Master's thesis
Master's Programme in Data Science

A Reinforcement Learning Application for Portfolio Optimization in the Stock Market

Andres Huertas

June 16, 2020

Supervisor(s): Professor Jussi Kangasharju

Examiner(s): Professor Jussi Kangasharju
Professor Dorota Glowacka

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Andres Huertas

Työn nimi — Arbetets titel — Title

A Reinforcement Learning Application for Portfolio Optimization in the Stock Market

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	June 16, 2020	44

Tiivistelmä — Referat — Abstract

Investment funds are continuously looking for new technologies and ideas to enhance their results. Lately, with the success observed in other fields, wealth managers are taking a closes look at machine learning methods. Even if the use of ML is not entirely new in finance, leveraging new techniques has proved to be challenging and few funds succeed in doing so. The present work explores de usage of reinforcement learning algorithms for portfolio management for the stock market. It is well known the stochastic nature of stock and aiming to predict the market is unrealistic; nevertheless, the question of how to use machine learning to find useful patterns in the data that enable small market edges, remains open.

Based on the ideas of reinforcement learning, a portfolio optimization approach is proposed. RL agents are trained to trade in a stock exchange, using portfolio returns as rewards for their RL optimization problem, thus seeking optimal resource allocation.

For this purpose, a set of 68 stock tickers in the Frankfurt exchange market was selected, and two RL methods applied, namely Advantage Actor-Critic(A2C) and Proximal Policy Optimization (PPO). Their performance was compared against three commonly traded ETFs (exchange-traded funds) to asses the algorithm's ability to generate returns compared to real-life investments. Both algorithms were able to achieve positive returns in a year of testing(5.4% and 9.3% for A2C and PPO respectively, a European ETF (VGK, Vanguard FTSE Europe Index Fund) for the same period, reported 9.0% returns) as well as healthy risk-to-returns ratios. The results do not aim to be financial advice or trading strategies, but rather explore the potential of RL for studying small to medium size stock portfolios.

ACM Computing Classification System (CCS):
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

layout, summary, list of references

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1. Introduction

Reinforcement learning has become one of the hotspots in the modern developments of machine learning. Since the important breakthroughs achieved by research groups such as DeepMind and OpenAI, RL has been proved useful for solving complex problems, ranging from board games such as chess and Go, to robotics and control theory tasks. It is reasonable to think that the modeling capabilities displayed by RL can be extrapolated to other fields, going beyond games and control theory. The purpose of the present report is to explore RL algorithms for portfolio optimization, seeking an answer to the question: Can RL agents learn market patterns and take advantage of them to make a profit? The present report focuses on model-free RL methods, where we do not seek for modeling or predicting stock prices itself but rather training agents to interact with the market in a way that is profitable in the long term.

An increasing number of funds is beginning to include machine learning techniques as part of their tools for wealth management, and large firms count with vast amounts of historical data to train their models, and so I better understanding on how such methods works is needed, not only by firms seeking to increase their profitability but for the general public looking for better understanding on how modern stock markets works.

The present report explores the possibility of using RL to manage a small size portfolio of stocks, training agents to make profits by learning from interacting with a trading environment, and observing returns as a reward signal.

The report is structured as follows:

- **Chapter 2: Reinforcement learning** This chapter offers an overview of the definitions, concepts, and algorithms used in RL. From basic concepts such as trajectories, rewards, and policies; to the general description of state-of-the-art methods such as A2C, TRP, and PPO, this chapter aims to provide a strong enough background on RL in general.

- **Chapter 3: Portfolio managements** Here the problem of portfolio management is introduced. Important finance concepts needed to understand the portfolio optimization problem. Concepts such as returns, risk-to-return ratios,

maximum drawdown, portfolio vectors, etc. This chapter also introduces three exchange trade funds (ETFs) that are going to be used as benchmarks, with the idea of drawing comparisons between the algorithms and well-established funds.

- **Chapter 4: Experimental setup** This chapter describes the datasets, the methodology, and results. As an important result we highly the algorithms ability to capture the overall market behavior, as well as exhibit similar performance to the ones reported by much larger funds (ETFs)

- **Chapter 5: Conclusions** Final thoughts and a summary of the results are provided. A general conclusion is a positive answer to our research question, the RL methods tested were able to pick up market signals and build profitable portfolio trajectories. However, the current approach has important limitations that limit it's a direct application to a real-life case and the findings presented can be taken as financial advice.

2. Reinforcement learning

Over recent years, few subfields of the machine learning domain have received recognition and press highlights as reinforcement learning(RL). There are many reasons why researchers and artificial intelligence enthusiasts display interest in the topic. The basic idea that a system can learn complex behaviors and solve difficult tasks while providing a minimum amount of prior information is at the very least intriguing. In a few words that is the main promise of RL. Self-teaching agents that can learn from environments (real-life or simulated) in a fashion similar to how humans experience, by obtaining rewards and punishments out of the decisions made.

As the recent milestones achieved by RL algorithms it is possible to name a few: DeepMind's AlphaZero, the AI capable of defeating the world champion in the game of GO as well as surpassing by far the existent chess engines [17]; Taking a step further, OpenAI introduced a multiplayer agent named FIVE, trained to play against human teams in the multiplayer game of DOTA2 [1] This set an important milestone for RL applied to videogames, such algorithms proved that RL can be used even to solve complex problems characterized for longtime horizons for rewards, continuous actions, and quick change of strategy of the enemies. The main downside of such systems is the immense amount of computational power and data required for training.

Large scale OpenSource RL systems have been under development the recent years, betting on increasing the community around RL. Facebook's Horizon [2] (recently renamed ReAgent) aims to ease the training and deployment of large scale RL systems. Other resources such as TensorFlow Agents [16] or stable Baselines [6] have certainly lowered the entry barrier for using and experimenting with complex RL algorithms.

By now we are aware of the capabilities of RL in general, let's take a step back into the formal definition of the RL problem and other necessary terminology to understand the entire picture.

The most important components in any RL algorithm are the **agent** and the **environment**. Most RL problems are also framed in a time-step setting. Each time step two fundamental things occur: The agent observes the environment state and takes an action, and the environment reacts to such action and returns a **reward**

to the agent. To begin introducing some notation for the terminology used: actions are named as a_t, the states observed by the agent and the reward obtained s_t and r_t respectively.

The reward obtained by the agent is a real number that reflects how good or bad was the action the agent chooses to improve the state of the environment. The fundamental goal of any RL algorithm is to teach the agent how to **maximize the cumulative reward** that is the sum of the rewards obtained by interacting with the environment for a fixed time period:

$$CR = \sum_{t=0}^{T} \gamma^t r_t \tag{2.1}$$

The discount factor, γ is number in the interval $(0, 1]$ that quantifies how important are past rewards (rewards obtained during the firsts interactions with the environment) in the computation of the total cumulative reward.

The reward signal is fundamentally important in the RL. The feedback from the reward function is the only thing the agent has to guide it's learning process. The research of smarter and more adequate reward functions is called **reward function design or engineering**.

It is perhaps the simplicity of the setting and the wide variety of problems that can be framed within the RL schema what makes it so appealing. Convolutional neural networks and deep learning entailed a similar philosophy, along the lines of "just make the network deeper" or "we need more data to improve performance", that is somehow the hope, that a simplistic enough set of rules(or billions of data points) is enough to model and understand complex problems. RL abstract away the nature and complexity of the data, environments can be anything from physics simulations, text generation to robotic interfaces; and on the other hand agents can also have many faces. However, as always the devil is in de details and a careful understanding of the subtleties within RL is important to succeed at training agents. No amount of data or *smart* algorithm can fix a fundamentally flawed problem.

2.1 Definitions

To jump into the details it is necessary to define (and expand on the ones already introduced) some other important concepts. The following subsections dive deeper into the concepts of action, states, and observations as well as introduce the concepts of policies, trajectories, RL as an optimization problem and value functions.

2.1.1 States and observations

A state is defined as the complete description of the environment at some point in time, often noted as s_t, and an **observation** , noted as o_t is the information available for the agent to see (and learn from). There are cases where the observation and the state are the same. In a game of chess, the state of the game is the location of each piece on the board, and that information is also available for both players (it doesn't matter if the player is human or an artificial agent). In a multiplayer videogame, the state of the game is defined by a lot of variables that are not accessible for a given player. A player has access only to his data (for instance position in a map, character health, available spells, etc.). Such environments are often called **partially observed**.

The mathematical nature of the states and observations is not fixed. In practice, the states and observations are often tensors. And in turn, the tensors can be discrete or continuous.

The same problem can have many possible state representations. If we look back at the chess example one first idea that could encode the board information into a tensor would be to use 8x8 matrix and specific numbers for each piece. At first sight, this representation encodes all the information required to understand the game (and it does indeed) but is it enough for a RL algorithm? or how easy is it to extract from it relevant information for appropriated decision making? Answer these questions a priori is not an easy task, and often experimentation is required. The state representation should aim to present the information to the agent in such a manner that the agent does not need to waste time learning obvious(or not useful) facts about the data. Again going back to the chess example if the number 2 represents a piece and the number 4 represents another piece, the agent could get lost trying to make sense that the second piece is not "double" the second one, the difference between the piece "2" and the piece "3" is not the same as the difference between "2" and "1". The naive state representation is correctly representing the board, but there are a lot of implied relationships that the agent would need to learn before learning to play good chess moves.

There are other tradeoffs to be considered when choosing state representations. Once again using chess as an example, the AlphaZero engine [17] uses as state representation a $NxNx(MT + L)$ binary matrix; where $N = 8$ board size, $T = 8$ lookback window, M is the total different types of pieces(black pawn, white pawn, etc) and L constant matrices signaling additional chess rules. This representation allows us to feed the agent with much more rich information to learn from but is also significantly more memory intensive. The main take away is, of course, there is no silver bullet, and what is going to work for a specific problem may depend on the algorithms used (do

this algorithm likes discrete representations?) the agent's nature (is the model behind the agent able to learn, for instance, nonlinear relationships present in the data?)

2.1.2 Policies

This one speaks for itself. A policy is a rule followed by an agent to take actions on the environment. Often noted with μ for deterministic policies or π when dealing with stochastic ones. In practice, policies are parametrized (a neural network for instance) so it is common to subscript them with the letter θ :

$$a_t = \mu_\theta(o_t) \tag{2.2}$$

$$a_t = \pi_\theta(.|o_t) \tag{2.3}$$

The policy being actual desition maker in the RL framework it's often used interchangeable with the agent term, in phrases like: "The agent/policy tries to maximize the reward"

Deterministic policies directly predict the action that should be taken by the agent. In general, a stochastic policy predicts the parameters for another probabilistic model from which actions can be sampled. In a sense deterministic policies are easier to understand, a simple feed-forward neural network can be a deterministic policy, the same input the same output. Stochastic policies require a bit more care, to be understood and trained. There are two operations to be taken into account when talking about Stochastic policies, first actions can be sampled from a given policy, and last, likelihoods of particular actions computed. Later chapters will provide examples of how the two types of policies are used in various algorithms.

2.1.3 Trajectories

Easily enough a trajectory is a sequence of action-state pairs:

$$\tau = (s_0, a_0, ..., s_t, a_t) \tag{2.4}$$

The states evolve according to the laws of the environment (physics simulation behaviour, a game of chess, robot movement) and often the depency with the action is tied only to the most recent action taken by the agent (in some cases the agent is unable to actually change the environment behaviour as it wil be shown when discussing the stock market and perfect market hypothesis)

$$s_{t+1} = f(s_t, a_t) \tag{2.5}$$

$$s_{t+1} \sim P(.|s_t, a_t) \tag{2.6}$$

Where the stochasticity comes from the fact that actions are sampled from a stochastic policy.

At the end of each trajectory is the cumulative reward the agent aims to maximize

The notion of reward function was already introduced. A real number that quantifies how good or bad is the performance of an agent. In a game of chess, a reward could be the advantage in piece material, in a physics simulation such as the classic cartpole [11] example the agent receives +1 every time step the pole remains up a certain threshold.

Rewards can be written taking into account the explicit dependency of the current state, the action taken by the agent, and the future state of the environment:

$$r_t = R(s_t, a_t, s_{t+1}) \tag{2.7}$$

Rewards are the only feedback the agent has from the environment. An inappropriate choosing on the reward function can lead to poor performance.

Here we run into a situation that resembles the choosing of numerical state representations that successfully convey the information the agents need. A reward function poorly crafted might guide the agent to learn suboptimal behavior or *hack the system* to obtain more rewards instead of solving the task.

An example could be as follows: A system where a robotic arm is trained to stack blocks on top of each other. how do we reward the agent to achieve such a goal? one approach could be simply a positive reward for each block successfully stacked. That reward perfectly describes the goal, and maximizing it would lead to solving the problem but it is hard to know a priori if the feedback would be enough to guide the agent. If the agent is learning without any previous pretraining, the first trajectories sampled from any policy would follow random behaviors, and successfully stock pilling two blocks by taking random moves is unlikely, and so the first trajectories will contain little feedback that the agent can use for learning. This problem appears when dealing with sparse rewards [4], what it is important is the proportion on trajectories that end up with no reward vs the ones that succeed at the task.

It is not to say that spare rewards should not be considered. Once again the AlphaZero milestone works as an example since in there the agent was trained by receiving a reward only at the end of each match (+1 for winning, 0 for a draw and -1 for losing)

Complex environments may benefit from more progressive rewards. In the robot arm example, perhaps it is worth training the robotic hand to learn first how to grab

a block, then lift it on top of another block and lastly drop it. Agents can learn to play the environments and solve the tasks in unpredictable ways, and agents rewarded for stacking blocks could be encouraged to stack a block take it down, and re-stack it forever. Rewards should also make clear when the environment has reached a terminal state.

2.1.4 The RL problem

The formal objective of the RL problem is train an angent to interact with the environment such that it's actions **maximize the expected cummulative reward**. Cummulative reward, along side with the discount factor γ, were defined in (2.1).

Suppose there is a trajectory τ, the probability of observing such trajectory is given by the current policy followed by the agent and the probability of the initial state:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_t + 1|s_t, a_t)\pi(a_t|s_t) \tag{2.8}$$

Then the cumulative reward $R(\tau)$ for the trajectory can be computed and the expected cumulative reward defined:

$$E_{\tau\sim\pi}[R(\tau)] = \int_t P(\tau|\pi)R(\tau) \tag{2.9}$$

Then what any RL algorithm is looking for is a policy that maximize (2.9), the optimal policy :

$$\pi^{opt} = arg\,max_\pi E_{\tau\sim\pi}[R(\tau)] \tag{2.10}$$

Everything boils down to finding π^{opt}. One integral part of any RL algorithm is the assessment of policies and states, this is often done via value functions that are to be defined in the next subsection.

2.1.5 Value functions

The **value** of s state (or state, action) pair is defined as the expected return if the system is initialized in that particular state and the agent follows a given policy. There are two possible definitions for the value function depending on if the first action taken by the agent comes or not from the given policy.

The **Value function** is formally defined:

$$V^\pi(s) = E_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.11}$$

Sometimes a small addition is done to the definition (2.11) to define the value function for a state-action pair. In this case, the value is the expected cumulative reward starting from initial state s, taking a random action a (which may not come from any specific policy) and then acting according to the policy π.

$$Q^{\pi}(s,a) = E_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.12}$$

Some algorithms might benefit from using an alternative formulation, instead of using the value and action-value, the **advantage** function is defined as follows:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.13}$$

Finally, the **optimal value function** and the the **optimal action-value function** are defined as before but when the policy used, is the optimal policy 2.10

$$V^{*}(s) = E_{\tau\sim\pi^{opt}}[R(\tau)|s_0 = s] \tag{2.14}$$

$$Q^{*}(s,a) = E_{\tau\sim\pi^{opt}}[R(\tau)|s_0 = s, a_0 = a] \tag{2.15}$$

These action value equations satisfy a special set of equations called the **Bellman equations**. Bellman equations state that the value of the certain initial state is the expected reward from starting there (the action-value function) plus the value of wherever the current policy takes you in the next step.

Mathematically:

$$V^{\pi}(s) = E_{a\sim\pi,s'\sim P}[r(s,a) + \gamma V^{\pi}(s')] \tag{2.16}$$

$$Q^{\pi}(s,a) = E_{s'\sim P}[r(s,a) + \gamma E_{a'\sim\pi}[Q^{\pi}(s',a']] \tag{2.17}$$

Similar equations can be written for optimal value and action-value functions. The next section will provide an overview of how optimal policies can be found using a gradient ascent algorithm.

2.1.6 Policy optimization

It is important ot keep in mind the basic problem: find the parametrized policy π_θ such as the expected return $J(\pi_\theta) = E_{\tau\sim\pi_\theta}[R(\tau)]$ is maximized. This is known as policy optimization.

Policy optimization is normally done via gradient ascent, this means the parameters are updated according to the following equation:

$$\theta_{t+1} = \theta_k + \alpha \nabla J(\pi_\theta)|_k \tag{2.18}$$

With α being the classic learning rate of gradient-based algorithms. The expression $J(\pi_\theta)$ is called the **policy gradient**. The next step is to find a way to compute the policy gradient. Since what it is possible to measure are the agent-environment interactions, a way to compute the policy gradient from (a finite) number of such interactions is needed.

From (2.8) it is possible to consider a parametrized policy:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_t + 1|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2.19}$$

The gradient for (2.19) can be computed as:

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta)\nabla_\theta log P(\tau|\pi_\theta) \tag{2.20}$$

And the term inside inside the ∇ operator (that is the log probability for the given trajectory) can be computed as follows:

$$log P(\tau|\pi_\theta) = log\rho_0(s_0) + \sum_{i=0}^{T} (log(P(s_t + 1|s_t, a_t)) + log(\pi_\theta(a_i, s_i))) \tag{2.21}$$

Combining the expressions and from the fact that the environment does not depend on the parametrization same as well as the initial state:

$$\nabla_\theta log P(\tau|\pi_\theta) = \sum_{i=0}^{T} \nabla_\theta log \pi_\theta(a_t, s_t) \tag{2.22}$$

Now it is possible to plug everything together to derive an expresion for the gradient $\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\sim\pi_\theta}[R(\tau)]$:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \int_\tau \nabla_\theta E_\theta[R(\tau)] && (2.23)\\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) && (2.24)\\
&= \int_\tau P(\tau|\theta)\nabla_\theta log P(\tau|\theta)R(\tau) && (2.25)\\
&= E[\sum_{i=0}^{T} \nabla_\theta log \pi_\Theta(a_t, s_t)R(\tau)] && (2.26)
\end{aligned}$$

Since the final expression in (2.26) is an expected value, it means that is possible to build an estimation for it collecting a set of trajectories D by letting the agent act according to the policy π_θ and computing the mean for (2.26) :

$$grad = \frac{1}{|D|} \sum^{\tau} \sum_{i=0}^{T} \nabla_\theta log \pi_\theta(a_i, s_i) R(\tau) \tag{2.27}$$

Equation (2.27) is the simplest version of the policy gradient and provided that the gradient for the policy $\nabla_\theta log \pi_\theta(a_i, s_{,i})$ is computable (a neural network for example) all that is needed is collect trajectories, sampling for the policy and update the policy parameters until convergence.

More complex algorithms aim to find smarter ways to update the policy gradients. Those methods are the topic for the next section. Starting with Deep-Q learning one of the pioneer methods, introduced by DeepMind [10]. It is not an overstatement to say that the paper launched an entire field in an unprecedented race. An overview of more advanced policy gradient methods (those aiming to maximize the expected value via gradient ascent and policy gradients), methods such as Trust Region Policy Optimization, Deterministic Policy Gradient, and Soft actor-critic will be presented.

2.2 Deep Q learning

So far we laid the foundation of policy gradient algorithms. Deep Q-learning uses other ideas. Q learning aims to find the optimal action-value function leveraging the Bellman equations and a deep learning approximation for the Q function.

In (2.16) and (2.17) the Bellman equations for the given policy π were given. The Bellman equations for the optimal value-action functions are:

$$V^*(s) = \max_a E[r(s,a) + \gamma V^*(s')] \tag{2.28}$$

$$Q^*(s,a) = E_{s' \sim P}[r(s,a) + \gamma \max_{a'} Q^*(s', a')] \tag{2.29}$$

The main change is the inclusion of the max operator indicating that to act optimally, the agent should pick the action that yields the highest reward. The idea behind the Q learning algorithms is to estimate the value-action function $Q*$ by iterative updates:

$$Q_{i+1}(s,a) = E[r + \gamma \max_{a'} Q_i(s', a')] \tag{2.30}$$

In (2.30) the function Q_i approximate the optimal function Q^* as the number of steps tends to infinity [18]. This method doesn't work in practice, a common solution

is using a parametrizable estimator for the value-action function: $Q(s, a, \theta) \sim Q^*(s, a)$. Such an estimator can be a deep neural network whose parameters are ought to be learned. As is common for neural networks a loss function is required to train the network, in this case, the loss function comes from the Bellman equation (2.30) minimizing the difference between the left and right side of the equation:

$$y_i = E[r(s, a) + \max_{a'} Q(s', a', \theta_{i-1})] \tag{2.31}$$

$$L_i(\theta_i) = E[(y_i - Q(s, a, \theta_i))^2] \tag{2.32}$$

This is somehow similar to the classic minimization problem for mean squared error, with the particularity that the target of the network (y_i) depends on the network's previous weights.

The final element is a expression to compute the gradient for (2.32) :

$$\nabla_\theta L_i = E\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})]\right) - Q(s, a, \theta_i)\nabla_\theta Q(s, a, \theta_i)\right] \tag{2.33}$$

The gradient can be estimated from a sample of sequences i, and then a classic gradient descent step computed to update the weights. The last conceptual component for the Deep Q-learning algorithms is the system of experience replay. In such a system the agent's interactions with the environment are stored in a dataset $\mathcal{D}$, during the training phase of the experiences are drawn at random from the subset $\mathcal{D}$ and the Q-learning updates applied. In Q-learning the exploration-exploitation tradeoff is handled by using an ϵ greedy policy (the agent has a ϵ probability of selecting a random action)

The Q-learning is what is known as model-free algorithm; Q-learning solves the optimization problem without modeling the environment, opposite to model-based algorithms whose goal is to use a model to build an understanding of the world the agent is interacting with, in order to maximize the rewards, models as Monte Carlo Tree Search [8] and Dyna-Q, a variant of Q learning method that also trains a model for the environment [12].

The original research [10] introduced the Deep Q-learning for solving Atari games, surpassing (at its time) every other RL algorithm and human performance, sparking developments in the field. Later variants such as the Double Q-learning [19] were introduced to solve issues present in the original deep Q-learning formulation. Namely, the double Q-learning algorithm aims to deal with the overestimation for the values given by the Q network by introducing a second network with different parameters θ', during the training phase one network is used to determine the actions and the other

to compute the values. This can be written as :

$$y_i = E[r(s,a) + \gamma Q(s', \max Q(s', a; \theta_i); \theta_i')] \tag{2.34}$$

From this equation is clear that the actions are drawn from the network parameterized by θ_i, and the value estimation used for the optimization steps is computed from the network parameterized by θ_i'. The parameters θ and θ' are trained symmetrically, alternating the role of the networks. Expression (2.34) is ought to replace y_i in equation (2.32)

The following sections go back to the policy gradient methods and how to deal with some of the limitations presented in Q-learning algorithms. Policy gradient methods have proved to be useful for dealing with continuous action spaces, this is ought to be useful when addressing the problem of portfolio optimization as it will be shown in section 3.

2.3 Policy gradient methods

All the following methods are extensions to the vanilla policy gradient algorithm derived in the previous chapter. First, we will discuss how the Trust region policy optimization and Proximal Policy Optimization algorithms try to take the largest step possible in the direction of the gradients without destroying the performance. Next, the soft actor-critic and actor-critic methods will be discussed.

2.3.1 Trust Region Policy Optimization

The main idea of the Trust Region Policy Optimization(TRPO) is to provide a guarantee monotonic improvement while taking non-trivial step sizes.

Large step sizes can hurt performance in gradient descent algorithms and small steps lead to slower convergence times. TRPO solves this by taking the largest step possible while satisfying a constraint on how different the new policy and old policy are allowed to be. Such constrain is determined by the KL-divergence between policies.

The other important definition in TRPO is the use of a surrogate loss function instead to optimize directly the expected cumulative returns.

The surrogate loss function is defined as:

$$\mathcal{L}(\theta_k, \theta) = E_{s,a\sim\pi_{\theta_k}}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)\right] \tag{2.35}$$

Where π_θ and π_{θ_k} are the new and old policy respectively. And the advantage function is defined as the difference between the $Q_\pi(s,a)$ and the value function $V_\pi(\pi)$,

given by the Bellman equations (2.16) and (2.17)

It can be shown that policy updates that have a non-negative expected advantage value at all possible states areg guaranteed to increase the policy performance, that is the policy updates in TRPO satisfy:

$$\sum_a \pi_\theta(a|s) A_\pi(s,a) > 0 \tag{2.36}$$

The equation holds for the original expected discounted reward $J(\pi_\theta)$. In the works by Kakade & Langford [7] a relationship between the $J(\pi_\theta)$ and the surrogate function was derived:

$$J(\theta) > \mathcal{L}(\theta_k, \theta) - \frac{2\epsilon\gamma}{(1-\gamma)^2}\alpha^2 \tag{2.37}$$

Where $\epsilon = \max\left(E_{a\sim\pi_{\theta_k}}[A_\pi(s,a)]\right)$, and α a parameter that quantifies the mixture of policies: $\pi'(a,s) = (1-\alpha)\pi_{\theta_k}(s,a) + \alpha\pi_\theta(s,a)$

This condition however, proved to be too restricted in practice. It can be shown that the second term in the righthand side of the equation can be written using the KL-divergence:

$$J(\theta) > \mathcal{L}(\theta_k, \theta) - CKL(\pi_{\theta_k}, \pi_\theta) \tag{2.38}$$

With C a constant determined by ϵ and γ.

Then the optimization problem to be solved can be written as:

$$\theta_{k+1} = \arg\max_\theta \ \mathcal{L}(\theta_k, \theta) \text{s.t. } \bar{D}_{KL}(\theta||\theta_k) \leq \delta \tag{2.39}$$

The DKL-divergence defined as the average of the KL-divergenge across the states sampled by using the old policy:

$$\bar{D}_{KL}(\theta||\theta_k) = E\left[D_{KL}\left(\pi_\theta(\cdot|s)||\pi_{\theta_k}(\cdot|s)\right)\right] \tag{2.40}$$

From the theoretical point of view this is enough for the algorithm to work, but from the practical implementation the TRPO update expression it is hard to work with, and so the authors propose to work with a second order aproximations:

$$\mathcal{L}(\theta_k, \theta) \approx g^T(\theta - \theta_k) \tag{2.41}$$

$$\bar{D}_{KL}(\theta||\theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \tag{2.42}$$

Thus resulting in an alternative optimization problem. Given the Taylor approximation, the update rule for the policy parameters needs to be modified. It is important

to remember that one of the keys of the TRPO algorithm is the KL-divergence constraint, and so any gradient update must fulfill it. Details on the derivation of the final update equation can be found in the works by Kakade et al. [7] for a theoretical background and the final building blocks for practical application developed by Schulman et al. [14].

2.3.2 Proximal Policy Optimization

TRPO achieves its goal, namely, taking large steps in the parameters space while keeping the policies close enough to avoid the collapse of the model. However, it does so by introducing a surrogate function, introducing a second-order approximation to solve another optimization problem. Proximal Policy Optimization (PPO) aims to do the same by carefully clipping the objective function to penalize parameter updates that land the new policy far from the old one [15]. PPO methods retain the advantages of TRPO but it is simpler and as shown in the works by Shuman et al. [15] and Heess et al. [5] PPO shows, empirically, better overall performance.

The policy update equation in PPO is given by the following equations:

$$\theta_{k+1} = \arg\max_{s,a\sim\pi_\theta} E\left[L(s,a,\theta,\theta_k)\right] \tag{2.43}$$

$$L(s,a,\theta,\theta_k) = \min\left(\frac{\pi_\theta}{\pi_{\theta_k}} A^{\pi_{\theta_k}}(s,a)), g(\epsilon, A^{\pi_{\theta_k}}(s,a))\right) \tag{2.44}$$

Where the A the function g is defined as:

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0 \end{cases} \tag{2.45}$$

Where ϵ is a hyperparameter that controls how far is the new policy allowed to land from the old policy. As usual, a solution for the optimization problem is found using many gradient ascent steps. .

2.4 Asynchronous Advantage Actor-Critic and Soft Actor-critic

The final set of methods to be presented are the family of algorithms known as actor-critic methods.

The **Asynchronous Advantage Actor-Critic** (A3C) was proposed by the DeepMind team in 2016 [9], providing serious improvements over the RL algorithms

published by that time. A3C proved better than DQ-learning for solving the Atari 2600 environment problems, whilst also being able to train at a fraction of the computational cost of training DQ networks; by proposing a naturally parallelizable framework, A3C can use multicore CPUs instead of relying on GPUs. A3C discards the use of replay memory buffers as done in previous off-policy learning algorithms, instead, it replaces a single agent interacting with an environment by a handful of different agents (that are in nature the same, but potentially differently parametrized) interacting with different instances of the environment.

This achieves two fundamental things: the entire workload can be distributed among agents, and thus the learning process speedup; and since the agents are interacting independently from each other, the experience to learn from is more diverse and therefore the generalization capabilities of the algorithm, enhanced. This is where the **Asynchronous** in the title cames from.

The actor-critic duo is made from a policy function (that is the actor or agent part) and a critic function, in the case of the A3C algorithm an estimate for the value function (2.11) is used as the critic. The critic is trained to be able to tell how good is a given state (in terms of the expected reward).

Finally, the advantage term comes from the update rule used:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta \log \pi_\theta(a_k, s_k) A(a_k, s_k, \theta') \tag{2.46}$$

Where $A(a_k, a_k)$ is the advantage function as defined in (2.13), with the twist of using an *Advantage estimate* and replace the Q^π for the discounted reward. The value function is the output from a neural network.

The algorithm begins with a global network for both the policy and the value function, the global parameters can be noted as θ and θ_v. Each actor operates in a separated thread in the CPU, and stores it's own set of parameters θ' and θ'_v. At the begiing of each training iteration, each actor syncronizes it's weights with the weights from the global networks: $\theta' = \theta$, $\theta'_v = \theta_v$.

The actor then interacts with the environment following the policy given by $\pi_{\theta'}$ collecting rewards as usual until a terminal state is reached. The gradients for $\nabla_{\theta'}$ and $\nabla_{\theta'_v}$ are accumulated and updated, the gradients for θ' are updated according to (2.46) and θ'_v by solving the optimization problem given loss function:

$$\mathcal{L} = (R - V^{\pi_{\theta'_v}})^2 \tag{2.47}$$

In actor-critic methods regularization as well as control over the exploration-exploitation behavior of the agents is achieved by including the policy **entropy** in

the policy loss function.

$$\mathcal{L}_\theta = \log \pi_\theta(s,a) A(a_k, a_k) + \beta H(\pi_\theta) \tag{2.48}$$

The hyperparameter β controls how strong is the entropy term. Finally, the agents perform an asynchronous update to the global parameters. The agents do not need to communicate with each other since they are sharing information via the global network, and the global network encompasses the learning experience of all the agents.

Mnih et al. [9] propose a general framework, not restricted to the actor-critic algorithms. It is possible to develop asynchronous Q-learning and Sarsa [13] algorithms.

2.4.1 Soft actor-critic

The last method to be reviewed is the soft actor-critic. The key feature of the algorithm is the use of the entropy term not only as regularization but to maximize the tradeoff between the expected reward and the entropy. That is soft actor-critic aims to train agents that perform the best possible whilst acting as random as possible.

The optimal policy is defined as the solution for the maximization problem:

$$\pi^* = \arg\max_\pi E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \alpha H\left(\pi(\cdot|s_t)\right)\right)\right] \tag{2.49}$$

Every time step the agent is rewarded by an amount proportional to the policy's entropy H. The basic RL framework needs some modifications to work with this addition; the value and action-value functions need to be changed accordingly:

$$V^\pi(s) = E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \alpha H(\pi))\right] \tag{2.50}$$

$$Q^\pi(s,a) = E_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) + \alpha \sum_{t=1}^{\infty} \gamma^t H(\pi)\right] \tag{2.51}$$

Bellman equation should also be written accordingly:

$$Q^\pi(s,a) = E_{s'\sim P, a'\sim\pi}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') - \alpha \log \pi(a'|s')\right)\right] \tag{2.52}$$

Actions and states (s,a) are sampled from the replay buffer, whereas a' are sampled from the currently available policy. expected value (2.52) can be estimated as usual using a finite number of samples.

There are different variants of how the soft-actor critic is trained. In the paper by Haarnoja et at. [3] two methods were proposed; first, a method where the value, Q and policy functions are trained concurrently; and second, a method where 2 Q-networks

are trained separately and the minimum value is used to build the estimator for the value gradient.

The value function is trained to minimize the following squared loss:

$$\mathcal{L}_\psi = E_{s\sim\mathcal{D}}\left[\frac{1}{2}(V_\psi(s) - E_{a\sim\pi_\phi}\left[Q_\theta(s,a) - \log\pi_\phi(a|s)\right])^2\right] \tag{2.53}$$

This expression tells us that we aim to minimize the difference (across a replay buffer $\mathcal{D}$) between the prediction of the value network V_ψ and the expected value of the Q-network plus the entropy, represented by the negative log value of the policy function.

The gradient for this expression can be estimated :

$$\nabla_\psi\mathcal{L}_\psi = \nabla_\psi V_\psi(V_\psi - Q_\theta + \log\pi_\phi) \tag{2.54}$$

Similarly for the optimization equations for the Q-networks are given by:

$$\mathcal{L}_\theta = E\left[\frac{1}{2}(Q_\theta - \hat{Q})^2\right] \tag{2.55}$$

$$\hat{Q}(s,a) = r(s,a) + \gamma E_{s_{t+1}\sim\pi_\psi}\left[V_{\hat{\psi}(s_{t+1})}\right] \tag{2.56}$$

$$\nabla_\theta\mathcal{L}_\theta = \nabla_\theta Q_\theta(a_t,s_t)(Q_\theta(a_t,s_t) - r(s,a) + \gamma V_{\hat{\psi}}(s_{t+1})) \tag{2.57}$$

The minimization step aims to minimize the difference between the value of the network and the reward function $r(s,a)$ plus the expected value of the following step $t+1$. Another thing to notice is that the value function used here is parametrized by $\hat{\psi}$, this new set of parameters is the moving average of the original ψ of parameters, this change is introduced to help stabilize the training process.

The only piece missing is the optimization equation for the policy parameters:

$$\mathcal{L}_\phi = E_{s\sim\mathcal{D}}\left[\mathrm{D}_{KL}\left(\pi_\phi||\frac{\exp Q_\theta(s_t,a_t)}{Z_{\theta(s_t)}}\right)\right] \tag{2.58}$$

the Kullback-Leibler Divergence appears once more, in this case, to signal that the policy parameters are updated so that the expected difference between distributions (the policy π and the normalized Q_θ. In general, the expression (2.58) is untractable due to the partition function Z_θ and so, to compute the gradient, a reparameterization trick is introduced where the actions are writes as a function depending on the policy parameters:

$$a_t = f_\phi(\epsilon_t, s_t) \tag{2.59}$$

And the gradient for (2.58) :

$$\nabla_\phi \mathcal{L}_\phi = \nabla_\phi \log \pi_\phi + (\nabla_{a_t} \log \pi_\phi - \nabla_{a_t} Q) \nabla_\phi f_\phi(\epsilon_t, s_t) \tag{2.60}$$

As said before, one idea to mitigate the bias induced by overly optimistic aproximation given by the Q_θ function it is to use two Q-networks parametrized by different parameters θ_1 , θ_2 and selecting the minimum of them when computing the expresions (2.60) and (2.54).

3. Portfolio management

The present work focuses on the application of reinforcement learning (RL) methods to portfolio management and optimization of asset allocation. To address the problem and understand how RL can be applied to it we need to define important finance-related concepts.

One of the basic definitions in portfolio management is one of assets. Assets are defined as items that hold economic value. Such value can be cash, stocks, real state, loans, commodity holdings, etc. From now the main interest will be understanding stock options as assets and how they behave in a real-world stock exchange.

Going back to the definitions, a portfolio is formally defined as a set of assets. The portfolio is an asset itself and can be treated as such in further analysis. When building a portfolio out of a set of similar assets, such as stock holdings in a market, the portfolio can be characterized by a portfolio vector that defines the proportion of the total value invested in a given asset:

$$w = (w_1, ..., w_M) \tag{3.1}$$

where $w_i \in \mathbb{R}$ under the constrain $\sum_i^M w_i = 1$. The sum goes over a total of M assets. Each one of the total M assets (commonly known as the constituents) has a value and the total value of a portfolio is given by the cash value obtained by liquidizing all the assets in it. One important characteristic(or idea behind) the use of complex portfolios to spread the financial risk. Risk and how to measure and manage it is an important part of the portfolio optimization problem as it will be shown later.

Is not explicitly stated in 3.1 but portfolio value changes over time, as the value of the assets on it moves with the market (house prices going up, stock prices changing day by day, etc.) To make it explicit:

$$w^t = (w_1^t, ..., w_M^t) \tag{3.2}$$

Portfolio optimization is then defined as the problem of finding the values for the weights w_i^t such that portfolio value is maximized overtime under certain constraints. The weights w_i^t as commonly known as the **portfolio vector** . The following sections

will deal on how to asses portfolio performance in general and also introduce a few of specific computations used to build features from stock market data, for further use in machine learning algorithms.

3.1 Portfolio features

The main goal of any portfolio optimization algorithm is to maximize its value over time. The return of investment is often the most important metric regarding portfolio performance, and often the returns are related to the risk taken by the portfolio manager. High-risk investments are expected to yield larger returns whereas low-risk investments are expected to achieve smaller but consistent returns.

Returns are defined using the relative change in asset prices over a fixed period of time, this is known as the gross return, formally defined for a single asset:

$$R_t = \frac{p_t}{p_{t-1}} - 1 \tag{3.3}$$

Where p_t is the asset price at time t. The quantity p_t/p_{t-1} is known as the rate of return.

Using the gross return definition the simple return, the percentage of change in the price from time $t-1$ to time t can be defined as

$$r_t = \frac{p_t - p_{t-1}}{p_{t-1}} = 1 - R_t \tag{3.4}$$

Returns are more appealing for modeling than raw price behavior, in many cases researchers and investors are not so interested in knowing the exact market prices, but understanding the trends and the changes happening as time passes. Directly comparing a time series for two or more assets is not straightforward and it is hard to get out it information such as risk and return-of-investment.

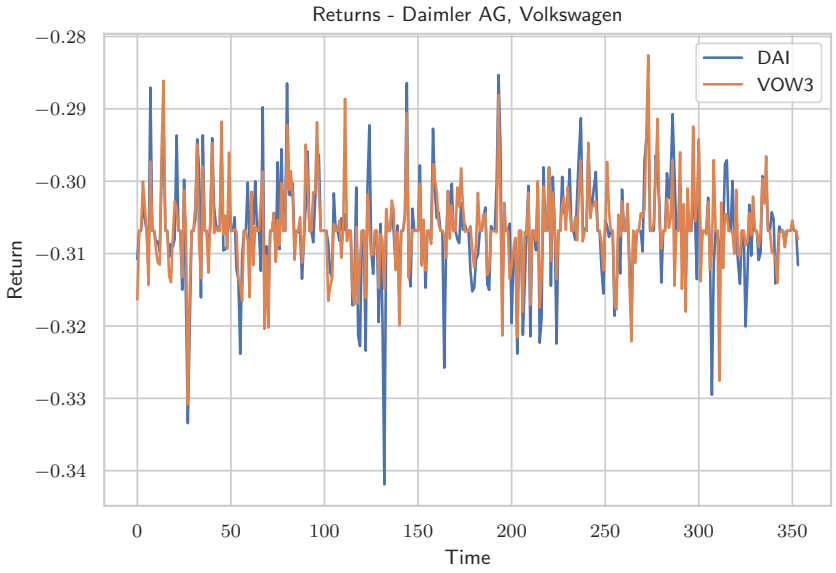

Figure 3.1: Daily returns for *Daimler AG* and *Volkswagen*

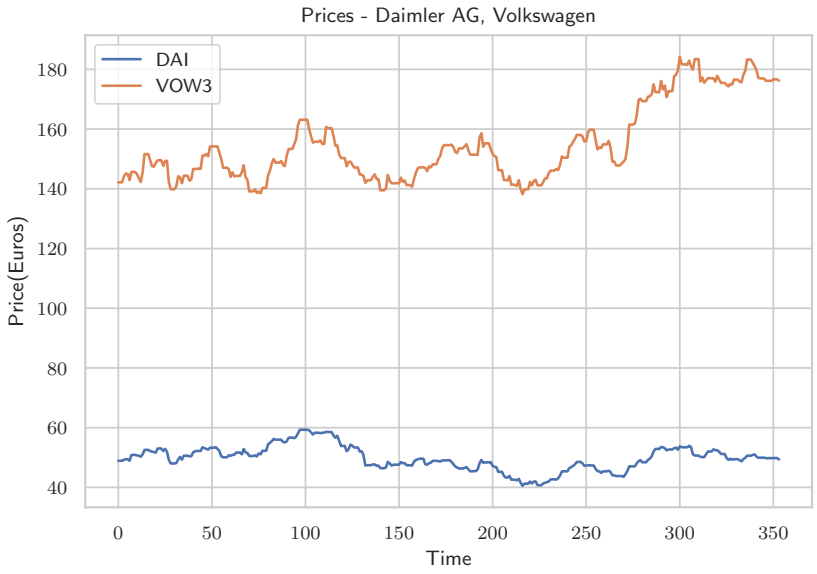

Figure 3.2: Daily prices of shares for *Daimler AG* and *Volkswagen*

One of the advantages of using returns like the ones shown in figure 3.1 is that data now is on the same scale, this is a desired feature for machine learning and in particular neural networks. The price series 3.2 can be used for another kind of qualitative analysis, for instance, one might be interested in price cycles, or trading strategies based on moving averages.

The definition of simple return given for a single asset can be extended to portfolio return, given a portfolio vector and the portfolio constituents' vector of returns at a given time t:

$$\vec{r^t} = \left[r_1^t, ..., r_M^t\right] \tag{3.5}$$

$$PR = \sum_{i=1}^{M} w_i^t r^t \tag{3.6}$$

The portfolio return (3.6) is real number, at every time step t. Although the usage of returns offers an easy way to quantify assets behavior, it has been shown that returns alone lack of some desired symmetry properties. To address this problem is common to use log returns, defined as :

$$\rho_t = ln\left(\frac{p_t}{p_{t-1}}\right) \tag{3.7}$$

And for a portfolio defined by a portfolio vector w:

$$pr_t = ln\Big(w_t \cdot r_t\Big) \tag{3.8}$$

Equation (3.7) will be important for building features for reinforcement learning algorithms (more details in the methodology chapter) as a preview we can already say that the goal the RL agents will seek is the maximization of the portfolio log return (3.8).

if we consider from the previous definitions and the fact that the portfolio changes value every time step, the portfolio final value is defined as:

$$p_f = p_0 \exp\left(\sum_{t}^{T} lnw_t \cdot r_t\right) \tag{3.9}$$

For a portfolio managed during a fixed number of time steps T. Mathematically, equation (3.9) could be simplified using the properties of exponential and logarithms, but in this particular case, the numerical computations are more stable when computed without simplification. The important term in the equation is the exponent, the maximization of $\sum_t^T \log w_t \cdot r_t$ is equivalent to the portfolio maximization task.

The final portfolio value is the most important metric, but not the only one to be taken into account. There are a few important metrics used by finance experts to asses portfolio performance, next a definition of the most important ones will be provided, such metrics will later be used to compare the performance of the algorithms.

1. Cumulative returns: Cumulative returns are cumulative sum of the daily returns. It can also be calculated as a single number, based on the final and initial value

of the portfolio:

$$CR = \frac{\text{Final value } - \text{Initial value}}{\text{Initial value}} \tag{3.10}$$

It does not take into accoun other sources of income related to the holdings such as dividends.

2. Annual returns: It is the return the porfolio provides over a period of one year. For stocks the annualized return is often calculated as follows:

$$CAGR = \left(\left(\frac{\text{Final value}}{\text{Initial value}}\right)^{\frac{1}{years}}\right) - 1 \tag{3.11}$$

CAGR stands for compound annual growth rate.

3. Sharpe ratio: Developed by Nobel laureate William F. Sharpe, was introduced to help investors to better understand the return-risk relationship. It can be computed as follows:

$$SR = \frac{\vec{w}^T \cdot \vec{\mu_T}}{\sqrt{\vec{w}^T \Sigma \vec{\mu_T}}} \tag{3.12}$$

Where $\vec{w}$ is the portfolio vector, μ_T is the average return of each one of the portfolio constituents for a time window of size T, and Σ is the covariance matrix of the returns. In general the higher the Sharpe ratio, the higher the return expected for the risk taken, A sharpe ratio of 0.5 is considered to match the market's overall performance (0.5 is roughly the sharpe ratio of the S&P 500 index) Sharpe

4. Calmar ratio: Similar to Sharpe ratio, the Calmar ratio aims to provide a risk-adjusted return. It is defined as:

$$CalR = \frac{\text{Average annual rate of return}}{\text{Maximum drawdown}} \tag{3.13}$$

Where the **Maximum drawdown** *is the maximum observed loss from a peak to a trough of a portfolio before a new peak is attained* (Investopedia 2020). As before higher ratios are an indicator of better performance, similar to the Sharpe ratio, a value of 0.5 is considered the market benchmark.

5. Sortino ratio: It is a variation of the Sharpe ratio, it aims to quantify the downside deviation, that is, the volatility of negative returns. It is computed the same as the Sharpe ratio, but instead of dividing the excess of returns by the total standard deviation of returns, only the deviation of the downside (standard deviation of negative returns) is taken into account. The common benchmark for the Sortino ratio is 1.0, 2.0 is considered good and above 3.0 exceptional.

Alongside with the previously defined metrics, graphs for rolling or windowed ratios (Sharpe, Sortino or Calmar) can be used to asses the portfolio evolution going beyond a single number (since,as seen before, portfolios evolve)

3.2 Benchmark investments

Any good algorithm of the method needs something to be compared with. In the case of investments in the stock market, there are many different indexes tracking the performance of different market sectors and companies.

The most famous one is the S&P500, which tracks the 500 largest U.S. publicly traded companies, and combine their stock prices in a single number. Alongside the S&P500, in the US market, there are other famous and useful indexes such as the Dow Jones Industrial, and the NASDAQ Composite. Specific economic sectors have their own indexes aiming to provide a general (and rather simplified) view of the overall sector performance.

These indexes track the market behavior but the regular investor can not buy shares of them, there is no "buying" S&P500, they are not per se financial products but indicators of overall market performance.

This problem was solved by the creation of **Exchange trade funds**, ETFs. ETFs are mutual investment funds (own by many individuals and companies that pool resources together) invested in a wide number of assets (usually, stock holdings) with the aim of replicating the performance observed by market indexes such as S&P500 or Dow Jones. Unlike the indexes they track, ETFs can be traded in a stock exchange just like any other stock. The main advantage is, of course, the fact that ETFs are carefully crafted by financial experts to follow the market, hence, they are less prone to volatility. ETFs are not risk-free investments, but they have grown in popularity over the past 25 years, proving to be an overall good investment in the long term.

The number of available ETFs is growing every day, they cover a wide variety of market sectors, such as the stock market (ETFs following classic indexes) to oil and biotech industries. The present work will consider three ETFs, and use them as benchmarks for the portfolio optimization methodology proposed. The ETFs are as follows:

1. SPY ETF: SPY (or as it is formally known: SPDR S&P 500 ETF Trust) is one of the most popular ETFs aiming to track the S&P500 index. SPY allocates funds in a wide variety of sectors which include technology, healthcare, financial services, communication, utilities, and real state. SPY has reported a 11.04% average annual return over a period of ten years, making it one of the most

attractive low-risk investment for investors looking to diversify risk in the US equity market.

2. EWG ETF: EWG (full name EWG iShares MSCI Germany ETF) is a concentrated holding of German equities from large to mid-size companies. It has holdings mainly in financial, technology, and consumer cyclical sectors. This ETF is included as a benchmark for the present work, given that data available for training the algorithms corresponds to a German stock exchange. EWG has yielded an average annual return of 3.24% over a period of 10 years.

3. VGK: Vanguard FTSE Europe ETF, as it is known, holds stocks for companies from developed European countries. Country-wise, the largest portfolio weights belong to Uk (25.3%) Switzerland (16.98%) and France (15.58%). Sector-wise, the most important are financial, healthcare, and consumer cyclical. VGK has reported a 3.70% average annual return over a period of 10 years.

All the previously discussed metrics and ratios apply as well for ETFs, that is one of its main advantages, classic techniques developed to asses portfolio performance can be applied to ETFs, or portfolios built out of many ETFs. The results chapter will evaluate the performance of the reinforcement learning algorithm compared with SPY, EWG, and VGK, aiming to draw conclusions about the feasibility of automated RL for day to day stock trading.

4. Experimental setup

The following sections describe an attempt to train a model-free reinforcement learning algorithm to optimize a stock portfolio. First, a description of the data sources used and the preprocessing used will be introduced, then the methodology and methods will be discussed. The chapter finishes with the results, and discussion as well as known caveats of the present approach and a few words about future research.

4.1 Data description

There are many available sources for data related to the stock market. In some cases access the most interesting data is restricted by paywalls and licenses, nevertheless, there are open alternatives to look up; for instance, the Pystock-data project (although this particular project no longer offers support or new data) has collected stock prices from the US market since 2010 up to August 2019. Paid alternatives can include Bloomberg or Morningstar, and the cost of such licenses can range in the thousands of dollars annually.

For the present research the data was collected from the `Quandl.com` API. Quandl offers a wide variety of data, paid as well as free. From quandl's free tier is possible to query three interesting markets, the **Frankfurt exchange**, Hong kong exchange, and the Euronext exchange. Sadly not all the datasets are equal in terms of quality, the number of available stocks, and timeframes.

The API allows us to query data for various stocks in a given timeframe, as always machine learning requires quality data and in the case of model-free deep RL, where a large number of agent-environment interactions is required, a large collection of data is needed. Exploring the datasets it was found that the Frankfurt data was the most complete, providing a significant number of companies available for a large enough time frame. The free data available from the other markets was simply not enough for a deep reinforcement learning approach.

The data was queried to build a dataset for daily stock prices ranging from January 2010 to December 2019. Since companies can stop trading in the market, only companies whose stock was traded throughout the entire timeframe were kept. Other

companies with large sudden unexplained drops or rises in the price were dropped from the analysis as well.

The data contains the classic open, high, low, close, and volume values for each day for each stock. It is possible to think about further methods for finding the best performant stocks and focus on optimizing those, but the main idea behind using reinforcement learning is providing little prior information and allow the systems to learn from interaction with the environment. The final dataset contains the daily price data for 68 companies.

The next step needed is to build a training environment, that is, a data representation suitable for training a neural network, a reward function to define performance for each time step and the logic to update the environment after each step. Those requirements are explained in the following section.

4.2 Methodology

Going back to the reinforcement learning problem, defining a training environment is at the core of the problem. We need to think about that information on how the system evolves as timesteps take place, what information is available, and how to change the environment's behaviour according to external agent interaction with it.

In this case, we have to constraint ourselves to a couple of restrictions and suppositions. The most important one is the suppositions that our agent interaction does not modify the market's behavior and that there is no delay between our orders and the moment when they are executed. Model aiming to be deployed into the real market can not disregard so easily those factors.

The first hypothesis is feasible for agents trading in small quantities. Retail traders have little to none impact in large markets. The second constraint can be also swept away if we consider an agent that trades only at the end of the day, and the supposition is that the trades are going to be executed first-hour in the morning, next trading day and small variations are not supposed to largely impact on portfolio performance. This supposition is reasonable in settings were the rebalance of the portfolio takes places during large enough time windows(overnight) in contrast to high-frequency trading, where the profit is made by successfully executing trades with millisecond precision,

The proposed training environment consists of serially transversing the data, reporting for each time step an observation tensor. The environment keeps track of the returns at each time step as well as and the portfolio value.

4.2.1 Data representation

For each time step t, the environment is allowed a fixed lookback window T. Based on the price information for that window of time, an observation tensor is returned.

The observation tensor is made up of three components (can be seen as the different channels in an image) making the dimension of the price tensor equals to $[M, T, 3]$ where M number of possible stocks to trade in a given market.

At a given time t The first channel of the observation is made from the ratio p_t^c/p_{t-1}^c where p^c is the closing price of the stock at time t. The second and third channel are p_t^h/p_{t-1}^h and p_t^o/p_{t-1} with p^h and p^o being the highest price and the opening price respectively. Given that the policy is going to be a neural network is ofter recommended normalize the data to increase training stability, Here the simple normalization proposed is computing the element-wise natural logarithm of the data. In this setting, the entries of the tensor are the log-returns, whose properties are well studied and have been proved of providing good features for market modeling.

At each time steps, the environment receives an action, this action is the portfolio weights w_t (vector size M) and computes the returns according to :

$$r_t = \vec{w_{t-1}} \cdot R_t \tag{4.1}$$

with R_t being the daily return at time t, assuming the environment trades at the end of the day: $R_t = \frac{p_t^c}{p_{t-1}^c} - 1$. It is important to note that the rewards at time t are given by the current return R_t and the portfolio weights in the previous time step, w_{t-1}.

This return can be used as the **reward** for each time step, and so the agents will train to maximize the cumulative log returns, thus for training to be profitable. designing a suitable reward function is one of the main challenges when applying RL method, and it is often a good idea to provide smooth reward functions to gradually guide the learning process of the agent rather than sharpe and uninformative ones. From a numerical point of view, it can also happen that the scale of the reward affects the numerical stability of the computational methods. The experiments conducted used the natural logarithm of (4.2.1) multiplied by 100. Empirical research without the 100 factors showed the agents were unable to learn from the data. It is also a direct way to optimize the portfolio final value:

$$p_f = p_0 \exp\left(\sum_t^T r_t\right) \tag{4.2}$$

Another important part of the training process is the definition of **training**

episode. In RL the agents interact with the environments until the task is completed or the environment signals to finish the process. As seen before each one of the interactions is a trajectory or episode, in our case one episode has a fixed length (in trading days) and those episodes are randomly sampled from the whole training data. More clearly, at the beginning of each episode, the environment selects a random day from 0 to the maximum number of days minus T, the agent will interact with the environment from that day for the following n_steps, after that the training episode ends and the process can start over. The agent is being trained to maximize the cumulative log return during that period of time (a short period of time if compared to the total available days). This randomly sampling process exists for two reasons; first, it allows to greatly enlarge the diversity of the data being feed to the network which helps to deal with overfitting and regularization, and second agents are trained to work for a short period of time which is aligned with typical algorithmic trading research, it is unlikely that a single algorithm can keep performing well for long periods of time without reasonable updates.

The experiments consider a fix $T = 30$, and a total of $n_steps = 90$; namely, the algorithm is allowed to look 30 days into the past to make a decision and it trades for 90 consecutive days. After the 90 training days the environment resets itself and another random training episode can be sampled from it.

In previous chapters different methods were explained, the following experiments were conducted using the Asycncronous Actor-Critic and Proximal Policy Optimization. The main reason is the possibility to train A2C agents without specialized hardware such as GPUs whilst being able to achieve the same performance that more computing costly methods.

As training data, the daily stock prices for 68 assets from January 2010 to December 2018 were used. The test period was the year 2019. The model parameters are found using the Adam optimizer. The only hyperparameters of the model are the learning rate, set to $1e-5$ and the discount factor $\gamma = 1$, usually the γ factor is set to discount the cumulative rewards as the time moves forward in the simulation, but in the case of profit or monetary rewards, a dollar made today worths the same as a dollar made a week ago (the model doesn't take into account any currency devaluation). The model also includes a risk-free asset (cash) that has returns equal to zero at any point in time.

The final element of the framework is the neural network architecture:

The input is the price tensor described before, The first part of the network is shared between the value network and the policy network (the policy network can also be referred as the agent) The shared part is two layers of LSTM networks, with 256 units each. LSTM network has been used successfully to solve time series prediction

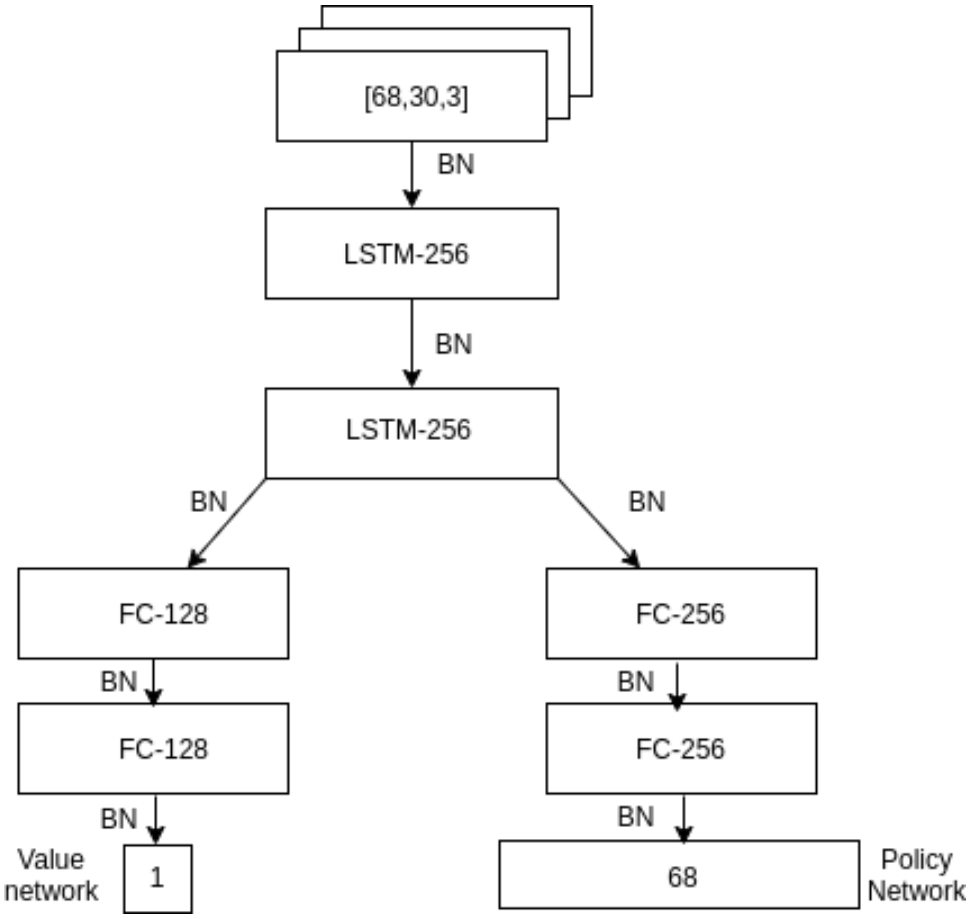

Figure 4.1: Network architecture

and forecasting problems, hence, it makes sense to take advantage of such architecture for extracting features from the price tensor(that is a concatenation of time series for different assets). Afterward, the value network and policy networks are made of fully connected layers of sizes 128 and 256, respectively. All layers are connected with ReLu activation and batch normalization layers for regularization.

the outputs from the policy network are the portfolio weights w_t. It is important to remember that such weights most satisfy $\sum_i^M w_i^i = 1$, this can be achieved with a softmax activation in the last layer, but in this case, the normalization of the weights occurs inside the environment, simply by diving the outputs by the maximum weight.

4.3 Results

The ultimate goal of the algorithm is to maximize the final portfolio value. Nonetheless, an agent that is able to optimize the portfolio returns is not necessarily a good investment option, as seen in the portfolio optimization chapter, returns are often cou-

pled with other risk and performance metrics. To asses, the results of the presented method as a possible investment strategy, a comparison between the algorithm's returns a couple of ETFs is proposed. The results would compare the output from two different reinforcement algorithms, Asynchronous Actor-Critic (A2C) and Proximal Policy Optimization (PPO).

The following figures present the learning curves observed during the training for A2C method (Similar ones were observed for PPO), for the average discounted reward and the policy loss vs timestep:

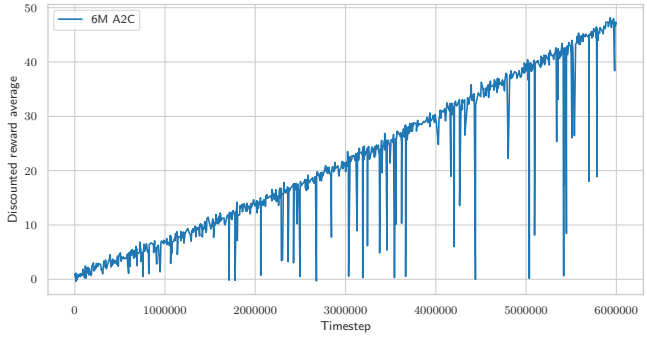

Figure 4.2: Average discounted vs timestep

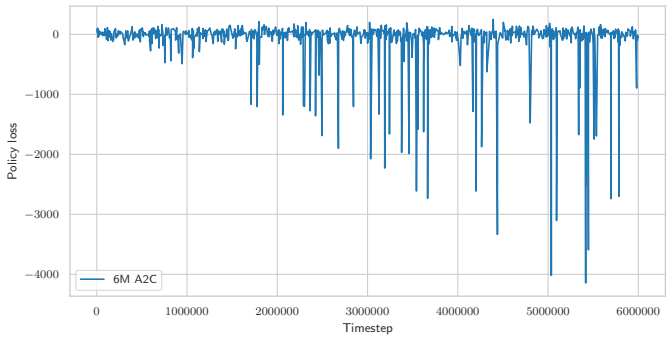

Figure 4.3: Policy loss vs timestep

The policy loss, as defined in (2.48) displays an overall noisy behavior but the trend is slowly decreasing. The average discounted rewards increase over time showing that the agent is effectively exploiting the environment. Despise the big jumps taking place in the policy loss, the algorithm was able to keep the numerical stability and continue monotonically improving.

After the training process, the algorithm was tested against unseen data, the stock prices for the year 2019. The result of the testing is a series of daily returns, the

following figures show the cumulative returns achieved by the models compared with the German fund **EWG-ETF** for the year 2019:

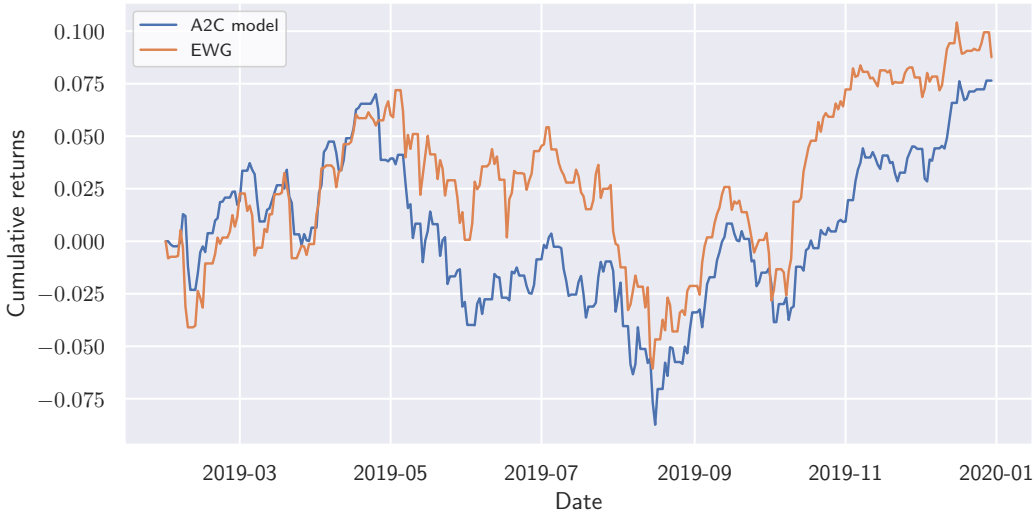

Figure 4.4: Cumulative returns, A2C vs EWG

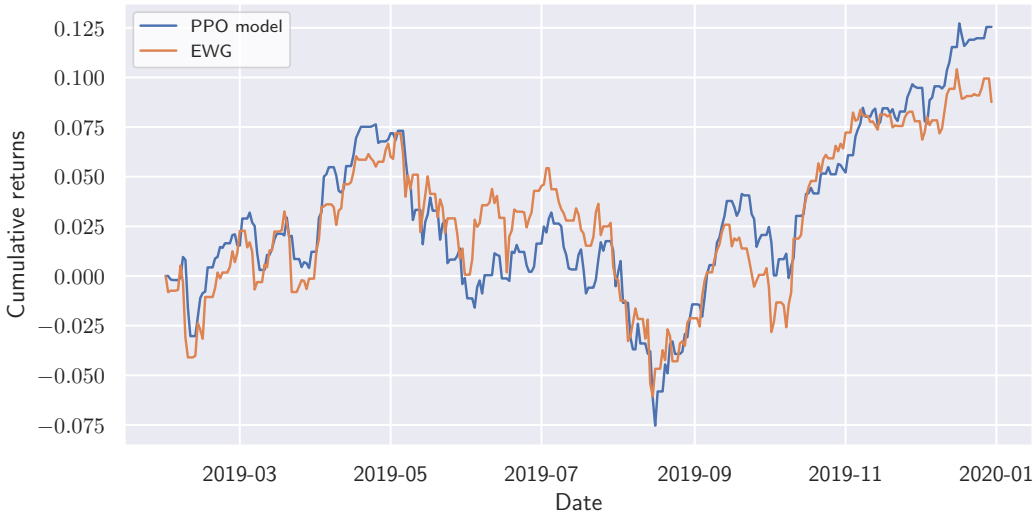

Figure 4.5: Cumulative returns, PPO vs EWG

From figure 4.3 we can see the A2C model behaves similarly to the ETF. The cumulative return for the algorithm was 7.2% and the Sharpe ratio for the portfolio is 0.55. A good rule of thumb stabilizes that Sharpe ratio above o.5 signifies above the market performance.

Figure 4.3 shows how PPO largely outperforms the benchmark, achieving a total cumulative return of 12.5% and 0.9 Sharpe ratio.

The value of the Sharpe ratio is not fixed and it depends on the portfolio behavior for a fixed window of time, the **rolling sharpe** can be used to asses the evolution of the portfolio expected return over time:

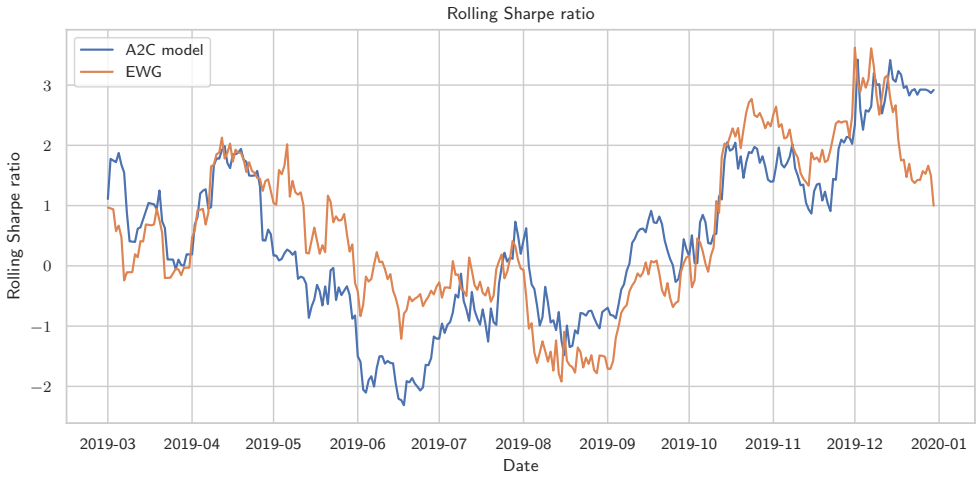

Figure 4.6: Rolling Sharpe ratio, 60 days window

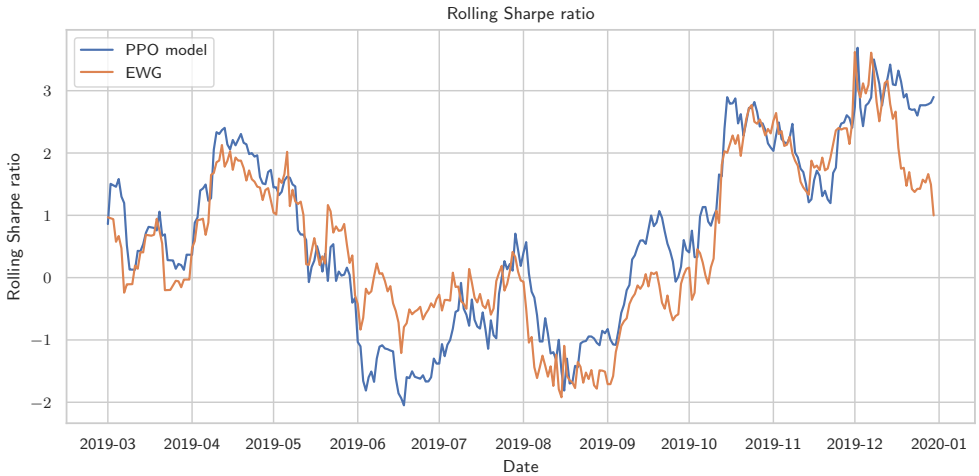

Figure 4.7: Rolling Sharpe ratio, 60 days window

It is interesting noticing how for the last month of 2019 EWG showed a consistent drop in it's rolling Sharpe ratio, whilst both algorithms were able to keep a constant performance.

The distribution of the returns by month can be visualized as follows:

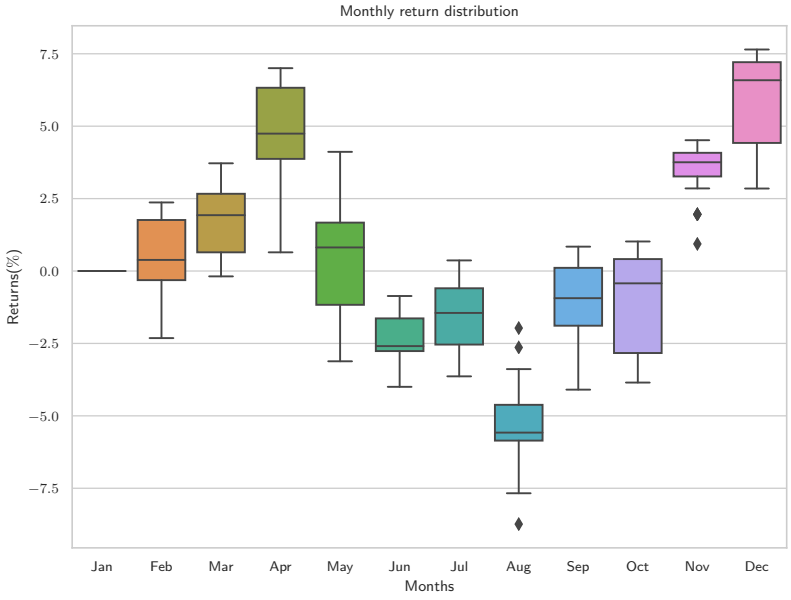

Figure 4.8: Montly returns, A2C model

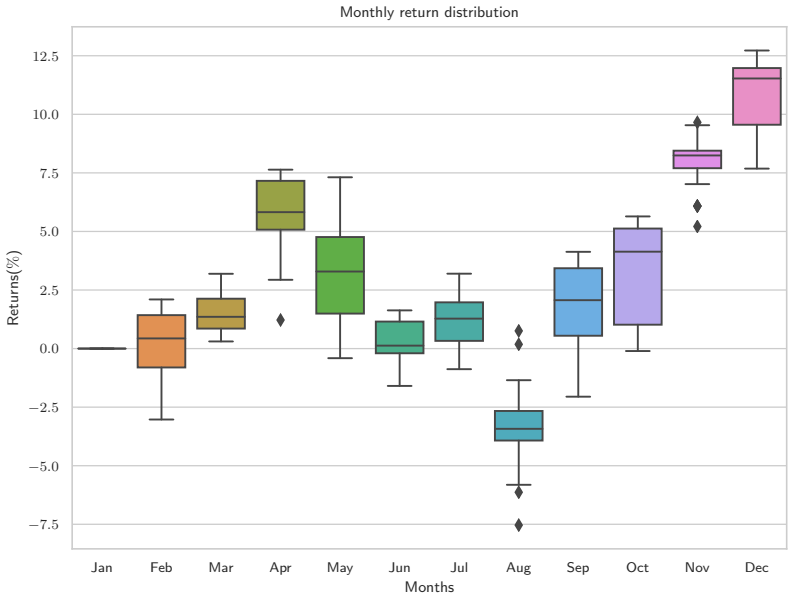

Figure 4.9: Monthly returns, PPO model

The overall trend is present in both models, although PPO seems to be doing a better job at mitigating the losses during the worse month (August). The following

table summarizes both algorithm's performance and offers a comparison with the other two commonly traded ETFs:

Performance metric	**A2C**	**PPO**	**EWG**	**SPY**	**VGK**
Annual growth rate(%)	5.40	9.30	6.10	14.7	9.00
Cumulative return(%)	7.20	12.5	8.10	20.0	12.1
Sharpe ratio	0.55	0.90	0.56	1.44	0.91
Calmar ratio	0.36	0.65	0.48	2.23	1.03
Sortino ratio	0.73	1.23	0.75	2.01	1.24
Max drawdown(%)	-14.9	-14.4	-12.8	-6.6	-8.7

Table 4.1: Overall performance comparison

The results point towards the idea that the implemented methods are successfully learning to follow profitable paths in the market, the closeness to the ETFs (that are carefully crafted by finance professionals) also signals this. Another interesting aspect of the strategy is that the algorithms seem to be risk-aware (Signaled by Sharpe ratios comparable with those exhibit by ETFs) even when the reward function does not contain explicit information about the risk. it is perhaps due to the fact that the risk is spread across 68 assets, thus reducing common risk factors.

4.4 Caveats and discussion

None of the previously reported results can be taken as financial advice. And there are important caveats to the presented approach that can modify the final overall performance. First, the testing period is rather short, one year of data could not be enough to asses the long-term value of the portfolio built by the algorithms. ETFs have a long history of performance and their holdings are not frequently rebalanced. Portfolio rebalances lead to transfer and commission fees with brokers. The presented models ignore any commission fees which is unrealistic in real life, although there are online platforms such as Alpaca or Robin Hood, who offer some extend of commission-free trades. It can be argued that commission fees will eat a small percentage of the returns (under 1%) thus not affecting the overall portfolio value trend.

Another important caveat is the training methodology. When training machine learning algorithms it is common to have a 3-fold split of the data, usually named as: training, test, and evaluation data set. The training and test dataset are used in the training loop, where for instance training is stopped once a performance metric measured over test data stops improving, then the final performance of the algorithm is

assessed using the validation data. This methodology helps to avoid the overfitting of training and test data. The presented methodology lacks this third validation dataset, here training data ranges for a period of 8 years, and the training is carried out for a fixed number of timesteps(in the order of millions) without regard to the performance on test data. As a consequence, it is impossible to argue if the results found to correspond to an optimal solution, or if more training steps are required.

Deploy such an algorithm to trade into a real-world environment has its own challenges. From the data engineering point of view, from feeding the data on time and making sure that such data is correct and update, to submit orders to brokers. Even if the current setup proposes only daily portfolio updates(contrary to high-frequency trading, with deals being done, and decisions are taken in a matter of milliseconds) at the end of the day, the algorithm is completely blind to prices changes that can occur during the off-times of the market, nor can easily adjust if sudden changes modify the overall market's behavior. On top of this point, in the present approach, there is no way to deal with companies going broke or stop trading stock in the market.

More than providing a way to beat the market, the present research aims and answers the question: are there market trends that can be picked up by a model-free reinforcement learning algorithm? The answer seems to be yes, given the portfolio performance observed for the A2C and PPO algorithms.

5. Conclusions

Two model-free reinforcement learning algorithms were successfully trained to optimize a stock portfolio made up of 68 different companies in the Frankfurt exchange. The two algorithms, A2C and PPO, rely on the same deep neural architecture based on LSTMs and both achieve comparable performance to the observed for two European ETFs used for comparison, namely the EWG German and the VGK European ETFs. PPO outperforms the EWG benchmark, whilst A2C performs slightly worse. The only metric in which the proposed models are lagging behind is the max drawdown, which is probably a consequence of the used reward function (cumulative log return) which is completely unaware of the risk, thus causing the algorithm to hold assets prone to price drops.

When compared with each other PPO outperforms A2C, by achieving a higher cumulative return (+5,3%) as well as higher return-risk ratios(+0,35 Sharpe ratio, +0,12 Calmar ratio, and +0,03 Sortino ratio). Another point in favor of the PPO algorithm is that it required half the number of iterations that A2C to converge to better results, proving to be, as often claimed, a more sample efficient method.

The results suggest that model-free RL is able to pick up profitable trading strategies from daily price data, however, the current methods are unlikely to be directly applied to algorithmic trading due to a couple of important limitations such as, the not inclusion of trading fees or sources of data other than stock prices to help the model understand outside events that can affect the market's overall behavior.

There are a few interesting points that could lead future research on the topic, aside from the ones already mentioned, setting up a more robust training-testing strategy, different network architectures as well as using richer sources of data can result in more complete strategies. Richer sources of data could mean increase the granularity of the data from days to hours or minutes, or include other financial information such as financial news, dividends, and quarterly reports.

Bibliography

[1] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. W. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning. *ArXiv*, abs/1912.06680, 2019.

[2] J. Gauci, E. Conti, Y. Liang, K. Virochsiri, Y. He, Z. Kaden, V. Narayanan, and X. Ye. Horizon: Facebook's open source applied reinforcement learning platform. *ArXiv*, abs/1811.00260, 2018.

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *ArXiv*, abs/1801.01290, 2018.

[4] J. M. Hare. Dealing with sparse rewards in reinforcement learning. *ArXiv*, abs/1910.09281, 2019.

[5] N. M. O. Heess, T. Dhruva, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments. *ArXiv*, abs/1707.02286, 2017.

[6] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[7] S. M. Kakade and J. Langford. Approximately optimal approximate reinforcement learning. In *ICML*, 2002.

[8] M. Lanctot, M. H. M. Winands, T. Pepels, and N. R. Sturtevant. Monte carlo tree search with heuristic evaluations using implicit minimax backups. *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[9] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[11] S. Nagendra, N. Podila, R. Ugarakhod, and K. George. Comparison of reinforcement learning algorithms applied to the cart-pole problem. *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 26–32, 2017.

[12] M. S. Peñas, H. JoséAntonioMartín, V. López, and G. B. Juan. Dyna-h: A heuristic planning reinforcement learning algorithm applied to role-playing game strategy decision systems. *Knowl. Based Syst.*, 32:28–36, 2011.

[13] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. 1994.

[14] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz. Trust region policy optimization. In *ICML*, 2015.

[15] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[16] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gabor Bartok, Jesse Berent, Chris Harris, Vincent Vanhoucke, Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. [Online; accessed 25-June-2019].

[17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[18] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 16:285–286, 1988.

[19] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, 2015.