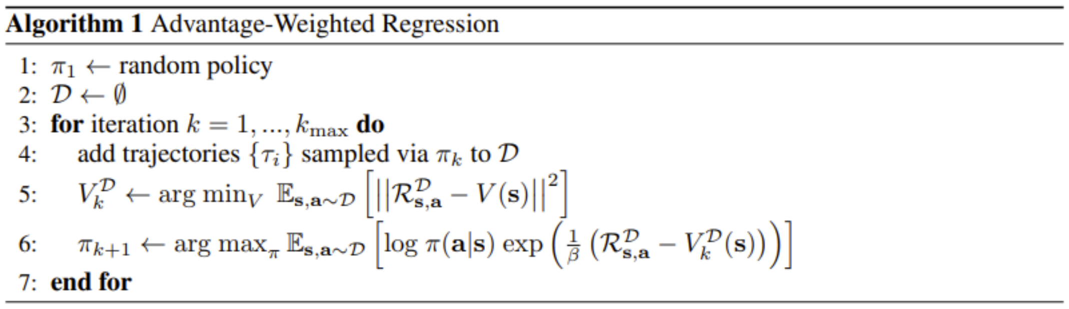
where β is a hyper-parameter specifying how much we want to differentiate actions with different advantage (how much influent is the advantage). And the advantage function $A(s_t, a_t) = \mathbb{E}_{\pi|s_t,a_t}(R_t - V^{\pi}(s_t))$ is estimated with a neural network.

Performing multiple iterations of a procedure similar to MARWIL, but introducing exploration at each iteration, we obtain Advantage Weighted Regression. In other words, after weighted regression is performed, then the obtained policy is used to generate new data with exploration and a new policy is obtained from this new data at each iteration. This allows to find better and better policies exploiting exploration.

This technique was introduced by (57), where the authors explain their aim is to develop a scalable reinforcement learning algorithm by exploiting standard supervised learning techniques as subroutines. The algorithm relies solely on uncomplicated and convergent maximum likelihood loss functions while also capitalizing on off-policy data. The algorithm performs two main steps similar to MARWIL at each iteration:

- regression onto target values for a value function.
- regression onto weighted target actions for the policy

The pseudocode provided by the paper is the following:



```
Algorithm 1 Advantage-Weighted Regression
1: $\pi_1 \leftarrow$ random policy
2: $\mathcal{D} \leftarrow \emptyset$
3: for iteration $k = 1, ..., k_{\max}$ do
4:     add trajectories $\{\tau_i\}$ sampled via $\pi_k$ to $\mathcal{D}$
5:     $V_k^{\mathcal{D}} \leftarrow \arg\min_V \ \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[ \left|\left| \mathcal{R}_{\mathbf{s},\mathbf{a}}^{\mathcal{D}} - V(\mathbf{s}) \right|\right|^2 \right]$
6:     $\pi_{k+1} \leftarrow \arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[ \log \pi(\mathbf{a}|\mathbf{s}) \exp\left( \frac{1}{\beta} \left( \mathcal{R}_{\mathbf{s},\mathbf{a}}^{\mathcal{D}} - V_k^{\mathcal{D}}(\mathbf{s}) \right) \right) \right]$
7: end for
```

Figure 14: Advantage Weighted Regression pseudocode taken from (57)

Where the reward is estimated using temporal Difference learning with the target value of the previous iteration as bootstrap. (52) (57)

3.3.4 Inverse Reinforcement Learning

Instead of directly learning actions, Inverse Reinforcement Learning (IRL) seeks to understand the underlying reward function that the expert is optimizing. This is based on the assumption that the reward function, and not the policy, provides a synthetic, solid, and transferable definition of the task. Once the reward function is modeled starting from optimal expert demonstrations, RL can be employed to learn the behavior policy. After identifying a suitable reward function, the issue mutates to finding a policy that can be solved with common RL methods. This procedure involves iteratively refining it by comparing the value of the currently optimal expert policy with some generated policies. While this approach offers better generalization to unseen states, it is usually more complex and computationally intensive.

The first formulation of Inverse Reinforcement Learning in the Markov decision process setting dates back to 2000 in the paper. (58) A challenge arises when simplifying a complex task into a reward function as a given policy may be optimal for various reward functions. Despite having actions from an expert, multiple reward functions may be candidates for what the expert aims to maximize. However, the objective of Inverse Reinforcement Learning is to discover a reward function that encapsulates meaningful task information and effectively distinguishes between desired and undesired policies. To address this, Ng and Russell frame Inverse Reinforcement Learning as an optimization problem, aiming to select a reward function for which the provided expert policy is optimal while maximizing certain essential properties under this constraint. (58) (59)

3.3.5 A Minimalist Approach to Offline Reinforcement Learning

The model proposed by this paper (62) has been used for the third variant implementation. (see paragraph 4.8) The paper tackles the common approach of constraining or regularizing the policy with actions from the dataset to create an offline reinforcement learning algorithm. Typically, making an RL algorithm function offline introduces additional complexity. The objective here is to enable a deep RL algorithm to operate offline with minimal adjustments and therefore avoiding as much as possible additional complexity. This is achieved by incorporating behavioral cloning into TD3 (see section 3.1.8 for more details on TD3), simply by introducing a regularization term to the policy of deterministic policy gradient. The amount of regularization is controlled by the hyperparameter alpha. The formulas below describe this regularization term addition.

$$\pi = \underset{\pi}{\operatorname{argmax}} \, \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,\pi(s))]$$

$$\pi = \underset{\pi}{\operatorname{argmax}} \, \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\lambda Q(s,\pi(s)) - (\pi(s)-a)^2\right]$$

Figure 15: Deterministic Pollicy Gradient for (62)

Moreover, in addition to the regularization term, another simple modification has been made by the authors by normalizing the states over the dataset, such that they have mean 0 and standard deviation 1. This improves the stability of the learned policy. The two changes that have been described above are the only two changes made with respect to the original TD3 algorithm. The model will be referred to as TD3-BC in the following sections.

3.3.6 Other Related Works

In this section, other imitation learning papers are briefly explained mentioning for which aspects they can be relevant. The last paper is an application to the portfolio optimization problem.

3.3.6.1 Conservative Q-learning for Offline Reinforcement Learning

This paper (60) is again in the field of offline reinforcement learning. The model's objective is to tackle the overestimation of values caused by the distributional shift between the dataset and the learned policy, particularly evident when training on intricate and multi-modal data distributions. CQL establishes a lower limit on the current policy's value and can seamlessly integrate into a policy learning process with theoretical improvement assurances. In practical terms, CQL enhances the standard Bellman error objective by introducing a straightforward Q-value regularizer. This regularization method is easy to implement as an extension to existing deep Q-learning frameworks.

3.3.6.2 Critic Regularized Regression

The Critic Regularized regression model (61) is a form of offline reinforcement learning. It simplifies offline policy optimization to a type of value-filtered regression, requiring minimal modifications to standard actor-critic methods. Consequently, it is straightforward to comprehend and implement. The model involves comparing the critic's Q-value prediction for an action in the trajectory with the Q-value of the action from the policy. If the critic's Q-value exceeds the policy's Q-value, the corresponding action is employed to train the policy; otherwise, the action is disregarded. This algorithm effectively filters out suboptimal actions, facilitating the learning of improved policies from data of lower quality.

3.3.6.3 Demonstration-regularized RL

The paper aims at theoretically quantifying the degree to which the additional information decreases the sample complexity of RL. Specifically, by investigating demonstration-regularized reinforcement learning, which utilizes expert demonstrations through KL-regularization for a policy acquired via behaviour cloning. (63)

3.3.6.4 Adaptive Quantitative Trading: An Imitative Deep Reinforcement Learning Approach

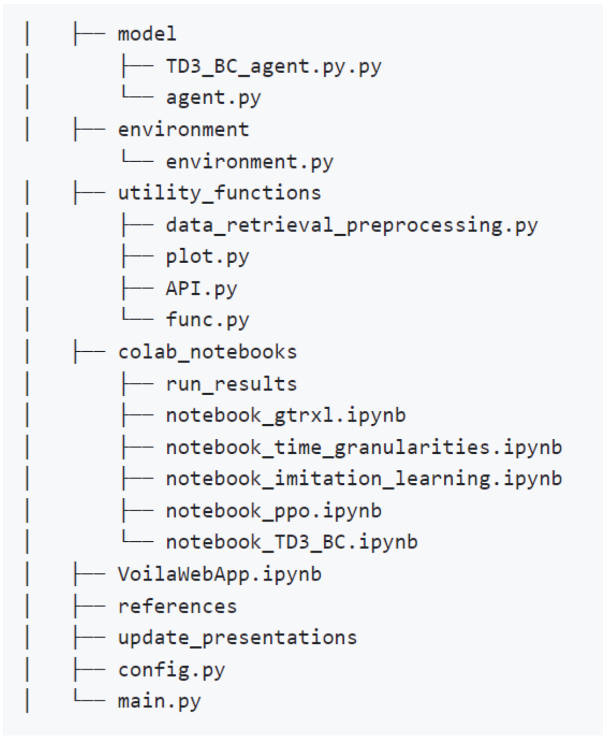
This paper (64) combines together Reinforcement Learning with Imitation Learning for the Portfolio Optimization problem. The base model used is Deep Deterministic Policy Gradient (see section 3.1.9) but in its recurrent extension form that is Recurrent Deterministic Policy Gradient. (RDPG) (65) The observations are constituted of Technical indicators from the dual thrust trading strategy, the prices and rewards that are probably the same information included in the state together with the account balance. The input data granularity is minute and the training set is composed of two years of data from 2016 to 2018, while the testing period is one year in 2019. The traded assets are futures on the top 300 Chinese stocks. Imitation learning is used in two different ways with a demonstration buffer containing dual thrust strategy demonstration episodes and behavioral cloning using prophetic expert actions created in hindsight. The obtained returns and Sharpe ratio seem very promising. This paper is interesting because it merges recurrent sequence modelling with a recurrent Reinforcement Learning model, with imitation learning, but there is no code available and some implementation details are not clear.

4 Implementation and contributions

In this Chapter, the implementation of the Deep Reinforcement Learning model and the GTrXL and imitation learning variants will be explained, leaving to chapter 6 the backtesting and results discussion.

4.1 Project code structure

The following schema allows the reader to get a better understanding of the repository structure and consequently of how the different sections of code are used together to produce the final result. The repository is available at this link.



```
|   ├── model
|       ├── TD3_BC_agent.py.py
|       └── agent.py
|   ├── environment
        └── environment.py
|   ├── utility_functions
|       ├── data_retrieval_preprocessing.py
|       ├── plot.py
|       ├── API.py
|       └── func.py
|   ├── colab_notebooks
|       ├── run_results
|       ├── notebook_gtrxl.ipynb
|       ├── notebook_time_granularities.ipynb
|       ├── notebook_imitation_learning.ipynb
|       ├── notebook_ppo.ipynb
|       └── notebook_TD3_BC.ipynb
|   ├── VoilaWebApp.ipynb
|   ├── references
|   ├── update_presentations
|   ├── config.py
|   └── main.py
```

Figure 16: Repository structure

- **Model**: contains the code related to the PPO agent therefore the interface with the Ray RLib library (see chapter 4.2.1), the models customizations and the inputs of the PPO model. For the TD3-BC variant Ray RLib is not used and this folder contains the full model Pytorch code.
- **Environment**: The environment code is separated since it is specific to the portfolio optimization application. It handles all the tasks related to calculating states, observations and rewards.

- **Utility functions**: This folder contains all the functions called by the main or the notebook for the tasks of data preprocessing, results manipulation, interface with the API for retrieving stock market data (Financial Modeling Prep) and more generic main functions removed from main.py for clarity.
- **Notebooks**: In this folder all the different notebooks are stored together with the run results data stored as notebook saved after a run.
- **VoilaWebApp**: This notebook is a simple Voila Web Application. Voila is a library that allows to easily turn a jupyter notebook into a WebApp avoiding more complex implementations.
- The folders **references** and **update_presentations** contain respectively the main papers used divided by relevance with a brief markdown comment and the power point presentations that have been created while working on the project throughout time.
- **Main.py** contains a function to make a train and test run from terminal without using the colab notebooks
- **Config.py** contains the main paramers to set for a run (training and testing periods, technical indicators used, API parameters, RL models parameters)

4.2 Programming Libraries used for the implementation

For the implementation of the model, multiples libraries have been used since, due to the complexity of the project and of Deep Reinforcement Learning state-of-the-art models, it was deemed as more effective to start from researcher's competitive implementations and modify it when necessary as it is common practice in the field. In particular, two libraries had more impact on this project:

- Ray RLib (66) algorithms were used for the implementation of the Agent with some modifications mostly for the contributions part.
- FinRL (67) was used as a starting point for the Environment, data preprocessing and results manipulation, but most of the code for this parts has been cleaned, restructured and rewritten.
- Other libraries were used as well for minor parts such as pyfolio for results manipulation and plotting, stockstats for technical indicators calculations.

4.2.1 Ray RLib

The ray project, (66) is an open source project carried out on a GitHub repository with 30k starts that provides a library with a lot of Artificial intelligence and Machine learning models coded in python. The focus of the library is on distributed runtime and scaling of AI workloads. The project was created by two students of the RISELab at Berkeley University and it is constantly (daily commits) maintained and advanced by a thousand of contributors. It is used by companies such as Uber, Spotify and Netflix and obtained publications in famous AI conferences such as NeurIPS. The aim of the project was to create a framework that integrates the diverse components required to deploy machine learning or deep learning algorithms effectively in practical scenarios. The framework can be divided in three layers as indicated by the official documentation where cloud refers to cloud deployment support and facilitation:

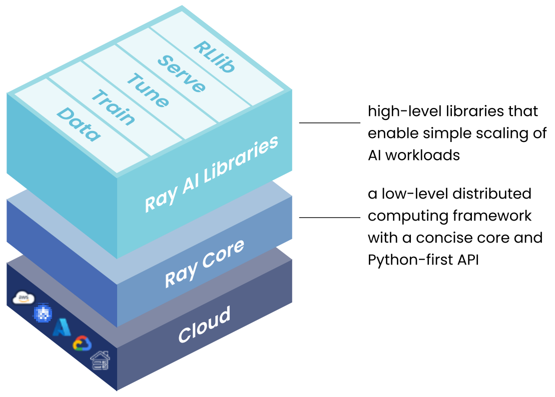

RLlib
Serve
Tune
Train
Data
Ray AI Libraries
Ray Core
Cloud
high-level libraries that enable simple scaling of AI workloads
a low-level distributed computing framework with a concise core and Python-first API

Figure 17: Ray RLlib high level structure taken from (66)

As shown by the picture above it is composed by five libraries:

- *Data*: handling data loading and transformation
- *Train:* enabling distributed, multi-core training with support for popular training libraries
- *Tune*: handling scalable hyperparameters optimization
- *Serve*: handling deployment of models for online inference.
- *RLlib*: library specific to Reinforcement Learning

RLib library is maintained by researchers in the field and used by companies and is really up to date implementing very quickly algorithms just after they are published in new papers. This reasons led to the choice of this framework for this thesis project. As mentioned above, two of the main goals of the library are to offer a wide variety of algorithms and scalability that is usually very important in Reinforcement Learning where a lot of data are needed. However, this adds much more complexity because the framework is built with a lot of abstractions to have as much as possible in common between different algorithms and to be able to run in complex and powerful environments with a lot of resources available. For instance the library also provides a built-in Server/Client setup that makes the library very intricated and full of content. The fact that a lot of contributors work on the project also ends up increasing code complexity.

It is worth mentioning that this, together with the fact that the documentation is limited, resulted in additional work for this thesis project where computing resources as well as stock market data are limited and therefore a simpler framework would have been enough and easier to modify. As a matter of fact, it has progressively been observed during this work that, while ray RLlib works very well when used as-it-is, it requires a lot of understanding of the underlying functioning with limited documentation when contributions and modifications to the models are needed.

RLlib can be used both with PyTorch and TensorFlow and it is divided into two main parts: Algorithms and Environments that respectively provide implementation for a lot of Reinforcement Learning models agents and many famous environments supported implementations. The environments features have not been used in this thesis project since there is no financial environment available. More details on how the algorithms have been used are available in section 4.5.

Ray Tune library (68) has also been used for hyperparameters tuning, more details are provided in section 4.7.

4.2.2 FinRL

FinRL (67) is an open source framework for financial Reinforcement Learning developed by the AI4Finance community that was created by some students and now is also supported by Columbia University a shown on their open finance foundation website. The repository gained a lot of popularity reaching 8.5k stars and about one hundred contributors. It is relevant to mention that the financial community is usually not open source and therefore it is really difficult to find quantitative finance publicly available material and this makes this project very atypical. The library managed to public papers to important conferences such as NeurIPS and ICAIF confirming the relevance of the work.

Another positive note is the fact that this projects merges a lot of different sources from data retrieval and APIs to agent's implementations and applications collecting a very good set of information for someone starting of with Deep reinforcement Learning for finance.

The picture below is reported to show more details on the previously mentioned variety:

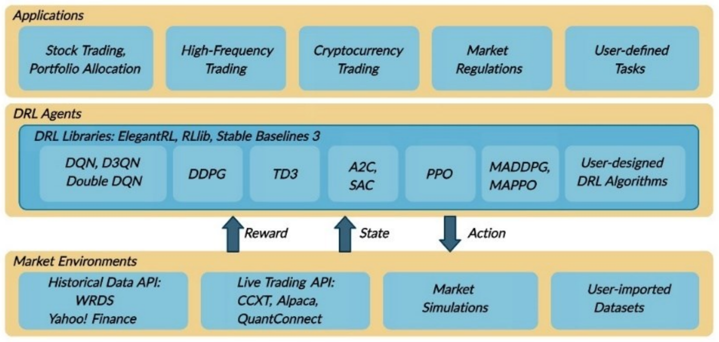


Figure 18: FinRL high level structure taken from (67)

However, digging deeper in it in the context of this thesis project, it was possible to notice that the material is often repeated in different libraries and publications and sometimes this, together with a limited documentation, makes it difficult to understand and use the provided tools. For this reason, the code was cleaned, restructured and corrected in many parts before using it for the thesis work and the library is not just used importing it as it is the case for Ray RLlib.

FinRL was used for this thesis as an initial implementation for the environment, the data preprocessing and results manipulation parts.

4.2.3 Other libraries

The other libraries used or that where studied because relevant to the project are the following:

- **Stockstats**: This library (12) is a wrapper library built on top of pandas DataFrame, aiming to offer instant access to a range of measures and technical indicators associated with stock prices just starting from OHLC prices. In section 2.3.1 more details on each single indicator used are given. It has 1.2k stars on github.

- **Pyfolio**: It is a Python library (69) with 5.3k stars on github designed for assessing the performance and risk of financial portfolios. Developed by Quantopian Inc., it seamlessly integrates with the Zipline open-source backtesting library. Quantopian is a platform enabling users to create, test, and implement algorithmic trading strategies using Python. Zipline is a python library allowing users to automate the execution of their trading strategies thus enabling users to define trading algorithms and conduct backtests using historical data. Pyfolio has been used for the portfolio positions, monthly and annual returns heatmaps and distributions.
- **PyPortfolioOpt**: It is a library (70) that incorporates portfolio optimization methods, encompassing classical efficient frontier techniques as well as recent advancements such as shrinkage, and experimental features like exponentially-weighted covariance matrices. It was used for testing the inclusion of Efficient Frontier (see section 2.2.2) weights in the agent's state.
- **Stable baselines 3**: This library (71) is another valid option to Ray RLlib for Deep Reinforcement model's implementations. It has 7.2k starts on the GitHub repository. It has been explored and tried before choosing Ray. It is a PyTorch version of OpenAI's stable baselines library. This library is very clean, easy to use and stable both with respect to Ray RLlib and stable baselines, but it doesn't include the most recent algorithms and variants and it is updated less often than Ray RLlib.

4.3 Input data and preprocessing

The stock market data has been initially retrieved with the Yahoo Finance API that is free and has no specific limits. However, recently the number of API calls was limited and therefore a differen API called Financial Modeling Prep (72) was used instead. Financial Modeling Prep was chosen because despite being way cheaper with respect to most APIs, it offers a wide variety of data: from price with any time granularity (minute to daily) to financial analysis and fundamental analysis indicators as well as stock news starting from 2017 and social sentiment. Moreover, both doing some basics checks on data quality and searching online information on its reliability, reviews were really good. Two variants of the code have been developed to experiment with different data granularites, indeed when dealing with smaller timeframes than daily, it was necessary to also specify time in the dates and adapt consequently the rest of the code.

Data preprocessing include some basic checks such as that, for each date, all stocks have the same number of non-null entries. It is important to mention that only twenty-five out of the thirty stocks (see annex for the list of stocks) of the Dow Jones Industrial Average index have been used since only those were active from 2004 to 2023 without being delisted. As a matter of fact, this time frame was chosen to have enough data for the model without using too old data that are not relevant anymore. Indeed, many market mechanisms and patterns change over time due to change in the environment such as, for instance, the increase in algorithmic trading and faster and wider spread of news information.

The technical indicators used in the state are the ones explained in section 2.3.1. The input values have been rescaled to a value in between zero and one since this allows the neural network to have better convergence properties. However, for the indicators that are not bounded and stationary because their maximum and minimum value depends on the price trend or drift (see section 2.1.2) including the moving averages and Bollinger Bands, also their stationary version, obtained by using log-returns instead of prices, has been used. As a matter of fact, the property of additivity of log-returns ensures that the indicator is stationary. This paper (73) investigates more in detail the topic of stationarity related to technical indicators. However, it is not a widely discussed topic since, usually, this indicators are used in practice and to visually identify trends and momentum on charts and therefore there is not a lot of literature on the topic. The state of the agent also contains information on current portfolio weights and available balance (cash that can be invested at next timestep without selling other stocks)

4.4 Environment

As already briefly mentioned the task of the environment code is to simulate the stock market using true prices observed in the past and calculating all sorts of external information the agents needs to receive as observations including prices, return, rewards, remaining balance and portfolio weights. There are two main methods to explain to give more details on the environment functioning: the *reset* and the *step* methods.

The *reset* method handles initialization of the state at the beginning of an episode. This process needs to be differentiated between training and testing since, in training, the initial balance has a random 10% variation and the initial weights are also assigned randomly to give stochasticity to the training. In other words, in this way, from one training iteration to another one, the agent starts from a different initial condition in terms of portfolio weights and investable amount while, of course, the stock prices will always be the same for the same date.

The *step* method receives the actions form the agent and starts performing the sell operation (selling not more than the currently owned amount since the model output is raw), calculates the updated investable amount after the sales and then performs the buy operations (again checking to buy not more than what it is possible to buy with the available cash). This operations need the current stock prices to be carried out. Moreover, the reward is calculated and then it is returned to the model for the next timestep along with new prices, and updated portfolio weights after sell and buy operation. The picture below gives a visual representation of the step method process:

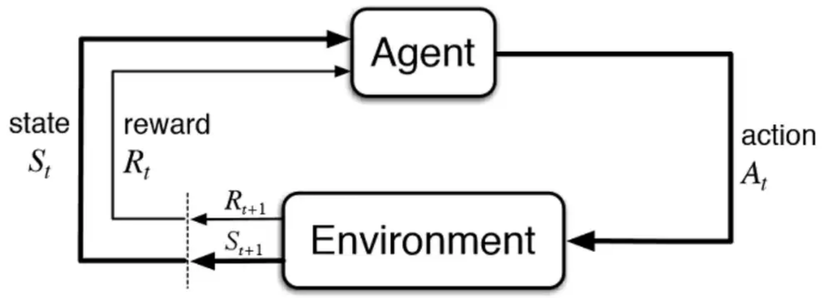


Figure 69: Agent's interaction with the Environment in RL taken from (2)

Where in this case the state is composed of the t+1 prices (and technical indicators calculated by the agent from the prices), the portfolio weights and the account balance.

4.5 Reinforcement Learning Agent

Among the different state-of-the-art models described in section 3.1.8, Proximal Policy Optimization (36) was chosen for this thesis project. This because although it is not the newest model, it is one of the most recent and the authors developed it focusing on data efficiency and sample complexity with a very simple trick avoiding complexity and even simplifying it with respect to TRPO and some other previous models. Moreover, before working on this project, there were opportunities to try and experiment with the different state-of-the-art models for the portfolio optimization problem with stable baselines 3 library (section 4.2.3) and Proximal Policy Optimization was found to be one of the most stable among the ones that were giving good results. Indeed, as will be discussed more in details later, one of the main issues in applying Reinforcement Learning to the Portfolio optimization problem, is the instability and low reproducibility of results of this kind of algorithms that is exacerbated by the limited amount of data available in the financial domain. In addition to that, being one of the most widely used and tested algorithms, many libraries provide additional support for experimenting with it. For instance, Ray RLlib provided an implementation of GTrXL (section 3.2.2.6) for Proximal Policy Otimization even if the paper is quite specific.

The main hyperparameters that were provided and tuned (many other parameters are also customizable) to the Proximal Policy Optimization implementation imported from Ray RLlib are reported below. Only some information is given now since more will be discussed in section 4.9 dealing with hyeparameters optimization

- **Entropy_coeff** determines the degree of exploration.
- **lr** that is the learning rate and it is used as usually by the optimizer for gradient descent.
- The **sgd_minibatch_size** parameters specifies how many data points are sampled at a time from the buffer.
- **Lambda** serves as a smoothing parameter, effectively reducing training variance and enhancing overall stability.
- Specifics on the **neural network model** kind and shape. Here it is possible to select a Multilayer Perceptron model that is a classic fully connected neural network and is the default, an LSTM (section 3.2.1.2) or even an attention model that is GTrXL. Moreover the specific parameters of the model can be specified. For instance, for the Multilayer Perceptron model and its shape can be defined as *model : {'fcnet_hiddens': [256, 256]}*.

In order to run the Proximal Policy Optimization model and obtain results three main steps need to be done:

1. Define the configuration object to be fed to the model indicating the model config parameters explained above plus any optional ones, the environment train and test objects containing the stock prices and technical indicators data and finally provide the number of iterations. Ray RLlib provides a default config file that needs to be imported and then modified.
2. Create a trainer object with this configurations and call its training method for the needed number of episodes on the training environment.

3. Eventually, use the trained algorithm object on the test environment and save return information together with any other needed information. To clarify, this is an example code for this third part:

```
while not done:
      action = trainer.compute_single_action(state)
      state, reward, done, _ = environment.step(action)

       total_value = (env_instance.amount
             + (environment.prices[environment.day] * environment.stocks).sum()
      )

      episode_total_value.append(total_value)
      episode_return = total_value / environment.initial_total_value
      episode_returns.append(episode_return)
```

The first two steps are different if Ray Tune (section 4.2.1) is used for hyperparameters optimization as it was the case for most runs in this thesis project. In that case, all the parameters are provided to Tune functions that then it takes care of all the trainings with different sets of hypeparameters and selects and outputs the best model. Then the final model obtained with Tune training is used in the same way as described in step 3 for testing.

Moreover, it is relevant to emphasize some more specific implementation details related to minibatches. PPO takes a train batch, whose dimension is determined by *train_batch_size* and defaults to 4000, divides it into n *sgd_minibatch_size* chuncks. For istance if *train_batch_size* is equal to 4000 and *sgd_minibatch_size* is equal to 400, then 10 smaller batches are created from the initial train batch. *Num_sgd_iter* parameter instead indicates how many times this minibatches are then used to updated the neural network. So in our example if *num_sgd_iter* was equal to 20, we would perform 20 x 10 updates at the same time on one single train batch. To avoid confusion, it is relevant to also specify the role of the *rollout_fragment_size* parameter which indicates the number of steps sampled each time. This parameter is valid only if the *batch_mode* is set to “truncated_episodes”, otherwise, in case it is ”complete_episodes”, this value is overridden by the episode length of the episode. This parameter represents the batch size collected from each worker (Ray RLlib distributes tasks across different workers to enable parallelism). Its role is to allocate a specific number of samples before verifying whether the *train_batch_size* has been reached. *Sgd_minibatch_size* is the size sampled from the

training buffer with a size of *train_batch_size*. Therefore, those are the two parameters we are mostly interested in from our perspective.
As partly explained in 3.1.6, in the experience replay part and also briefly mentioned in 3.3.1 and 3.3.2, although the reinforcement learning paradigm is based on continuous interactions with the environment and on feeding this experience on-policy and maybe also online therefore optimizing the algorithm while it is being used, neural networks work based on stochasticity. They are inefficient when optimized with a single sample at a time and need random sampling for better learning and convergence. If data are fed sequentially, the optimization risks to get suck in local extremas and is way less stable. The state-observation pair in experience at timeframe 0 tends to be very similar to the one in experience at timeframe one. Using an entire randomly sampled batch would be more effective since the update would have the gradients for all the different states that would be diversified and not similar.

This possibility of multiple updates with the same minibatch is instead a peculiar feature of the Proximal Policy algorithm whose clipped objective (section 3.1.8) allows multiple gradient descent passes over the same minibatch of experiences. However, experiences are not reused between mini-batches, only between epochs. The size of minibatch is an important parameter, indeed reducing minibatches can improve the model's final performance by ensuring convergence to a better local minimum. However, on the other side, if too small, it can cause instability in training.

Another needed clarification is related to seed and reproducibility of results. As a matter of fact, quite a lot of effort was put in obtaining reproducible results since there are many different seeds involved in the final algorithm implementation. In the end, reproducible results can be obtained by specifying four different seeds: one for PPO, one for the environment, one for Ray Tune library if it is used for hypeparameters tuning and one for the imitation learning algorithm if it is used.

4.6 Variant 1: GTrXL for sequence embedding

As already mentioned in the previous section, Ray RLlib allows to use GTrXL (section 3.2.2.6) as the neural network for the Proximal Policy Algorithm. Using it doesn't involve more specific steps in the training and testing of the algorithm apart from the initial GTrXL parameters definition:

- **Num_transformer_units:** The number of repeated Transformer units to use indicated as L by the authors of the GTrXL paper.

- **attention_dim:** The input and output dimensions of a single Transformer unit.
- **num_heads:** The number of attention heads to use in parallel, indicated with "H" in the transformer-XL paper (see section 3.2.2.5)
- **head_dim:** The dimension of a single attention head within a multi-head attention unit, denoted as "d" in the transformer-XL paper (section 3.2.2.5)
- **memory_inference:** The number of timesteps to concatenate over time and feed into the next transformer unit as inference input. The first transformer unit will receive this number of past observations plus the current one.
- **memory_training:** It is the same parameter as *memory_inference*, but for training.
- **position_wise_mlp_dim:** The dimension of the hidden layer in the position-wise MLP after the multi-head attention block inside a Transformer unit.
- **init_gru_gate_bias:** Initial bias values for the GRU gates. There are two GRUs per Transformer unit, one after the MHA, and one after the position-wise MLP.

As it will be later discussed in the future works, more tuning of those parameters could lead to results improvement if more effort is dedicated to it. It was not possible to do it in this project due to limited hardware resources as well as time limitations and many other possible improvements that were deemed more worthy and promising.

4.7 Variant 2: Pretraining with imitation learning

Imitation learning has been used mainly in the form of behavioural cloning with only some experiments using MARWIL (for more details section 3.3.3) as a form of pretraining. Meaning that a set of experts actions have been created in hindsight based on past data and have been used to train the MARWIL algorithm and then the so-obtained neural network weights of MARWIL have been copied to the PPO model to avoid starting the training from scratch using random initialization. The aim of this procedure was to use the available data twice thus increasing data efficiency and trying to reduce issues due to limited data that is one of the main problems of applying Reinforcement Learning to the Portfolio Optimization problem as already mentioned. Moreover, this process should give more stochasticity and diversity to the training process that usually enriches training in deep learning like it happens in self-supervised or multitask learning for instance.

MARWIL is also implemented by Ray RLib, however, the pretraining procedure is not. Therefore, the two algorithms networks' configurations needed to be the same. Then the final weights of MARWIL have been saved and moved to PPO. In order to copy the weights into PPO, the trainable function had to be modified. As a matter of fact, in Ray RLib, it is possible to customize the training loop by providing during initialization a custom trainable function overriding the common training procedure. Therefore, MARWIL weights have been added to the initial configuration parameters together with a callback to a modified trainable function that included, in the lines preceding the training loop, a PPO weight initialization using the function *set_weigths*. To make this possible, the entire training loop including various initializations and checkpoints saving needed to be reimplemented. Moreover, some Ray RLlib concurrency settings were causing the weight initialization to be cancelled so it is important to keep the same *local_workers* settings to obtain the desired pretraining effect.

Marwil parameters include classical neural network parameters such as learning rate, but also a specific parameter called beta. If this parameter equals zero, then MARWIL degenerates to plain behavioural cloning. For most experiments, beta has been kept equal to zero.

In this paragraph, more details on how the expert actions have been obtained are provided. As stated previously, prophetic expert actions have been calculating in hindsight meaning that information on future returns have been exploited to get the best action at each specific timeframe. For instance, to calculate the optimal action for the current state we can proceed as it follows:

- Assuming our training period is from January to June 2023 and we want to use those information to trade from July to December 2023
- We are trading daily so our goal is to train the model so that it can output the best action using the information available on that specific day (our state)
- Therefore, we will create a dataset of states and relative optimal actions for the whole training period in order to have some labeled data to train our model in a supervised manner.
- To make an example of how those optimal actions can be calculated, let's assume we need to calculate the optimal action for the ten of January 2023 for the AAPL stock. We know that the average price of AAPL in the previous ten days was 185 and we also know all its technical indicators values at that day and this set of information corresponds to our initial state.
- We also know in hindsight that the average price of AAPL from the first to the tenth of March was 195. Therefore, we can use this information to create the optimal action for the tenth of

January as a buy action since the stock price increased. Moreover, we can quantify by how much the stock rose using its return (195/185) and consequently how much we are willing to buy. This is useful since we want some indicative action values and not just some buy or sell signals.

- The model would be trained on the dataset obtained in this way from the training period automatically inferring relationships between input information of the state such as technical indicators values and future returns. Then this model could be used in the trading period or, as for this project, to pre-train and initialize the Deep Reinforcement Learning algorithm.

This procedure is based on some parameters that have been defined as the following:

- **Window:** number of days over which the average is computed. In the example above, that would correspond to ten days in between the first and tenth of January and the first and tenth of March.
- **Distance:** number of days between start and end *window*. In the example above, this would be equal to sixty.

As you can imagine, changing this parameters could influence a lot our training dataset, for example a higher *distance* value would give a longer term outlook to the model.

In order to improve this procedure, some other optimizations experiments have been made by introducing the following parameters:

- **Cutoff:** To avoid very small positive or negative returns that can introduce unnecessary noise and losses due to transactions fees when buying and selling. By specifying this value, it is possible to set to zero actions with value lower than cutoff value. However, in practice, surprisingly it has been noticed that this parameter doesn't improve results.
- **Buy_factor** and **sell_factor:** This is a multiplication factor applied equally to all output actions to rescale them. This parameter has been found to have a relevant effect.

Also in this case, there is plenty of unexplored possibilities concerning hyperparameters tuning both for MARWIL original parameters as well as expert actions creations parameters. As for GTrXL, the available time has been dedicated more to other experiment and only a limited hyperparameters optimization was made.

4.8 Variant 3: TD3-BC trained on multiple environments sampled from S&P 500

This variant was implemented to address the limitations observed from the results of the other two variants. The main issues were the limited amount of input data and the consequent instability of the results as well as the tendency to not diversify enough and the consequent overinvesting in a single stock. The use of TD3-BC (see paragraph 3.3.5) improves stability by allowing to regularize the policy training using the expert actions dataset obtained as explained for the second variant. (section 4.7) Moreover, since it is offline, it allows us to create a dataset containing training data from different portfolios. It is necessary for the algorithm to be an offline one since, for every different portfolio, a different environment has to be created due to different prices and indicators for different portfolios of stocks. This can be therefore considered a sort of multi-environment training setting.

To create multiple portfolios with different combinations of stocks, a number of stocks that matches the DJI30 are randomly selected from the S&P 500. As a matter of fact, both DJI30 and S&P500 can be considered as an approximation of the market since both track performances of respectively top 30 and top 500 largest-cap American companies. This justifies our choice of training the model on samples of the S&P500 stocks to then use it on the DJI30.

With this method, we can sample enough portfolios to obtain a training dataset with a size in line with the common sizes usually used by Deep Reinforcement Learning algorithms. Something that we were not able to achieve before due to the limited historical data available and since we observed that with increasing granularity also noise increases. In addition to that, training on different portfolios allows us to reduce bias (avoid behavior such as overinvesting in one stock because in the past it always performed well basing the decision-making entirely on the input indicators) and overfitting. (since, as already said, we have much more data available and a higher diversity of stocks creates more stochasticity in the dataset)

Moreover, another modification was to reduce the number of features in the state to avoid redundancy of information and since a high state dimension needs a lot of data for a successful training. Therefore, the state features were reduced to only Moving Average Convergence Divergence, Relative Strength Index, Commodity Channel Index, Log returns, Directional Movement Index, Efficient Frontier Weights with respect to the original set of indicators described in section 2.3.1. Moreover, the efficient frontier weights are a new indicator that was not present before. They consist of the plain efficient frontier weights (see section 2.2.2) with a maximum cutoff of 0.3. It was introduced mainly to also include information about correlation in the state for the model to be able to learn to diversify reducing unsystematic risk and therefore to reduce the observed tendency to overinvest in one stock (better explained in paragraph 5.1.3)

Moreover, the previous two variants were using FinRL library rescaling that just divides all indicators for the same factor of 2^7. Since we deemed this rescaling as imprecise since different indicators oscillate between different values, the indicators have been min-max rescaled between 0 and 1 for this third variant.

4.9 Results Manipulation and Visualization

This section of code takes care of transforming account values into returns and calculating portfolio performance measures and charts using pyfolio library (section 4.2.3).

The plot shown in sections 5.1.3 and 5.3.3 includes the following plot that are obtained with the functions below:

- The first main chart at the top compares the DJI30 index cumulative return with respect to the model's return and is plotted using plotly with the current granularity (*daily* for section 5.1.3 and 5.3.3 and *hourly* for section 5.2) with time on the x axis and cumulative return on the y axis.
- On the middle-left of the plot, the performance measures shown are obtained using pyfolio *backtest_stats* function since it includes all important performance metrics introduced in section 2.2.4.
- *Show_and_plot_top_positions*: On the middle-right position of the plot, the exposures and evolution of the top ten held positions throughout the entire trading history are plotted.
- *Plot_annual_returns*: On the bottom-left position, a bar graph comparing the different annual returns obtained each year is reported.
- *Plot_monthly_returns_dist* and *plot_monthly_returns_heatmap*: Respectively on the bottom-center and bottom-right position an histogram with a bar graph distribution of monthly returns counting number of occurrences of monthly returns values and a heatmap of returns by month relative to the whole testing period are plotted.

4.10 Hyperparameters tuning with Ray Tune

Ray tune library (section 4.2.1) was used for hyperparameters optimization only in variant 1 and 2, since, as already mentioned, variant 3 was not using ray Rlib and therefore it was performed manually changing hyperparameters. Ray Tune was used in conjunction with an hyperparameter optimization tool called Optuna. (74) In particular, the framework PyTorch+Optuna was exploited. To optimize a PyTorch model using Optuna, it is necessary to encapsulate the model in an objective function to be then able to access its config file to pick hyperparameters. The Optuna search space needs to be initialized as an object and passed to the run method. Moreover, the metric that has to be optimized needs to be specified and also the hyperparameters search optimization algorithms can be specified.

A snippet of the PPO parameters search space used for PPO is the following:

- `"entropy_coeff": tune.loguniform(0.00000001, 0.1),`
- `"lr": tune.loguniform(5e-5, 0.001),`
- `"sgd_minibatch_size": tune.choice([ 32, 64, 128, 256, 512]),`
- `"lambda": tune.choice([0.1,0.3,0.5,0.7,0.9,1.0])`

Where the search algorithm is defined as:

```
search_alg = OptunaSearch(
                  metric="episode_reward_mean",
                  mode="max",
                  seed=seed_search
                  )
```

And then used in Tune:

```
tuner = tune.Tuner(
                trainable=self.trainable,
                param_space=self.params,
                tune_config=TuneConfig(
                    search_alg=self.search_alg,
                         …
                    )
                 …
                 )
```

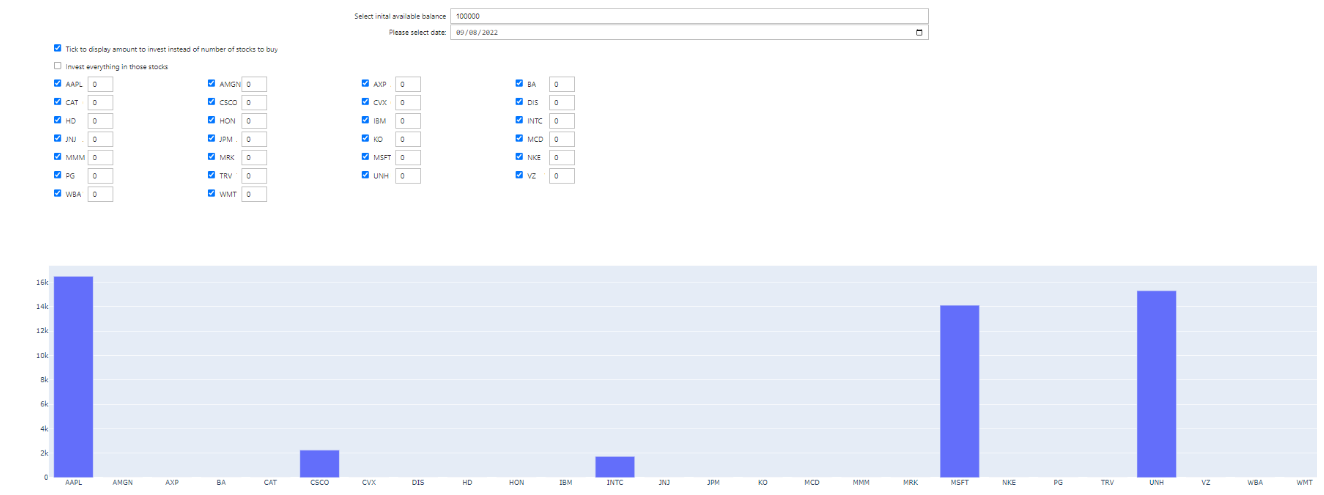
Tune also allows to define a scheduler such as the ASHA scheduler (75) that performs early-stopping for bad trials.

4.11 Portfolio Manager WebApp demo

To show one of the possible applications of the work done in this thesis project other than the obvious one of using the algorithm directly for algorithmic trading, a simple demo of a Portfolio Manager Assistant WebApp has been created. Such a platform could be used by banks or asset management funds to advise clients on the action the algorithm would take in a particular situation. As a matter of fact, Deep Reinforcement Learning would allow to specify any kind of information characterizing the current conditions and the needs of the investor. For instance, the investor could insert the current date that would allow the algorithm to retrieve prices and technical indicators, it could select only the stocks he wants the algorithm to consider among the available ones and it could insert the amount he already owns in each stock as well as the cash he has available to invest. In this way, the output actions selected by the algorithm (indicated in Figure 20 as the height of the bar chart) would be a very customized automatic advice based on the specific needs and situation of the client.

The WebApp from which the screenshot below was taken has been developed in a very simple and automatic manner not focusing on the interface, design and optimization of the application itself since the goal was not to invest much time and just give a good intuition and visual representation of this possible application. The tool used is called Voila (76) and it allows to easily build web applications starting from Jupyter notebooks. The tick boxes allow the user to specify if he wants to include the stock or not and the text box near the stock ticker allows to specify the already invested amount. At the top, it is possible to indicate the initial balance or cash investable amount and the date for the action prediction since the application allows to backtest the model. In a real application, the date would be the current date by default since past investments wouldn't be interesting. When changing any of the input variables, the bar chart output indicating the Portfolio Manager suggested actions variates accordingly.

Moreover, it was not possible to implement it in the context of this thesis project, but, as also mentioned in the future works, there are some techniques for explainability of Deep Reinforcement Learning algorithms. Among the researched ones, one of the simplest but yet most effective is SHapley Additive exPlanations or SHAP values. (77) SHAP works by assigning an importance value to each feature for a specific prediction without digging deeper into the substance of the algorithm but just by using states and actions. In the Deep reinforcement Learning context, this would mean knowing the importance of each state feature in calculating a specific action.


Select inital available balance 100000

Please select date: 09/08/2022

[✓] Tick to display amount to invest instead of number of stocks to buy

[ ] Invest everything in those stocks

[✓] AAPL 0	[✓] AMGN 0	[✓] AXP 0	[✓] BA 0
[✓] CAT 0	[✓] CSCO 0	[✓] CVX 0	[✓] DIS 0
[✓] HD 0	[✓] HON 0	[✓] IBM 0	[✓] INTC 0
[✓] JNJ 0	[✓] JPM 0	[✓] KO 0	[✓] MCD 0
[✓] MMM 0	[✓] MRK 0	[✓] MSFT 0	[✓] NKE 0
[✓] PG 0	[✓] TRV 0	[✓] UNH 0	[✓] VZ 0
[✓] WBA 0	[✓] WMT 0		

Figure 70: Portfolio Manager WebApp demo screenshot

For instance, we would know that the RSI value of APPL was particularly informative for the model when predicting the output action for AAPL. Therefore, the investor could check that RSI value to see if the model is taking decisions based on a consistent logic. Even if it must be taken into account that, in some cases, the model may detect connections between features that the investor is not fully able to understand even checking the values. For example, the algorithm could use a technical indicator calculated on the AAPL stock to infer the action for MSFT, this could seem off at first sight, but could have an implicit undetectable meaningful reason justifying it. However, this technique would give at least a decent glimpse of understanding of how the model is taking decisions, knowing how much it uses each piece of information in each situation.

5 Backtesting, Results and Conclusion

This section shows the results of the different variants implemented as well as the general final considerations on this project based on the results, other than on the entire research process that led to it.

5.1 Results of variants 1 and 2: GTrXL and pretraining with Behavioural Cloning

As already briefly mentioned, the first two variants have been developed at first, and the third variant was implemented only after, based on the observations obtained from the previous results. Therefore, results will be proposed following this order. This section focuses on the first two variants.

5.1.1 Overview of first two variants results

In the table below, the comparison between the Return and Sharpe of plain Proximal Policy Optimization versus the first two variants are reported. GTrXL variant uses the so called model instead of standard Multi-Layer-Perceptron for the PPO neural network structure and the pretraining variant uses behavioural cloning for setting initial weights of PPO.

Model	Best Return	Average Return improvement	Best Sharpe	Average Sharpe improvement
GTrXL	21.38%	-1.58%	0.85	-0.122
PPO with BC pre-training	**24.12%**	**2.57%**	0.95	0.065
PPO	23.29%	-	**0.99**	-

Table 1: Results comparison of variant 1 and 2 versus PPO. All the values are annual

Training: 2000-01-03 - 2016-07-30

Test: 2016-08-01 - 2023-01-02

Concerning pretraining with Behavioral Cloning, although an average improvement of 2.57 % in the average annual cumulative return may not seem significant, we should consider that the return with pre-training is higher than the other one by about 14.57%. Indeed, although it seems counterintuitive to calculate the percentage of a percentage return, it is relevant to look at this figure since it is the relative increase of the return obtained by adding Behavioral Cloning pre-training. As a matter of fact, an improvement of 2.57% on a 17% return is different than the same improvement on a higher return. Therefore, despite instability not being solved by pretraining, its influence on performance was noticeable. However, looking at the reward evolution during training, the algorithm seems to lose part of the pre-training effect quite soon. The reason for this behaviour has been identified in the actor-critic network. As a matter of fact, passing the Behavioral Cloning network weights becomes less effective since there is not only one network in PPO because the actor network is optimized towards the critic and then is iteratively used to update the critic only after a certain number of iterations. (for more details see section 3.1.7)

Conversely, GTrXL variant did not improve performances by achieving on average even worst results. Although the model's hyperparameters should be finetuned more to give a final opinion on its efficacy for this use case, it seems like it does not help the model to better embed the information in the state. This could be due to the fact that since the inputs of GTrXL were mainly indicators, they were already values obtained from a sequence of past timesteps. Following this line of reasoning, it is possible that by applying it on raw single-timestep prices to create a new indicator it could be more effective, however, in this way it wouldn't be possible to pass the gradient from the output to the initial input of GTrXL. This idea of making the sequence-modelling step also trainable with the gradient obtained from the PPO model and therefore from the output was the initial goal of introducing GTrXL in this project.

5.1.2 Considerations on results instability

When testing the algorithm and performing runs, the already mentioned high instability of Deep Reinforcement Learning algorithms trained on limited data was confirmed in practice. In particular, it has been quantified observing by how much the choice of the seed influences the results that was found to be almost as much as the chosen set of hyperparameters. The results in the table below demonstrates this by trying all combinations of four seeds with four different set of hyperparameters. In fact, changing seed influences the average cumulative return of 20.62%. In other words, this means that by keeping one set of hyperparameters fixed and changing five different seeds, the return changes on average 20.62% from the average return of the five runs. While changing the hyperparameters set and keeping the seed fixed has an influence of 27.06% on average on the average return. This is not a desired behaviour since the seed should not influence the results that much. This high instability makes it very difficult to choose hyperparameters and find a good set of hyperparameters and this inherent instability was not solved with our contributions.

	Params 1	Params 2	Params 3	Params 4	Average Return	Average change
Seed 1	11.07%	18.80%	17.72%	8.22%	13.95%	30.87%
Seed 2	4.85%	19.58%	17.28%	11.88%	13.40%	37.56%
Seed 3	9.90%	16.34%	12.96%	24.12%	15.83%	27.80%
Seed 4	10.14%	12.30%	14.28%	14.28%	12.75%	12.00%
Average Return	8.99%	16.76%	15.56%	14.63%		**27.06%**
Average change	23.03%	14.53%	12.47%	32.46%	**20.62%**	

Table 2: Seed vs Hyperparameters influence on results variant 2.

Training: 2000-01-03 - 2016-07-30

Test: 2016-08-01 - 2023-01-02

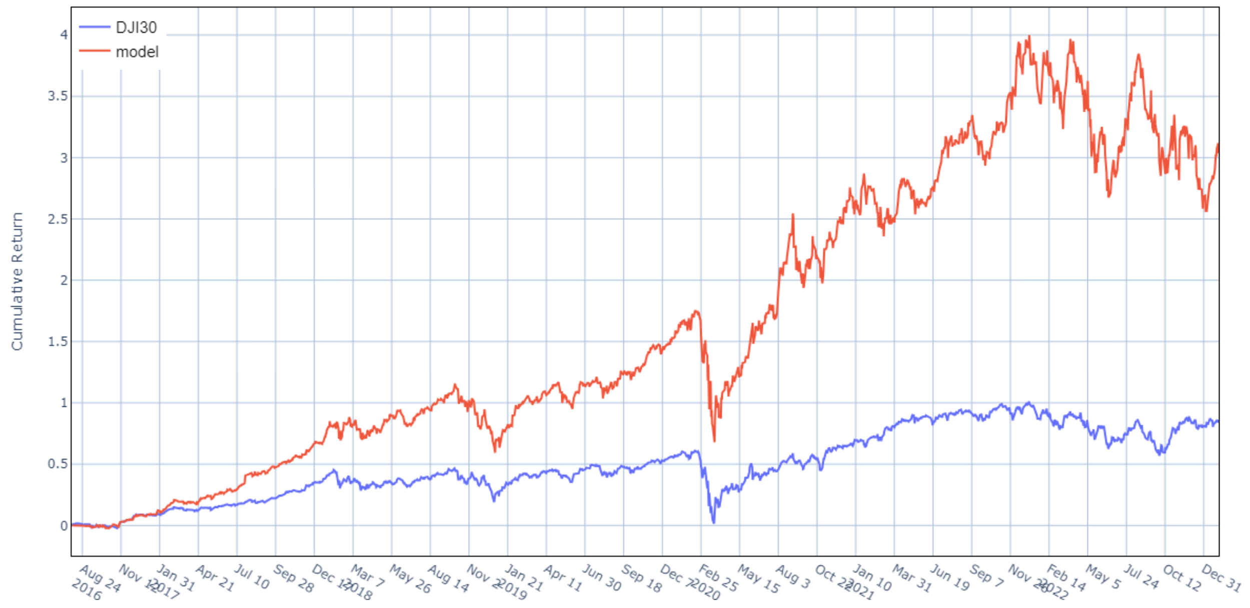
5.1.3 Focus on single runs behaviour

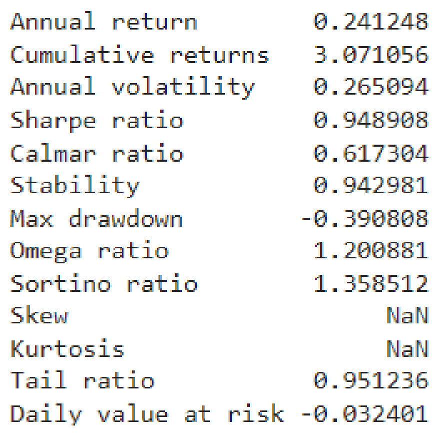
Below in plot 1 and plot 2 more detailed backtesting statistics and charts are showed for the best and worst runs of the the PPO with behavioural cloning pre-training variant which are the best and worst reported in Table 2. For each run, the first figure will represent the cumulative return over the testing period, the second will represent the portfolio performance indicators (refer to section 2.2.4 for details on the indicators) and the evolution of the top ten portfolio weights over time; and the third one will show distributions and heatmaps of annual and monthly returns. For more details on the single charts, refer to section 4.9.

Training: 2000-01-03 - 2016-07-30

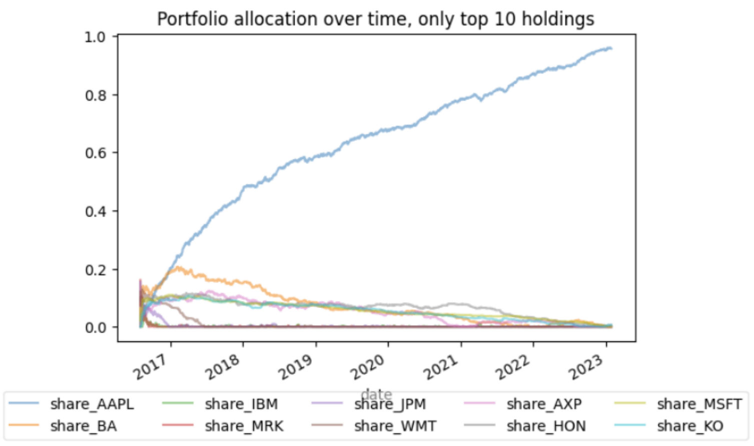
Test: 2016-08-01 - 2023-01-02

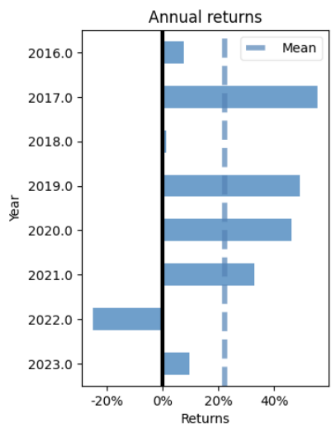
- **Plot 1**: Best run PPO with pre-training

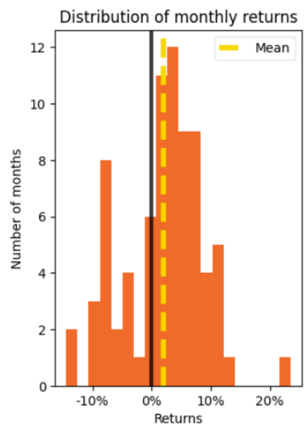


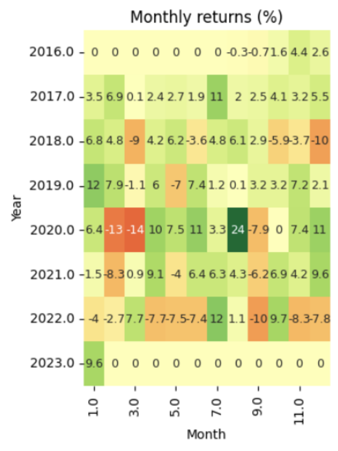


```
Annual return          0.241248
Cumulative returns     3.071056
Annual volatility      0.265094
Sharpe ratio           0.948908
Calmar ratio           0.617304
Stability              0.942981
Max drawdown          -0.390808
Omega ratio            1.200881
Sortino ratio          1.358512
Skew                        NaN
Kurtosis                    NaN
Tail ratio             0.951236
Daily value at risk   -0.032401
```

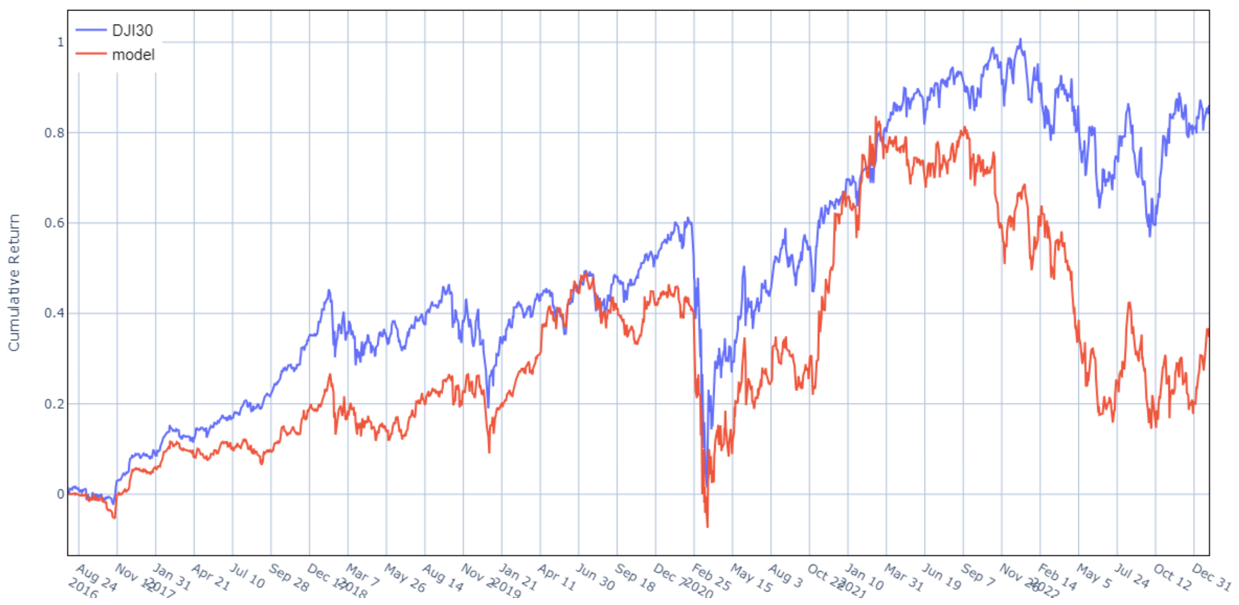




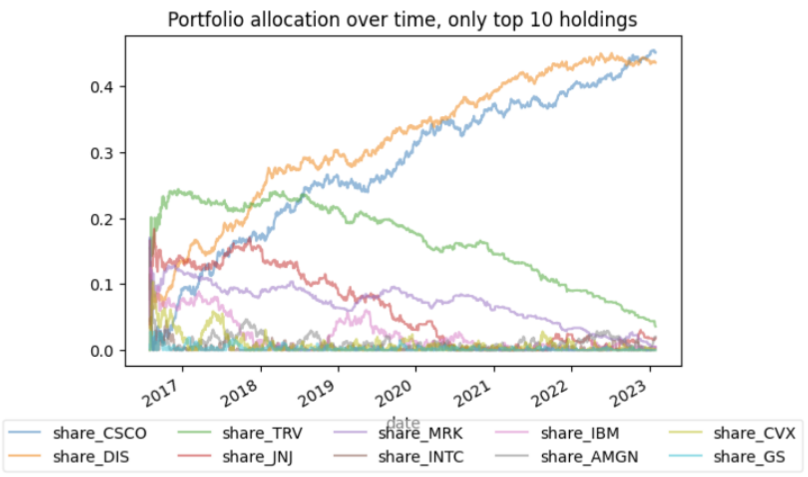


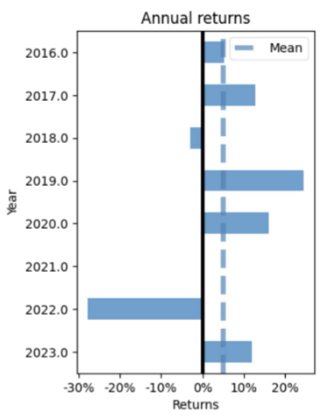


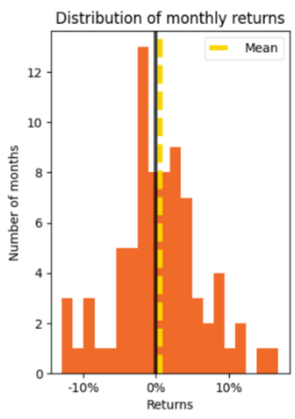
- **Plot 2**: Worst run PPO with pre-training

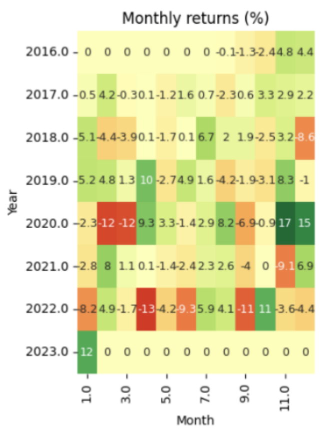


```
Annual return          0.048449
Cumulative returns     0.359805
Annual volatility      0.220611
Sharpe ratio           0.325073
Calmar ratio           0.127776
Stability              0.462367
Max drawdown          -0.379173
Omega ratio            1.066878
Sortino ratio          0.461247
Skew                        NaN
Kurtosis                    NaN
Tail ratio             0.998243
Daily value at risk -0.027510
```









In plot 1, the top chart shows a constant detachment of the model's cumulative return with respect to the index's cumulative return which indicates a constant improvement of the model with respect to the baseline. The cumulative return of 3.07 implies that the model would have tripled the invested amount in about 5 years and a half time. Average annual return is equal to 24.2% and Sharpe ratio is 0.95 almost reaching1 which is usually considered as a performant investment strategy. In the bottom-right bar chart, it can be observed that 2022 was the only year with a negative return and the monthly returns distribution is quite stable apart for few values over 10% in 2020 and 2022.

In plot 2, the top chart shows that the model alternates overperformance and underperformance with respect to the baseline. Especially in 2022, it considerably underperforms the Dow Jones index. Sharpe ratio is also very low at 0.33. In the bottom-right chart, it can be observed that other than 2022, also 2018 had a negative return and the monthly returns distribution shows many more months with a return higher than 10% in absolute value.

Another behavior of the models that is worth pointing out and was evident in most of the runs as well as in both the plots above in the middle-right portfolio positions evolution chart is the tendency to overinvest in one stock. In plot 1, the share of AAPL constantly increases until it reaches 100%. This is an undesired behavior that goes against the portfolio management principle of diversification explained in section 2.2.2. Also in plot 2, even if the maximum share of one stock never goes above 40%, the tendency of CSCO and DIS is to always increase, therefore, on a longer time period, they would probably reach 100% as it happened in most of the other observed runs.

On the other side, it was also pointed out in the introductory sections that higher return requires higher risk so, although the model doesn't diversify enough and therefore keeps some diversifiable volatility, the logic of higher risk for higher return is somehow embraced since the model tends to invest in stocks with higher risks and higher return. As better pointed out previously, we are not currently introducing variance and correlation information in the model's input, therefore, not giving to the model enough information to learn diversification.

5.2 Results using higher time granularities

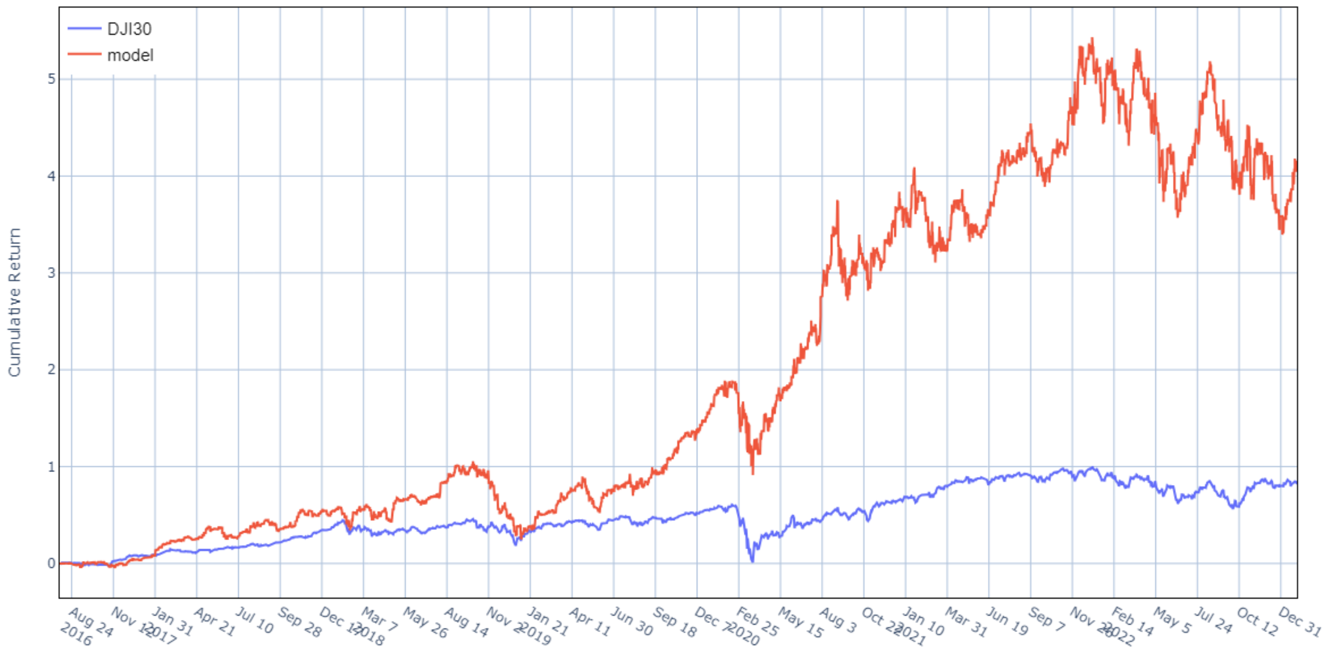
Following the assumption that the instability is due to a limited amount of input data, another experiment was to try to increase time granularity to have more available data points. Increasing granularity, also the training and testing period had to be reduced due to RAM and computation constraints. Therefore, daily results in the following table are not comparable with the other results because the training spanned over a smaller time window. The risk of high time granularity is that data could become more noisy. In the following table the results obtained with Ray Tune runs over different time granularities are reported. The runs have been performed over two different time periods where the first one is more bearish and the second more bullish.

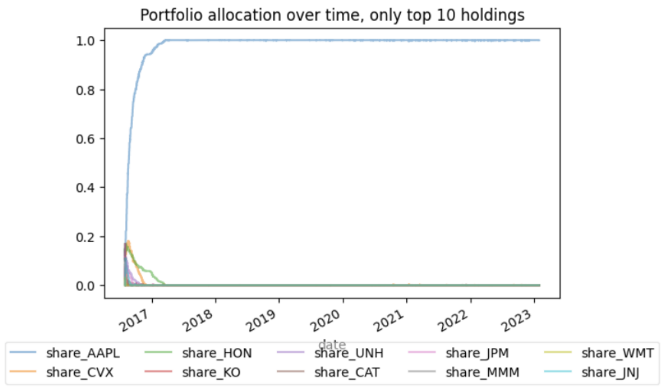
Time granularity	Training Period	Test Period	Return of PPO with pre-training	Sharpe of PPO with pre-training	Return of BC without PPO
5 min	01/01/2019 – 2021/12/31	01/01/2022 – 2022/12/31	-2.50 %	-0.01	-4.72 %
30 min	*	*	-30.80 %	-0.25	-40.90 %
1 hour	*	*	**9.00 %**	**0.18**	-3.59 %
daily	*	*	-9.31 %	-0.45	-5.37 %
1 min	01/06/2020 – 2021/12/31	01/01/2022 – 2022/05/31	-0.41 %	-0.14	**-0.41 %**
5 min	01/01/2016 – 2018/12/31	2019/01/01 – 2019/12/31	-1.26%	0.01	0.12%
30 min	*	*	**16.50%**	**0.30**	11.30%
1 hour	*	*	6.69%	0.16	10.08%
daily	*	*	1.81%	0.19	**15.41%**

Table 3: Different time granularities comparison

In both cases, the only granularity that seems to compete and maybe even do better than *daily* is the *hourly* one. All the other granularities seem to perform poorly and show even greater instability with respect to the daily granularity. For instance, the *30 minutes* granularity performs very well in the second time period, but badly in the first one. Therefore, some additional experiments have been conducted with the *hourly* granularity and it seems that it can lead to better results, but even to worst ones, depending on the run. Therefore, it appears to be even more instable than the *daily.* As a matter of fact, even if way less runs were performed with this granularity, in one run the algorithm was able to obtain much better return as showed in the following plot number 3. However, in the few runs performed, also very bad performances were observed confirming the initial hypothesis that higher granularities lead to more instability. Since this granularity is similar to *daily,* it was possible to try it on the same training and test period as the one in the initial results sections.

- **Plot 3**: Run with minute data PPO with pre-training Training: 2004-01-03 - 2016-07-31, Test: 2016-08-01 – 2023-01-02 (for more details on the chart specifics see section 4.9)





5.3 Results of variant 3: TD3-BC model with multiple environments sampled from S&P500 training

Another result that wasn't mentioned before is that behavioral cloning alone underperforms by about 2.45% in absolute cumulative final return the Proximal Policy Optimization algorithm results. Behavioural cloning is a much simpler algorithm based on supervised learning and it wasn't optimized as much as PPO. Therefore, for the third variant, all the complexity of Deep Reinforcement Learning algorithms was considered as not necessary and it was partly avoided by using a better finetuned Offline Reinforcement Learning method: the TD3-BC model. Moreover, we remark that the main drawbacks evinced from the previous two variants were the instability due to the limited amount of data and the overinvesting behaviour.

As explained more in details in section 4.8, the third variant addresses all those issues by training TD3-BC on a series of portfolios sampled rom S&P 500, reducing the input dimensionality to just 5 indicators with a different rescaling.

5.3.1 Main results and comparison with PPO pre-training

In table 4 below you can see a comparison between the results obtained with the second pre-training variant and the third one that is the TD3-BC variant over the 16 runs used to study seed and hyperparameters influence (shown in sections 5.1.2 and 5.3.2 table 2 and 5 for variant 2 and 3 respectively). The TD3-BC third variant seems to overperform the other one in best, worst and average Return and Sharpe over multiple runs.

Model	Best Return	Worst Return	Average Return	Best Sharpe	Average Sharpe
TD3-BC	**28.30%**	**10.50%**	**16.6%**	**1.14**	**0.73**
PPO with BC pre-training	24.12%	4.85%	13.98%	0.95	0.63

Table 4: Results comparison variants 2 and 3. All the values are annual.

Training: 2000-01-03 - 2016-07-30

Test: 2016-08-01 - 2023-01-02

As explained in section 3.3.5, one of the main modifications of the TD3-BC paper with respect to the original TD3 model is to normalize input states. Initial experiments were made to compare performances of preprocessing only using rescaling of the data frame with respect to also adding state normalization afterwards. Although using only rescaling showed a better diversification with a 30% average maximum share of a single stock in the portfolio, Normalization has been found to increase performance by on average 2.6% in Return while still keeping a 60% average share for the stock with the maximum share in the portfolio which was way better diversification than the previous variant which most of the times tended to reach 100%.

5.3.2 Considerations on results instability and hyperparameters optimization

Regarding instability, the situation did not improve much showing again a similar influence of changing seed and changing hyperparameters with the first one influencing by about 19.11% and the second one by 17.57%. Thus making the choice of the hyperparameters set quite meaningless in general apart from one set of hyperparameters. As a matter of fact, as you can see in the table 5 reported below, for hyperparameters set number one, the average change in Return is 1.47% which is much lower than the other ones. This could be due to the fact that, as you can see in the appendix, in this set of hyperparameters, *alpha* has a high value of 10. Alpha should determine how much the policy optimization is regularized with the expert actions, therefore it is reasonable to hypothesize that such a value results in a more stable training. However, also the set number three has the same *alpha* value and instability is high, this could be due to the different combination with the other hyperparameters such as the batch size that is lower. Moreover, with this third variant, on average, results oscillates less, the minimum, average and maximum return are higher as shown and pointed out in the previous paragraph.

	Params 1	Params 2	Params 3	Params 4	Average Return	Average change
Seed 1	18.20%	17.21%	14.00%	14.57%	15.59%	10.69%
Seed 2	18.90%	15.90%	18.20%	25.20%	19.55%	33%
Seed 3	18.90%	11.74%	24.70%	10.50%	16.46%	19%
Seed 4	19.01%	16.89%	24.28%	28.30%	22.12%	16%
Average Return	18.75%	15.43%	20.25%	19.64%		**19.11%**
Average change	1.47%	11.97%	20.67%	36.18%	**17.57%**	

Table 5: Seed vs Hyperparameters influence on results variant 3

Concerning the other hyperparameters, the default hyperparameters proposed by the TD3-BC paper were changed one at a time to create the set of fifteen runs reported in the annex. 256 seems to be the most effective batch size since also diversification tends to be higher when using this value. Also increasing a bit the exploration noise from 0.1 to 0.3 or 0.5 improved the performance.

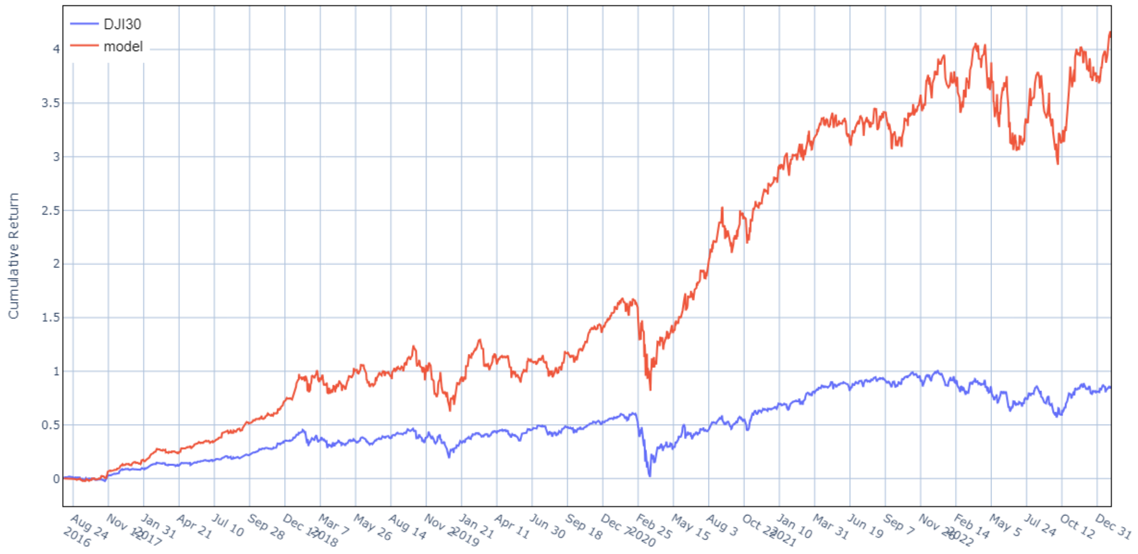
5.3.3 Focus on single runs behaviour

In this section, performances of the best and worst run of the 16 considered above in table 5 are reported in more details in plot 4 and 5. For each run, the first figure will represent the cumulative return over the testing period, the second will represent the portfolio performance indicators (refer to section 2.2.4 for details on the indicators) and the evolution of the top ten portfolio weights over time; and the third one will show distributions and heatmaps of annual and monthly returns. For more details on the single charts, refer to section 4.9.

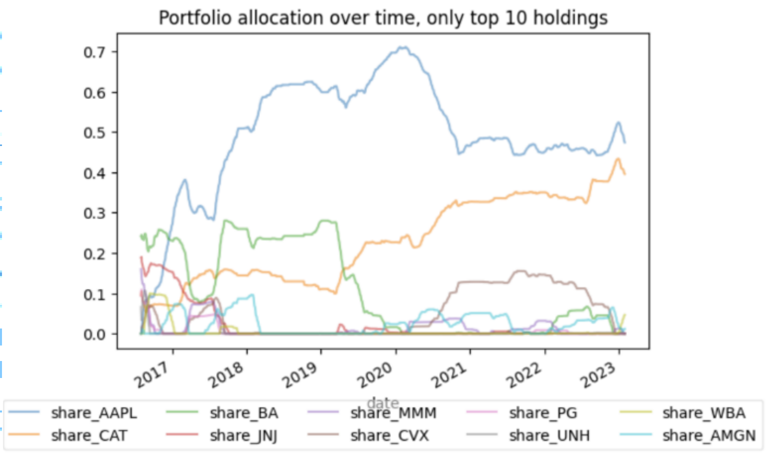
Training: 2000-01-03 - 2016-07-30

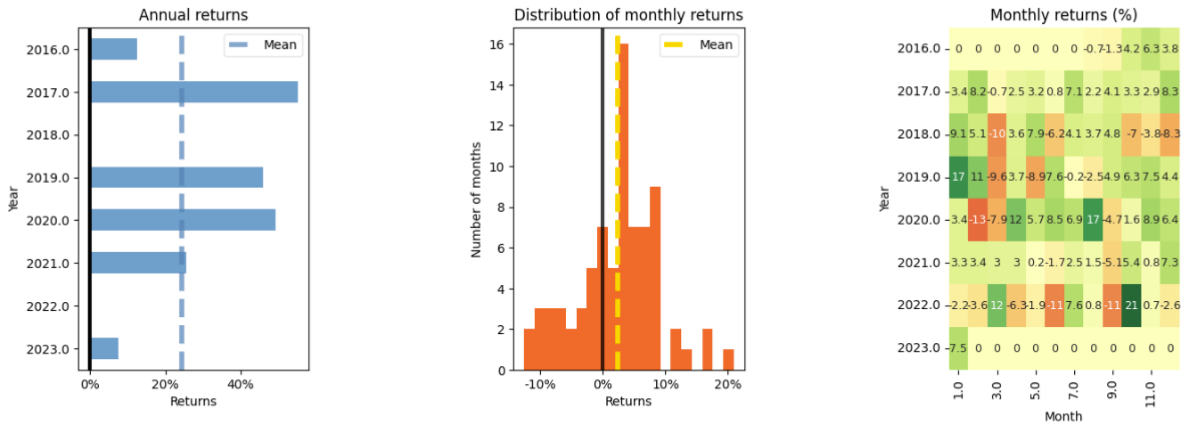
Test: 2016-08-01 - 2023-01-02

- **Plot 4**: Run with best return for TD3-BC

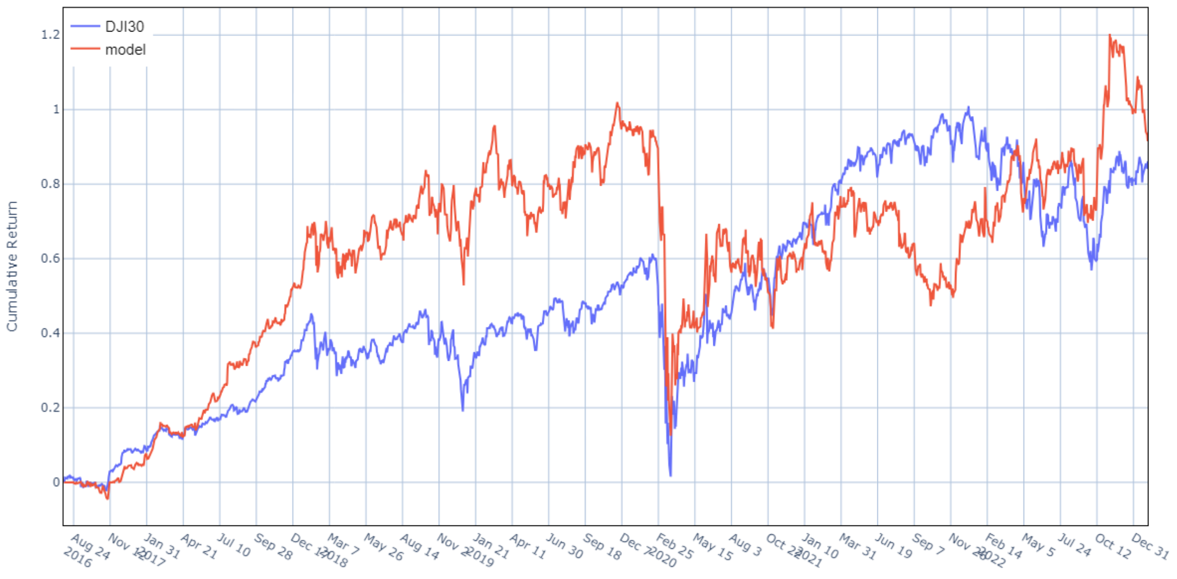


```
Annual return          0.282779
Cumulative returns     4.041469
Annual volatility      0.243071
Sharpe ratio           1.147422
Calmar ratio           0.877517
Stability              0.948528
Max drawdown          -0.322249
Omega ratio            1.240066
Sortino ratio          1.656783
Skew                        NaN
Kurtosis                    NaN
Tail ratio             1.046671
Daily value at risk   -0.029517
```

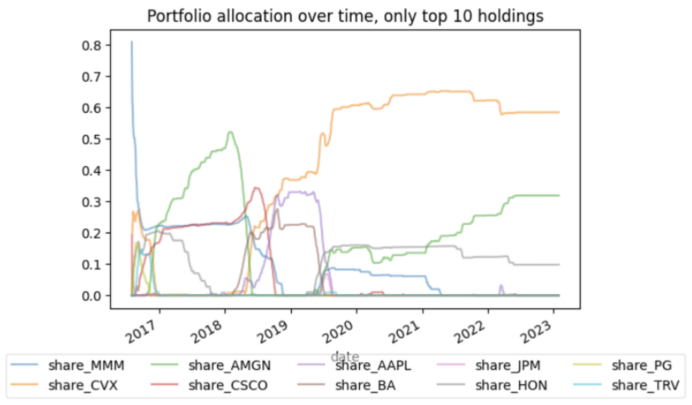


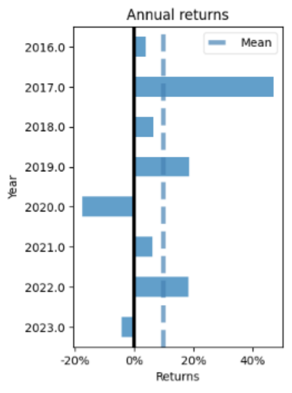


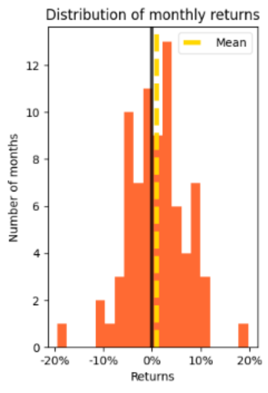
- **Plot 5**: Run with worst return for TD3-BC

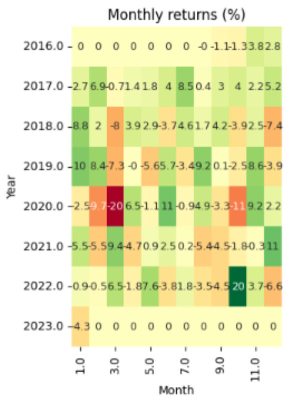


```
Annual return          0.105819
Cumulative returns     0.922089
Annual volatility      0.229345
Sharpe ratio           0.554170
Calmar ratio           0.238766
Stability              0.480507
Max drawdown          -0.443193
Omega ratio            1.116718
Sortino ratio          0.776665
Skew                        NaN
Kurtosis                    NaN
Tail ratio             0.957137
Daily value at risk   -0.028390
```









As you can notice in both plot 4 and plot 5 the overinvesting behaviour is reduced with respect to plot 1 and plot 2, diversification increases and also by observing all the single runs, more diverse and less asymptotical behaviours with respect to the previous variants were noticed.

In plot 4, the top chart shows again a constant detachment of the model's cumulative return with respect to the index's cumulative return which indicates a constant improvement of the model with respect to the baseline. Average annual return is 28.3% and as the total cumulative return of 4.04 shows that the model would have quadrupled the invested amount in about 5 years and a half time. Sharpe ratio is higher than one indicating a very good investment performance. In the bottom-right bar chart, it can be observed that all years showed positive return including 2020 and 2022. The monthly returns distribution is quite stable apart for five or six values over 10% in 2020 and 2022. Similarly to plot 1, AAPL was the stock with the highest share, but, as you can see in this plot, the amount invested doesn't always increase: it also diminishes in some periods and never goes above 70%. The fact that the model often choses AAPL is a good sign since it is one of the most performant stocks of the Dow Jones index.

In plot 5, the top chart shows that the model alternates overperformance and underperformance with respect to the baseline. Even if this was the worst performant model for the third variant TD3+BC, overall, in the considered testing period, it beats the index. Annual average Return is 10.6% and Sharpe ratio is 0.55. In the bottom-right chart, it can be observed that, differently than plot 2, years 2020 and 2023 are the ones that show a negative return. The monthly returns distribution shows only few months with a return higher than 10% in absolute value, it is worth notice a -20% in march 2020 due to covid that was better handled by the model in other runs.

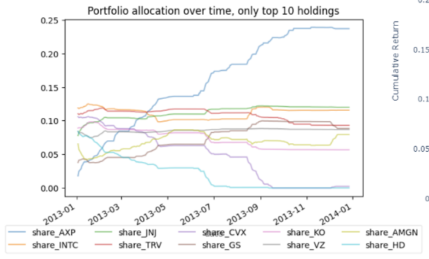
5.3.4 Results of bull and bear periods tests

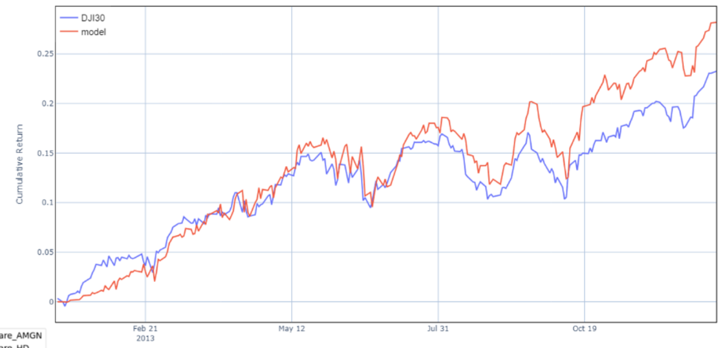
One last experiment was to check how the model performed in bearish and bullish periods. The same considerations made in section 5.2 regarding the fact that performances are not comparable with most other runs because of the shorter training period can be made in this section as well. However, this variant seems to beat the index more consistently over different periods. It almost always beats the index apart from Bullish period 2 where it slightly underperforms it. It was not possible to identify a determinant preference for Bear or Bull, however the model overperformed the index more in bearish periods. It is worth noting that Bearish period 2 showed a particularly good performance. Finally, it must be pointed out that it is difficult to find a long consecutive Bear or Bull period, therefore also the considered ones tend to be mixed. An interesting experiment could be to try to merged non-consecutive periods that have bearish or bullish to have more training data of a kind.

- **Plot 6: Bullish period 1**

TRAIN_START_DATE = '2010-01-01' TRAIN_END_DATE = '2012-12-31'
TEST_START_DATE = '2013-01-01' TEST_END_DATE = '2013-12-31'

Return model: 28.5%
Sharpe model: 2.16
Return DJI30: 23.24%

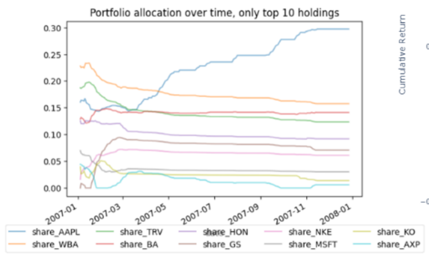


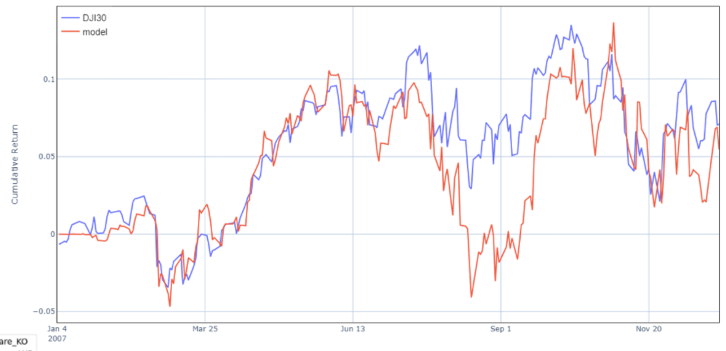


- **Plot 7: Bullish period 2**

TRAIN_START_DATE = '2004-01-01' TRAIN_END_DATE = '2006-12-31'
TEST_START_DATE = '2007-01-01' TEST_END_DATE = '2007-12-31'

Return model: 5.42%
Sharpe model: 0.37
Return DJI30: 7.04%

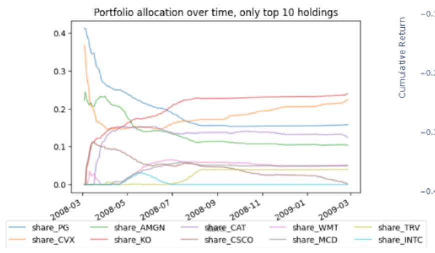


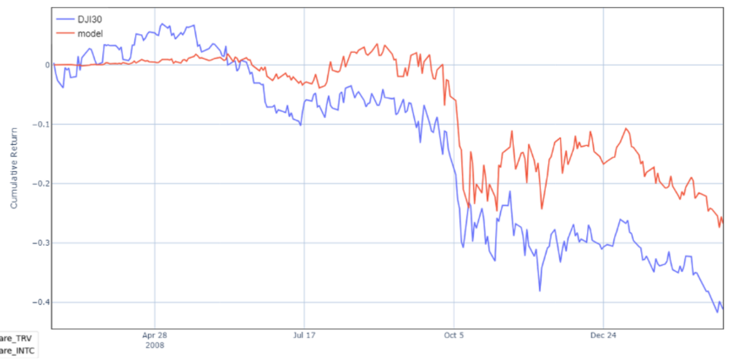


- **Plot 8: Bearish period 1**

TRAIN_START_DATE = '2005-03-01' TRAIN_END_DATE = '2008-03-01'
TEST_START_DATE = '2008-03-01' TEST_END_DATE = '2009-03-01'

Return model: -29.00%
Sharpe model: -0.76
Return DJI30: -41%

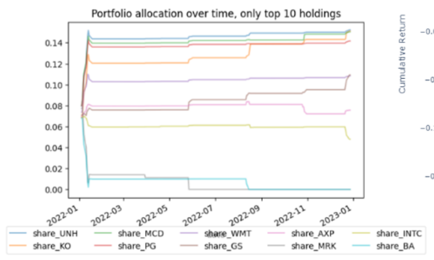


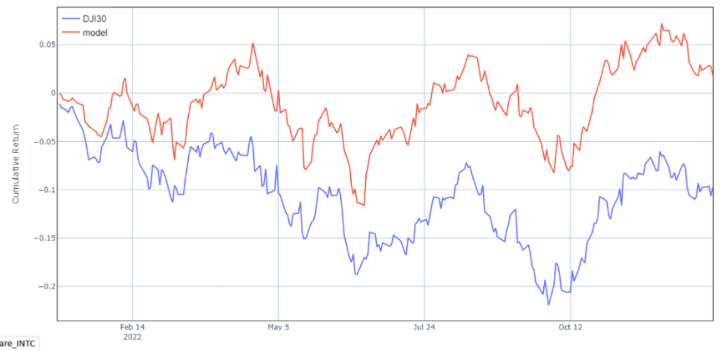


- **Plot 9: Bearish period 2**

TRAIN_START_DATE = '2019-01-01' TRAIN_END_DATE = '2021-12-31'
TEST_START_DATE = '2022-01-01' TEST_END_DATE = '2022-12-31'

Return model: 2.48%
Sharpe model: 0.226
Return DJI: -9.75%





5.4 Conclusions

As explained extensively in sections 5.1-5.3, GTrXL variant was not successful and underperformed the plain PPO while Behavioral Cloning pre-training of PPO considerably improved performance. Both variants showed lack of diversification in their investment strategies and high instability that was quantified by evaluating the effect of changing the seed.

The experiments with higher granularity data did not lead to improvements. Indeed, while higher granularity seem to have more information allowing the algorithm to learn even more in some instances, it leads to even grater instability.

TD3-BC variant managed to increase diversification thus addressing the unwanted overinvesting behaviour, reduced corner cases and showed a more consistent behavior while improving performances. This by exploiting the greater amount of data obtained by sampling random portfolios from S&P 500 as well as the expert actions regularization.

In almost all (all for the TD3-BC variant) the experiments performed with daily granularity, the algorithms overperformed the index, that is an approximation of the market. In trading, this is not considered as a straightforward task since most investors fail to do so. This shows that the potential of this technique is high also due to the impact of its practical applications as pointed out in section 4.11. And this stems from the natural parallelism between the Reinforcement Learning paradigm formulation and the stock market functioning.

However, as already mentioned before, the main issue of using Deep Reinforcement Learning is the instability of this kind of algorithms. And this problem, that is even more evident for our use case where data are limited and cannot be produced effectively, was not successfully solved even with the last variant.

Therefore, the most logic conclusion is that the data embedded in the state are not informative enough for the model to take effective decisions in a consistent manner. This does not totally come as a surprise considering technical indicators were used and that their effectiveness is debated. To briefly mention their the main limitations that are dealt with in section 2.3, other than technical indicators being designed to be used in practice by human traders often together with visual charts, the assumptions of the past being informative for the future and that all the information are embedded in the stock price are not always true in practice.

5.5 Future Works

As stated in the previous section, the main and essential modification needed would be to find more informative data and better model the state. This is a very challenging task since the Reinforcement Learning state should include all the relevant information from the history without noise and, in the stock market, relevant information and its cause-effect relationships are endless. The most effective information to add would probably be financial statements and fundamental indicators as well as news data and the sentiment that can be extracted from it since ultimately the price is determined by investors willingness to buy and sell a certain stock at a specific time. However, financial statements are published quarterly, therefore data granularity is too low to use this kind of data unless a way to increase granularity can be found. On the other hand, news data labeled by ticker are difficult to find and quite expensive and usually, they cannot be found for more than five years in the past and some stocks receive more mediatic attention than other ones.

Regarding diversification and the tendency to overinvest in single stocks, another important change could be to better shape the reward function so as to also reward diversification by negatively weighing volatility. The easiest way would be to have as reward the Sharpe ratio, but also more complex possibilities could be considered such as differential Sharpe ratio. Adding on to the client-centricity and customization discussion made in section 4.11, different reward functions could be designed to have algorithms with different risk appetites. For example simple return would imply higher risk, Sharpe ratio would be a trade off and a ratio such as return divided by volatility squared or Sharpe ratio divided by volatility would lead to a more risk averse algorithm. Conversely, limiting the algorithm to a maximum investment threshold in each stock should be avoided since this would limit also the learning capabilities of the model, diversification should be something learned by the model and not imposed externally by the environment.

In order to increase the algorithms stability, some ensemble model could be studied so as to reduce the instability of single models. However, it is difficult to create an ensemble for a RL algorithm due to its formulation. One simple option would be to select for trading only the algorithm with the best training reward among a set of algorithms. The production of synthetic data could also be considered to reduce instability, but it is a very pretentious experiment since we don't have a close enough model of how the stock market works. There are some interesting related works in this field such as (78).

As already mentioned in section 4.10 the SHAP values technique should be tried in this context since it could give a lot of insights on how the model takes decisions indicating which input feature determine a specific output action.

Regarding Deep Learning, it would be interesting to experiment other decision-making models options different from RL:

For instance, some transformers-based models focused on decision-making have been created and those would be interesting because of their capability to model time-series information as stock prices trends are. Just to mention two options, it could be worth to investigate (79) and (80)

Due to the recent success of Graph Neural Networks and their possible applications to multivariate time-series, also this field could be worth of exploration. Some interesting papers use Graph Neural Networks to leverage known or latent relationships between time variables, a possible strategy involves detecting interactions among variables using a graph learning framework. And subsequently, Graph Neural Networks are applied on top of the inferred graph. For instance, (81) and (82) link variables with attention-like mechanisms.

Moreover, neuroevolution ((83) and (84)) could be explored as well since some of their feature may indicate that they are able to better exploit data and have better data efficiency due to their logic of optimizing weights from data by trying multiple random weights.

Bibliography

1. *Portfolio selection.* Markowitz, Harry. 1952, The Journal of Finance 7.1, p. 77–91.

2. Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction.* 2014.

3. Goodfellow, Ian et al. *Deep learning. Vol.* s.l. : MIT press Cambridge, 2016.

4. Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre. Stabilizing transformers For Reinforcement Learning. [Online] 2019. https://arxiv.org/pdf/1910.06764.pdf.

5. Jonathan Berk, Peter DeMarzo & Jarrad Harford. *Fundamentals of Corporate Finance, 4th Global Edition.* s.l. : Pearson, 2017.

6. Hull, John C. *Options, Futures And Other Derivatives, 10th edition.* 2017.

7. *The Sharpe Ratio.* Sharpe, William F. 1994, The Journal of Portfolio Management.

8. *CAPITAL ASSET PRICES: A THEORY OF MARKET EQUILIBRIUM UNDER CONDITIONS OF RISK.* Sharpe, William F. 1964, The Journal of Finance.

9. Kenton, Will. Investopedia, Sortino Ratio. [Online] 2020. https://www.investopedia.com/terms/s/sortinoratio.asp#:~:text=The%20Sortino%20ratio%20is%20a,standard%20deviation%20of%20portfolio%20returns..

10. Hayes, Adam. Investopedia, Max Drawdown. [Online] 2022. https://www.investopedia.com/terms/m/maximum-drawdown-mdd.asp.

11. Kenton, Will. Investopedia, Var. [Online] 2023. https://www.investopedia.com/terms/v/var.asp.

12. Stockstats library. [Online] https://pypi.org/project/stockstats/.

13. Chen, James. Investopedia. [Online] 2023. https://www.investopedia.com/terms/e/ema.asp.

14. Dolan, Brian. Investopedia, Moving Average Convergence Divergence. [Online] 2023. https://www.investopedia.com/terms/m/macd.asp.

15. Hayes, Adam. Investopedia, Bollinger Bands. [Online] 2023. https://www.investopedia.com/terms/b/bollingerbands.asp.

16. Fernando, Jason. Investopedia, Relative Strength Index. [Online] https://www.investopedia.com/terms/r/rsi.asp.

17. Mitchell, Cory. Investopedia, Commodity Channel index. [Online] 2022. https://www.investopedia.com/terms/c/commoditychannelindex.asp.

18. Investopedia, Directional moving Index. [Online] 2021. https://www.investopedia.com/terms/d/dmi.asp.

19. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou. Playing Atari with Deep Reinforcement Learning. [Online] 2013. https://arxiv.org/pdf/1312.5602.pdf.

20. Silver, David. Lectures on Reinforcement Learning, Lecture 1: Introduction to Reinforcement Learning. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/intro_RL.pdf.

21. *Mastering the game of Go without human knowledge.* David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel & Demis. 2017.

22. John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino. Highly accurate protein structure prediction with AlphaFold. [Online] 2021. https://www.nature.com/articles/s41586-021-03819-2.

23. Silver, David. Lectures on Reinforcement Learning, Lecture 2: Markov Decision Processes. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf.

24. Bellman, Richard. *Dynamic programming.* 1957.

25. Silver, David. Lectures on Reinforcement Learning, Lecture 3: Planning by Dynamic Programming. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/DP.pdf.

26. Silver, David. Lectures on Reinforcement Learning, Lecture 4 & 5: Model-Free Prediction and Control. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/MC-TD.pdf.

27. Doshi, Ketan. Reinforcement Learning Explained Visually: Deep Q Networks, step-by-step. [Online] https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b.

28. Silver, David. Lectures on Reinforcmeent Learning, Lecture 7: Policy Gradient Methods. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf.

29. Lectures on Reinforcement Learning, Lecture 6: Value Function Approximation. *https://www.davidsilver.uk/teaching/.* [Online] 2015. https://www.davidsilver.uk/wp-content/uploads/2020/03/FA.pdf.

30. Yoon, Chris. Understanding Actor Critic Methods and A2C. [Online] 2019. https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f.

31. Timothy P. Lillicrap1, David Silver, Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. [Online] 2016. https://arxiv.org/pdf/1602.01783v2.pdf.

32. Karagiannakos, Sergios. The AI summer: The idea behind Actor-Critics and how A2C and A3C improve them. [Online] 2018. https://theaisummer.com/Actor_critics/.

33. Timothy P. Lillicrap, David Silver, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, Daan Wierstra. CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING. [Online] 2019. https://arxiv.org/pdf/1509.02971.pdf.

34. Uhlenbeck, G. E. On the THeory of Brownian Motion. [Online] 1930. https://djalil.chafai.net/docs/M2/history-brownian-motion/Uhlenbeck%20&%20Ornstein%20-%201930.pdf.

35. Yoon, Chris. Deep Deterministic Policy Gradients Explained. [Online] 2019. https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b.

36. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. Proximal Policy Optimization Algorithms. [Online] https://arxiv.org/pdf/1707.06347.pdf.

37. Sergey Levine, John Schulman, Philipp Moritz, Michael Jordan , Pieter Abbeel. Trust Region Policy Optimization. [Online] 2017. https://arxiv.org/pdf/1502.05477.pdf.

38. Karunakaran, Dhanoop. Proximal Policy Optimization(PPO)- A policy-based Reinforcement Learning algorithm. [Online] 2020. https://medium.com/intro-to-artificial-intelligence/proximal-policy-optimization-ppo-a-policy-based-reinforcement-learning-algorithm-3cf126a7562d.

39. Scott Fujimoto, Herke van Hoof, David Meger. Addressing Function Approximation Error in Actor-Critic Methods. [Online] 2018. https://arxiv.org/pdf/1802.09477.pdf.

40. Byrne, Donal. TD3: Learning To Run With AI. [Online] 2019. https://towardsdatascience.com/td3-learning-to-run-with-ai-40dfc512f93#:~:text=TD3%20uses%20clipped%20double%20Q,two%20evils%20if%20you%20will).&text=This%20method%20favours%20underestimation%20of,the%20algorithm%2C%20unlike%20overestimate%20values.

41. Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. [Online] 2018. https://arxiv.org/pdf/1801.01290.pdf.

42. V.Kumar, Vaishak. Soft Actor-Critic Demystified. [Online] 2019. https://towardsdatascience.com/soft-actor-critic-demystified-b8427df61665.

43. Karpathy, Andrej. The Unreasonable Effectiveness of Recurrent Neural Networks. [Online] 2015. https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

44. Yanhui, Chen. A Battle Against Amnesia: A Brief History and Introduction of Recurrent Neural Networks. [Online] 2021. https://towardsdatascience.com/a-battle-against-amnesia-a-brief-history-and-introduction-of-recurrent-neural-networks-50496aae6740.

45. T., Ryan. LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras. [Online] 2020. https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2.

46. Sepp Hochreiter, J□urgen Schmidhuber. Long Short-Term Memory. [Online] 1997. https://www.bioinf.jku.at/publications/older/2604.pdf.

47. Giacaglia, Giuliano. How Transformers Work. [Online] 2019. https://towardsdatascience.com/transformers-141e32e69591.

48. Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung. Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting. [Online] 2015. https://arxiv.org/pdf/1506.04214.pdf.

49. Ashish Vaswani, noam Shazeer, Niki Parmar, Jakob Uszkoreit. Attention Is All You Need. [Online] 2017. https://arxiv.org/pdf/1706.03762.pdf.

50. Alammar, Jay. The Illustrated Transformer. [Online] 2018. https://jalammar.github.io/illustrated-transformer/.

51. Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context. [Online] 2019. https://arxiv.org/pdf/1901.02860.pdf.

52. Sigaud, Olivier. Imitation Learning in RL. [Online] 2022. https://dac.lip6.fr/wp-content/uploads/2022/11/imitation.pdf.

53. Lőrincz, Zoltán. A brief overview of Imitation Learning. [Online] 2019. https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c.

54. Pomerleau, Dean A. ALVINN: AN AUTONOMOUS LAND VEHICLE IN A NEURAL NETWORK. [Online] 1988. https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

55. Sergey Levine, Aviral Kumar, George Tucketr, Justin Fu. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. [Online] 2020. https://arxiv.org/pdf/2005.01643.pdf.

56. Qing Wang, Jiechao Xion, Lei Han, Peng Sun, Han Liu, Tong Zhang. Exponentially Weighted Imitation Learning for Batched Historical Data. [Online] 2018. https://proceedings.neurips.cc/paper_files/paper/2018/file/4aec1b3435c52abbdf8334ea0e7141e0-Paper.pdf.

57. Xue Bin Peng, Aviral Kumar, Grace Zhang & Sergey Levine. Advantage Weighted Rgression: Simple and Scalable Off-Policy Reinforcement Leanring. [Online] 2019. https://arxiv.org/pdf/1910.00177.pdf.

58. Andrew Y. Ng, Stuart Russell. Algorithms fro Inverse Reinforcement Learning. [Online] 2000. https://ai.stanford.edu/~ang/papers/icml00-irl.pdf.

59. Alexander, Jordan. Learning from humans: what is inverse reinforcement learning? [Online] 2018. https://thegradient.pub/learning-from-humans-what-is-inverse-reinforcement-learning/.

60. Aviral Kumar, Sergey Levine. Conservative Q-Learning. [Online] 2020. https://arxiv.org/pdf/2006.04779.pdf.

61. Ziyu Wang, Alexander Novikov, Konrad Zolna. Critic Regularized Regression. [Online] 2021. https://arxiv.org/pdf/2006.15134.pdf.

62. Scott Fujimoto, Shixiang Shane. A Minimalist Approach to Offline Reinforcement Learning. [Online] 2021. https://arxiv.org/pdf/2106.06860.pdf.

63. Daniil Tiapkin, Denis Belometsny, Daniele Calndriello. Demonstration-Regularized RL. [Online] 2023. https://arxiv.org/pdf/2310.17303v1.pdf.

64. Yang Liu, Qi Liu, Hongke Zhao. Adaptive Quantitative Trading: An Imitative Deep Reinforcement Learning Approach. [Online] 2020. https://www.researchgate.net/publication/342537252_Adaptive_Quantitative_Trading_An_Imitative_Deep_Reinforcement_Learning_Approach.

65. Doo Re Song, Chuanyu Yang, Christopher McGreavy, and Zhibin Li. Recurrent Deterministic Policy Gradient Method for Bipedal Locomotion on Rough Terrain Challenge. [Online] 2019. https://arxiv.org/pdf/1710.02896.pdf.

66. Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. *https://www.ray.io/.* [Online] 2018. https://arxiv.org/pdf/1712.09381.pdf.

67. Xiao-Yang Liu, Hongyang Yang, Jiechao Gao, Christina Dan Wang. FinRL: Deep Reinforcement Learning Framework to Automate trading in Quantitative Finance. *https://github.com/AI4Finance-Foundation/FinRL/tree/master.* [Online] 2021. https://arxiv.org/pdf/2111.09395.pdf.

68. Richard Liaw, Eric Liang. Tune: A Research Platform for Distributed Model Selection. [Online] 2018. https://arxiv.org/pdf/1807.05118.pdf.

69. Inc., Quantopian. Pyfolio programming library. *https://github.com/quantopian/pyfolio.* [Online] https://pyfolio.ml4trading.io/.

70. Martin, Robert Andrew. PyPortfolioOpt. [Online] 2021. https://pyportfolioopt.readthedocs.io/en/latest/.

71. Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, Noah Dormann. Stable-Baselines3 - Reliable Reinforcement Learning Implementations. [Online] https://stable-baselines3.readthedocs.io/en/master/.

72. Financial Modeling Prep API. [Online] https://site.financialmodelingprep.com/developer/docs.

73. *Profitability of simple stationary technical trading rules with high-frequency data of Chinese Index Futures.* Jing-Chao Chen, Yu Zhou, Xi Wang. 2017.

74. Optuna Hyperparameters Optimization. [Online] https://optuna.org/.

75. Lisha Li, Kevin Jamieson, Afshin Rostamizadeh, Katya Gonina, Moritz Hardt, Benjamin Recht, Ameet Talwalkar. Massively Parallel Hyperparameter Tuning. [Online] 2018. https://openreview.net/forum?id=S1Y7OOlRZ.

76. Voila Jupyter server extension. [Online] https://voila.readthedocs.io/en/stable/using.html.

77. Scott M. Lundberg, Sun-In Lee. A Unified Approach to Interpreting Model predictions. [Online] 2017. https://arxiv.org/pdf/1705.07874v2.pdf.

78. Chunli Liu, Carmine ventre, Maria Polukarov. Synthetic Data Augmentation for Deep Reinforcement Learning in Financial Trading. [Online] 2022. https://dl.acm.org/doi/abs/10.1145/3533271.3561704.

79. Michael Janner, Qiyang Li, Sergey Levine. Offline Reinforcement Learning as One Big Sequence Modeling Problem. [Online] 2021. https://arxiv.org/pdf/2106.02039.pdf.

80. Zhengyao Jiang, Dixing Xu, Jinjun Liang. A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem. [Online] 2017. https://arxiv.org/pdf/1706.10059.pdf.

81. Zonghan Wu, Shirui Pan, Guodong Long. Connecting the Dots: Multivariate Time Series Forecasting with Graph Neural networks. [Online] 2020. https://arxiv.org/pdf/2005.11650.pdf.

82. Li, Maosen, et al. Online Multi-Agent Forecasting With Interpretable Collaborative Graph Neural Networks. [Online] 2022. https://ieeexplore.ieee.org/document/9728758.

83. Clune, Felipe Petroski Such Vashisht Madhavan Edoardo Conti Joel Lehman Kenneth O. Stanley Jeff. Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. [Online] 2018. https://arxiv.org/pdf/1712.06567.pdf.

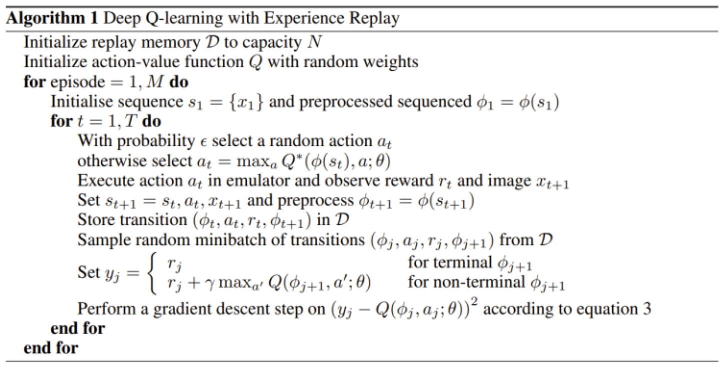
84. Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. [Online] 2017. https://arxiv.org/pdf/1703.03864.pdf.

85. Sharpe, William F and WF Sharpe. *Portfolio theory and capital markets.* s.l. : McGraw-Hill New York, 1970.

Annex

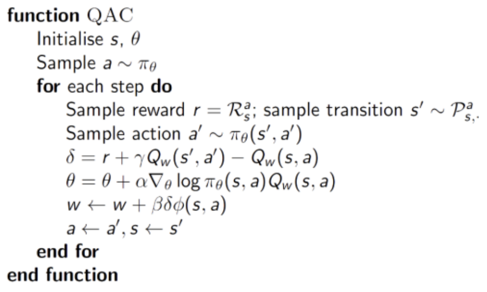
- **3.1.6 Value function Approximation, Experience Replay, policy gradient**

DQN pseudocode (19)



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

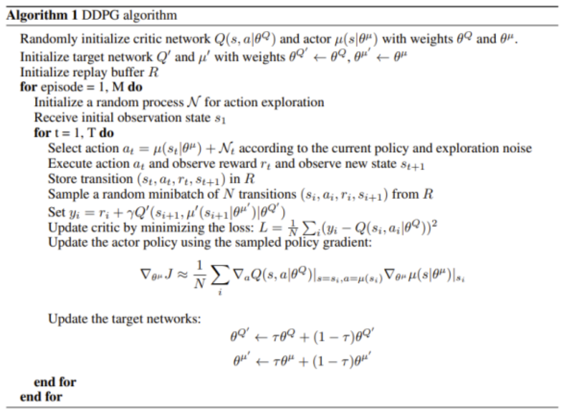
- **3.1.7 Actor-Critics Algorithms** (28)



```
function QAC
    Initialise $s, \theta$
    Sample $a \sim \pi_\theta$
    for each step do
        Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
        Sample action $a' \sim \pi_\theta(s', a')$
        $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
        $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
        $w \leftarrow w + \beta \delta \phi(s, a)$
        $a \leftarrow a', s \leftarrow s'$
    end for
end function
```

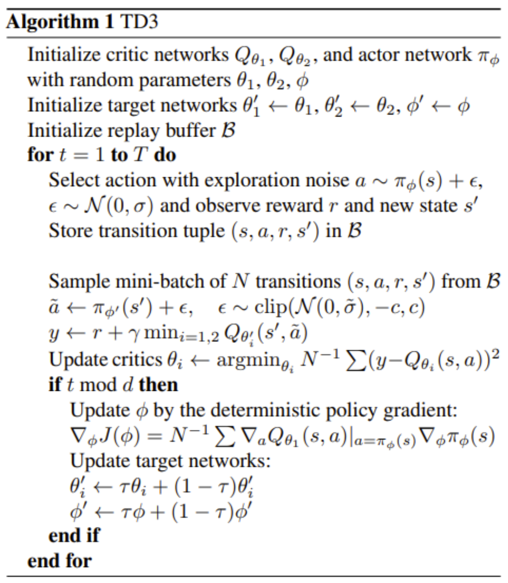
- **3.1.8 State-of-the-Art Deep RL algorithms pseudocode**

Deep Deterministic Policy Gradient (DDPG) (33)



```
Algorithm 1 DDPG algorithm
  Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
  Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
  Initialize replay buffer $R$
  for episode = 1, M do
     Initialize a random process $\mathcal{N}$ for action exploration
     Receive initial observation state $s_1$
     for t = 1, T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
              $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
        Update the target networks:
              $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
              $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
     end for
  end for
```

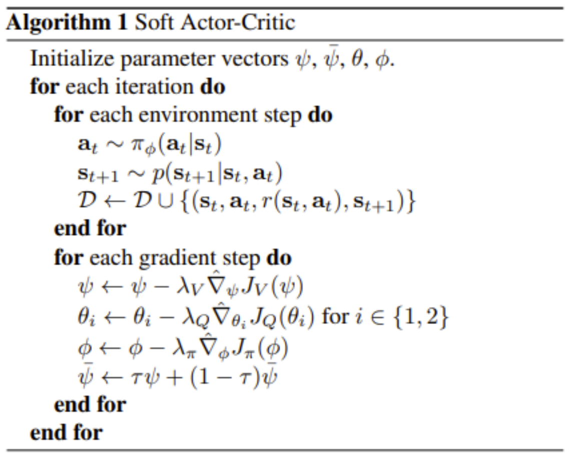
Twin Delayed Deterministic Policy Gradient (TD3) (39)



```
**Algorithm 1** TD3
  Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network $\pi_\phi$
  with random parameters $\theta_1, \theta_2, \phi$
  Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
  Initialize replay buffer $\mathcal{B}$
  for $t = 1$ to $T$ do
    Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
    $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
    Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

    Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if $t$ mod $d$ then
      Update $\phi$ by the deterministic policy gradient:
      $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
      Update target networks:
      $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
      $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
    end if
  end for
```

Soft Actor critic (SAC) (41)

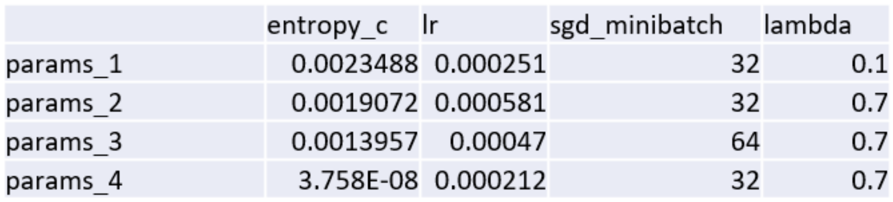


```
**Algorithm 1** Soft Actor-Critic
  Initialize parameter vectors $\psi, \bar{\psi}, \theta, \phi$.
  for each iteration do
    for each environment step do
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
    for each gradient step do
      $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$
      $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
      $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
      $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$
    end for
  end for
```

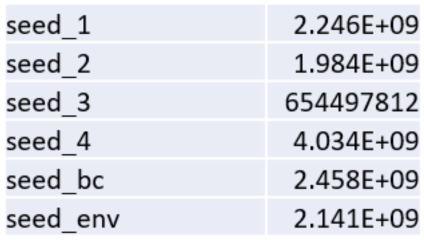
- **4.3 List of DJI30 tickers used**

```
"WMT","DIS","INTC","JNJ","KO","JPM","MCD","MMM","MRK","MSFT","NKE","PG","AXP","AMGN","
AAPL","BA","CAT","CSCO","CVX","GS","HD","HON","IBM","UNH","VZ"
```

- **5.1 Seeds and Hyperparameters sets used for stability considerations of BC-PPO**



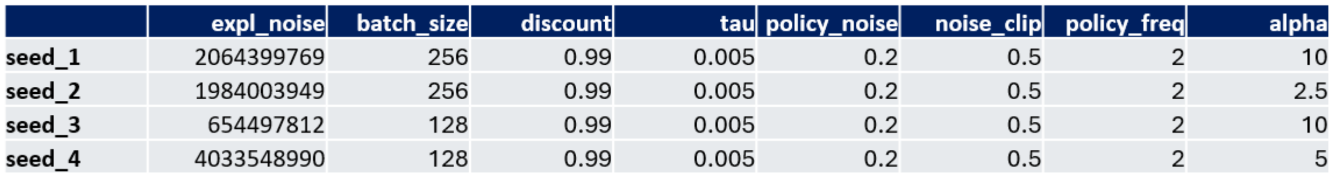
	entropy_c	lr	sgd_minibatch	lambda
params_1	0.0023488	0.000251	32	0.1
params_2	0.0019072	0.000581	32	0.7
params_3	0.0013957	0.00047	64	0.7
params_4	3.758E-08	0.000212	32	0.7



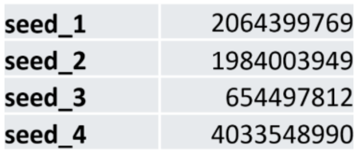
seed_1	2.246E+09
seed_2	1.984E+09
seed_3	654497812
seed_4	4.034E+09
seed_bc	2.458E+09
seed_env	2.141E+09

+

- **5.3.2 Seeds and Hyperparameters sets used for stability considerations of TD3-BC**

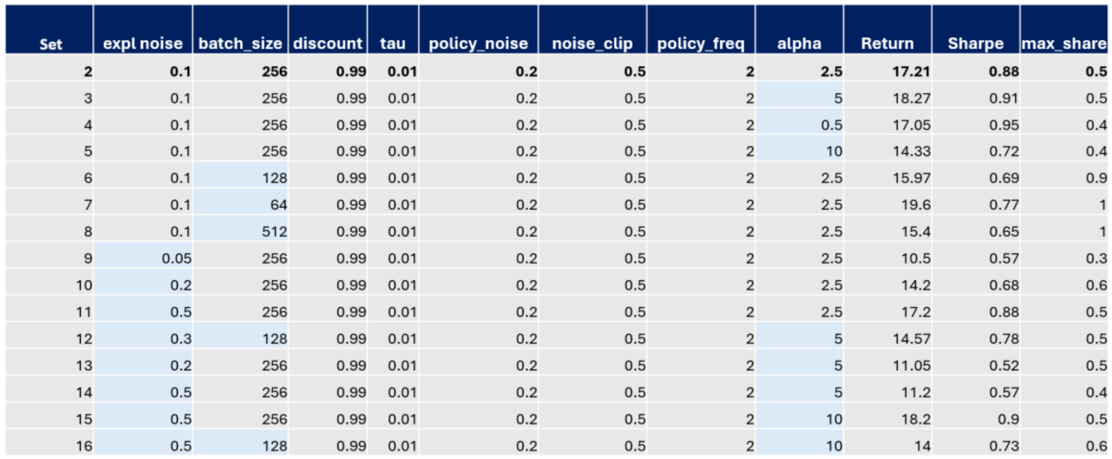


	expl_noise	batch_size	discount	tau	policy_noise	noise_clip	policy_freq	alpha
seed_1	2064399769	256	0.99	0.005	0.2	0.5	2	10
seed_2	1984003949	256	0.99	0.005	0.2	0.5	2	2.5
seed_3	654497812	128	0.99	0.005	0.2	0.5	2	10
seed_4	4033548990	128	0.99	0.005	0.2	0.5	2	5



seed_1	2064399769
seed_2	1984003949
seed_3	654497812
seed_4	4033548990

- **5.3.2 Hyperparameters optimization**



Set	expl noise	batch_size	discount	tau	policy_noise	noise_clip	policy_freq	alpha	Return	Sharpe	max_share
2	**0.1**	**256**	**0.99**	**0.01**	**0.2**	**0.5**	**2**	**2.5**	**17.21**	**0.88**	**0.5**
3	0.1	256	0.99	0.01	0.2	0.5	2	5	18.27	0.91	0.5
4	0.1	256	0.99	0.01	0.2	0.5	2	0.5	17.05	0.95	0.4
5	0.1	256	0.99	0.01	0.2	0.5	2	10	14.33	0.72	0.4
6	0.1	128	0.99	0.01	0.2	0.5	2	2.5	15.97	0.69	0.9
7	0.1	64	0.99	0.01	0.2	0.5	2	2.5	19.6	0.77	1
8	0.1	512	0.99	0.01	0.2	0.5	2	2.5	15.4	0.65	1
9	0.05	256	0.99	0.01	0.2	0.5	2	2.5	10.5	0.57	0.3
10	0.2	256	0.99	0.01	0.2	0.5	2	2.5	14.2	0.68	0.6
11	0.5	256	0.99	0.01	0.2	0.5	2	2.5	17.2	0.88	0.5
12	0.3	128	0.99	0.01	0.2	0.5	2	5	14.57	0.78	0.5
13	0.2	256	0.99	0.01	0.2	0.5	2	5	11.05	0.52	0.5
14	0.5	256	0.99	0.01	0.2	0.5	2	5	11.2	0.57	0.4
15	0.5	256	0.99	0.01	0.2	0.5	2	10	18.2	0.9	0.5
16	0.5	128	0.99	0.01	0.2	0.5	2	10	14	0.73	0.6