NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF MATHEMATICS

AUTOMATED TRADING SYSTEM WITH REINFORCEMENT LEARNING

JOSÉ NEVES

BSc in Physics

MASTER IN MATHEMATICS AND APPLICATIONS

NOVA University Lisbon
March, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
MATHEMATICS

AUTOMATED TRADING SYSTEM WITH REINFORCEMENT LEARNING

JOSÉ NEVES

BSc in Physics

Examination Committee

Chair: Doctor Marta Cristina Vieira Faias Mateus
Associate Professor, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Rapporteur: Doctor Cláudia Alexandra Magalhães Soares
Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

Adviser: Doctor Rui Alberto Pimenta Rodrigues
Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa

MASTER IN MATHEMATICS AND APPLICATIONS

NOVA University Lisbon
March, 2023

Automated Trading System with Reinforcement Learning

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.18) [14].

Abstract

This thesis focuses on using deep reinforcement learning to obtain agents capable of making valid portfolio management decisions that result in profit in increasingly more complex and realistic market simulator trading environments. The goal is to use an incremental evolution of the environment features and price patterns to have more control over the training of an agent capable of performing in more challenging and realistic episodes. The model used for this task is the Proximal Policy Optimization(PPO) algorithm. The thesis intends to present an environment-building-focused training process as a possible tool to answer the inherent challenges of the deep reinforcement learning model's training and a new perspective on the creation of automatic portfolio management systems with reinforcement learning.

Keywords: Machine learning, Deep Learning, Automatic trading systems, Deep Reinforcement Learning

Resumo

Esta tese foca-se na utilização de aprendizagem profunda com reforço para obter agentes capazes de tomar decisões válidas de gestão de portfólio que resultem em lucro em ambientes que simulam o mercado cada vez mais complexos e realistas. O objetivo é utilizar uma evolução incremental das características dos ambientes e dos padrões de preços para ter um maior controlo sobre o treino de um agente capaz de executar episódios mais realistas. O modelo utilizado para esta tarefa é o algoritmo Proximal Policy Optimization. A tese pretende apresentar um processo de treino centrado na construção de ambientes como uma possível ferramenta para responder aos desafios inerentes ao treino de modelos de aprendizagem profunda com reforço e uma nova perspetiva para a criação de sistemas automáticos de gestão de portfólio com aprendizagem por reforço.

Palavras-chave: Aprendizagem de máquina, Aprendizagem profunda, Sistemas de trading automáticos, Aprendizagem de reforço profunda

Contents

List of Figures

List of Tables

1

Introduction

If intelligence is a cake, the bulk of the cake is unsupervised learning, the icing on the cake is supervised learning, and the cherry on top of the cake is reinforcement learning.

Yann LeCun (2016)

Portfolio management consists in making a sequence of trading decisions to maximize the portfolio's value relative to some objective function representative of the client's satisfaction. Machine learning approaches to the problem have been around since the early 2000s. The 2010s were marked by the advent of deep learning. Today, companies like Goldman Sachs and JPMorgan actively include and study deep reinforcement learning approaches to asset management. In this thesis, we first focused on training an agent capable of solving the simplest version of the problem we could implement and then incrementally modifying that version until we arrive at an environment capable of approximating the task of a trading bot in the stock market. We trained an agent for each of those versions. The thesis is structured as follows:

- **Chapter 2 (State of the Art):**
  This chapter starts with a formal mathematical definition of the portfolio management problem. Then, a brief review of the literature on machine learning and non-machine learning solutions for the portfolio management problem is presented, to which a state-of-the-art revision of reinforcement learning and deep learning concepts follows.

- **Chapter 3 (Methods and Results):**
  This chapter details the characteristics and presents the results associated with each environment implemented. It also presents the different price patterns with which the agents were challenged.

- **Chapter 4 (Conclusion):**
  This chapter concludes the thesis, summarizes the work it encompasses, the main challenges faced, and possibly interesting future research scopes on the topic.

2

State of the Art

2.1 The Portfolio Management Problem

Portfolio management consists in making a sequence of trading decisions to maximize the portfolio's value relative to some objective function representative of the client's satisfaction. This objective function often includes terms representing the expected return and a measure of risk. The goal is to find a balance between maximizing returns and minimizing risks, as the optimal portfolio must reflect the trader's investment goals and risk tolerance. There are multiple methods through which we can mathematically define this problem. Consider:

- a market with n risky assets,
- an investor with wealth x_0 at $t = 0$ that aims to distribute wealth among the risky assets in order to maximize expected returns and minimize the risk of the portfolio in $t \in [0, T]$,
- an investment policy $u_t = (u_t^1, ..., u_t^n)$ where u_t^1 is the amount invested in the i-th asset at the time t,
- ϕ is a weighting parameter representative of how risk averse the investor is.

The goal is then to find a policy u_t so that:

$$\max_{\{\boldsymbol{u}_t\}_{t=0}^{T-1}} \mathbb{E}[x_t] - \phi \operatorname{Var}(x_t) \tag{2.1}$$

2.2 Non-Machine Learning Approaches

This section will discuss some of the most relevant traditional approaches to the portfolio management problem. Left out are techniques like the Black-Litterman model([3]) and Robust Optimization([7]), which are also relevant in today's applications and solution development.

The mean-variance model was the first relevant proposed solution for the portfolio management problem([16]). In it, the trading agent tries to maximize its expected returns while establishing a limit for what it considers a reasonable risk value or to minimize risk for a considered good enough expected return value. The expected returns are approximated through the weighted average of the past returns of the assets and the risk through their variance. The problem was first formalized in a single-period framework and then generalized to a multi-period framework, with [11] as an example and the first work to be able to arrive at an analytical solution to the discrete-time multi-period mean-variance problem. In [35], the authors were able to also arrive at an analytical solution for the continuous mean-variance framework of the problem. These and other approaches([2], [22]) also sought to arrive at analytical solutions focused on solving the inherent time-inconsistency problem of the mean-variance framework, which means the optimal strategy found in t is no longer optimal in $t + 1$([31]).

Risk management approaches prioritize minimizing risk when building a portfolio to protect the trading agent against the intrinsic volatility of returns and unexpected events. A relevant example of this family is the Risk-parity approach([15]). It consists in using a metric to quantify each asset's associated risk, and then distributing wealth to achieve an equal risk distribution among the assets.

The Capital Asset Pricing Model(CAPM) aims to establish assets' systematic risk and their expected return([27]) through a linear relationship. This relationship describes how the investor can be rewarded by accepting more risk.

Factor investment methods, on the other hand, admit that many possible factors can be correlated with high returns, like market risk, value, and company size. The Fama-French 3-factor model([5]) was developed in the 1990s. It uses these three factors, the risk of investing in the stock market as a whole, the price-to-book of the company (to detect undervalued companies), and the size of the firms behind the stocks to build portfolios. This model helped to explain patterns in stock prices correlated to firm characteristics that the capital asset price model treated as random anomalies([6]).

2.3 Reinforcement Learning Approaches

Applications of reinforcement learning for portfolio optimization is a relatively new field of research, with the first papers appearing in the early 2000s. In [19] and [20], J. Moody was one of the first authors to demonstrate the advantages of reinforcement learning approaches over supervised learning methods in the portfolio management problem. The

first used approaches of this kind were mainly value-based methods, in which a model like Q-learning([32]), Temporal Difference Learning(TD)([28]) or SARSA([24]) were used to map states to predicted returns, enabling the agent to choose the trading actions that lead to the path of greater expected returns([4], [21], [23]).

In [23], for example, the author built three discrete state agents to manage a two-asset personal retirement portfolio: two with discrete action space (SARSA and Q-learning) and one with continuous action space, that either focused on maximizing profit or the sharpe ratio. At the time, there was also great interest in recurrent reinforcement learning algorithms(RRL)([9]), a policy search method. They were purposefully designed for the portfolio management problem. Instead of calculating the value function as an intermediary between observations and action, RRL techniques use a recurrent neural network to predict actions directly based on observations, avoiding the dimensionality curse from which techniques based on the Bellman equation suffer. In some works, these techniques were shown to surpass value-based methods([4], [19]).

In [12], the authors compare the performance of Proximal Policy Optimization(PPO)([25]), Deep Deterministic Policy Gradient(DDPG)([13]) and Policy Gradient(PG)([30]) methods in achieving average daily returns and sharpe ratio while managing five stocks. PG is considered to be the "vanilla"policy search approach, while DDPG and PPO are built by adding features on top of it. However, PG techniques were the ones to achieve higher returns and Sharpe ratio in this paper, while the PPO and DDPG could not even perform on the training set.

[34] uses a Generative Adversarial Network(GAN)([8]) model to generate data that tried to simulate price dynamics, on which a model-based DDPG model was trained. RL approaches are divided between model-based and model-free. The authors of this paper suggested that one main disadvantage of previous RL approaches is their sample inefficiency, given their complete lack of intuition at the beginning of the training. Model-based approaches inform the agent of state transition dynamics. The paper does that through an Infused Prediction Model(IPM) to predict stock trends and a Behaviour Cloning Model(BCM) to directly inform the agent how to not diverge in previous unseen situations. The results showed a better performance than previously used model-free approaches.

[1] proposed merging the mean-variance model with the actor-critic framework present in all current state-of-the-art reinforcement learning techniques. The actor (which is the policy model) used the mean-variance model when choosing the action to take, while the critic (responsible for assigning the actor's loss function through value-based techniques) used the Kelly Criterion to associate a value to each state. All the major RL models for continuous action space state of the algorithms(SAC, DDPG, TD3, AC3, PPO, etc...) were adapted in this manner. An implementation of the Deep vanilla policy gradient(DPG) model was shown again to achieve the best performance.

2.4 Reinforcement Learning theory

2.4.1 Fundamental Concepts

- **Policy,** π: The model that maps states to actions.
- **Agent**: The entity that applies the policy and interacts with the environment.
- **Environment**: The entity upon which the agent acts and from which receives feedback of its actions.
- **State of the environment,** s_t: The collection of features that characterize the current situation of the environment and are important for the agent to consider when making a decision at the timestep t.
- **Observation,** o_t: The data the agent has access to in each timestep t to perceive the state of the environment and decide which action to take.
- **Action,** a_t: The decision of the agent at a timestep t on how to act upon the environment given this step's observation.
- **Action space,** A: the collection of actions a_t the agent has available at each timestep on how to act upon the environment given the observation.
- **State space,** S: the collection of states possible for the environment to be at any given time.
- **Reward,** r_t: the scalar value the agent receives as feedback relative to the contribution of the action chosen at the timestep t towards the completion of the purposed task.
- **Episode**: A set of experiences associated with a unique run of the environment, which usually take the form of a set $(a_t, s_t, r_t, Done)$, where $Done$ signals if that was the final step of the episode or not.
- **Training**: The process by which we update the policy in order for it to achieve higher rewards in future similar experiences.

The learning process in reinforcement learning is based upon the reward hypothesis, which is defined in [29] as:

Definition 2.4.1 ***Reward hypothesis****: All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

Which means that we assume that we can reduce learning a task to learning how to maximize a sum $r_0 + r_1 + ... + r_T$ specific to the problem (where T is the final time step of the episode). Additionally, to control how much the agent values distant future rewards compared to near future rewards, a discount factor $\gamma < 1$ is also used. The goal of the agent is then to find a policy able to maximize the discounted sum of rewards G_t, expressed in the following form:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1} \tag{2.2}$$

Where if $\gamma = 1$:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = r_{t+1} + r_{t+2} + r_{t+3} + \cdots \tag{2.3}$$

2.4.2 Value Based Methods

Value-based methods associate an estimate for the discounted sum of rewards to each state. The estimates work as a map that enables the agent to choose which path to take. To the function responsible for these estimates, we call value function($v_\pi(s)$). We define it as:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T-1} \gamma^k r_{t+k+1} \mid s_t = s \right] \tag{2.4}$$

And it represents the reward we expect to obtain until the end step T of the episode given the state we are in and our current policy π. To the value estimate of a state given the action chosen by the agent at that state and the policy π, we call Q-value, and it takes the form:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{T-1} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right] \tag{2.5}$$

Value-based methods don't update state-action mapping directly. Instead, with each update, they look to improve the precision with each the discounted returns sum is estimated.

The Monte Carlo technique of estimation of the value of a state consists in running a certain amount of episodes and then average the rewards the agent obtained after visiting that state. With each new batch of examples, we do a weighted update of the previous value of the state into a more precise estimation of the expected returns.

Another technique uses instead dynamic programming. The Bellman equation describes the relationship between the value of the current state and the value of the next state and is expressed as:

$$v_\pi(s_t) = max_{a_t} r(s_t, a_t) + v_\pi(s_{t+1}) \tag{2.6}$$

Where max_{a_t} comes from our assumption that the agent will choose the action associated with the higher value, $r(s_t, a_t)$ is the reward we obtain after doing a_t and s_{t+1} is the state we end up in given that we did a_t.

Temporal Difference is considered to be one of the main concepts of reinforcement learning. *TD(0)*, the most basic version of this technique, consists of performing an action on an environment, observing the reward, and using that reward to update the value of the previous state. Formally that means:

$$v_{\pi(n+1)}(s_t) \leftarrow (1-\alpha) v_{\pi(n)}(s_t) + \alpha(r_{t+1} + \gamma v_{\pi(n)}(s_{t+1})) \tag{2.7}$$

Where α is our learning rate and represents how much we trust the new values and $v_{\pi(n)}(s_t)$ is the value function of the state s_t after being updated n times.

By inserting the actions in the expression to obtain Q-values we arrive at one of the top breakthrough methods of reinforcement learning theory: Q-learning.

$$q_{\pi(n+1)}(s_t, a_t) \leftarrow (1-\alpha) q_{\pi(n)}(s_t, a_t) + \alpha(r_{t+1} + \gamma max_{a_t} q_{\pi(n)}(s_{t+1}, a_{t+1})) \tag{2.8}$$

In 2015, Deepmind introduced the Deep Q-Network method([18]), by fusing deep learning theory with Q-learning. In it, a neural network is used to map Q-values to state-action pairs. That means, they update the neural networks weights θ to minimize the loss function:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - q(s, a; \theta_i))^2 \right] \text{ where } y_i = r + \gamma \max_{a'} q(s', a'; \theta_{i-1}) \tag{2.9}$$

2.4.3 Policy Gradient Methods

Let us now come back to the concept of policy and give it a formal definition:

Definition 2.4.2 *Policy π is the probability of performing an action a given the state s, which means:*

$$\pi(a, s) = P(a_t = a | s_t = s) \tag{2.10}$$

The goal of the agent is to arrive at a policy that maximizes the total discounted rewards. $J(\pi)$ is the discounted reward sum we expect to obtain with the policy π. Formally:

Definition 2.4.3

$$J(\pi) = E_\pi[G_t] \tag{2.11}$$

Where:

$$G_t = \sum_{k=0}^{T-1} \gamma^k r_{t+k+1} \tag{2.12}$$

In value-based methods, we use value functions and Q-values as a middle step between observing the state and choosing what action to perform. In those methods, the policy, when the model is not purposefully exploring, consists in choosing the action which will result in the state associated with higher rewards.

As stated, the policy is a set of distributions of probability. In the beginning of the training, all the actions for every state will have the same likelihood of being chosen. With each update, we sum or take an update value from each action probability in each distribution for the agent to achieve higher rewards in the following episodes. The update is proportional to how much the action helped to achieve or prevented higher rewards.

We perform a process called gradient ascent, in which the gradient operation is used to check in which direction the change in θ corresponds to the maximum positive change in $J(\pi_\theta)$ given the collected experiences. The update rule can then be expressed as (considering an α learning rate):

$$\theta_{n+1} = \theta_n + \alpha \nabla_{\theta_n} J(\pi_{\theta_n}) \tag{2.13}$$

Now we will focus on how this gradient is calculated. We will start by inserting the transition probabilities, that is, the probability we have of having a certain sequence of states τ(also called trajectory of states) given our policy π_θ. Given this concept, we can expand the gradient update $\nabla_{\theta_n} J(\pi_{\theta_n})$ in the following way:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\pi_\theta}[G_t] = \nabla_\theta \sum_\tau P(\tau|\theta) r(s_t) \tag{2.14}$$

Where:

$$P(\tau|\theta) = P(s_0) \Pi_{t=0}^{T-1} P(s_{t+1}|a_t, s_t) \pi(a_t|s_t) \tag{2.15}$$

Now we are going to perform a series of mathematical operations that will enable us to arrive at the central equation of policy gradient theory.

$$\begin{aligned} \nabla_\theta \sum_\tau P(\tau|\theta) r(\tau) = \sum_\tau \nabla_\theta P(\tau|\theta) r(\tau) = \sum_\tau \nabla_\theta P(\tau|\theta) r(\tau) \frac{P(\tau|\theta)}{P(\tau|\theta)} = \\ = \sum_\tau \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} r(\tau) P(\tau|\theta) = \sum_\tau \nabla_\theta log(P(\tau|\theta)) r(\tau) P(\tau|\theta) = \\ = E_{\pi_\theta}[\sum_\tau \nabla_\theta log(P(\tau|\theta)) r(\tau)] \end{aligned} \tag{2.16}$$

We will now focus on expanding $\nabla_\theta log(P(\tau|\theta))$. Remembering the definition (2.15) and applying the logarithm and gradient to it:

$$\nabla_\theta log(P(\tau|\theta)) = \nabla_\theta log(P(s_0)) + \sum_{t=0}^{T-1} (\nabla_\theta log(P(s_{t+1}|a_t, s_t)) + \nabla_\theta log(\pi_\theta(a_t|s_t))) \tag{2.17}$$

Since $log(P(s_0))$ and $log(P(s_{t+1}|a_t, s_t))$ do not depend on θ, their gradients in relation to it are zero. Which means:

$$\nabla_\theta log(P(\tau|\theta)) = \sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t)) \tag{2.18}$$

This leaves us with:

$$\nabla_\theta J(\pi_\theta) = E_{\pi_\theta}[R(\tau) \sum_{t=0}^{T-1} \nabla_\theta log(\pi(a_t|s_t))] = E_{\pi_\theta}[\sum_{t=0}^{T-1} \nabla_\theta log(\pi(a_t|s_t))G_t] \tag{2.19}$$

The Policy Gradient theory is the basis of all policy search methods and a generalization of this last expression. Since:

$$q_{\pi_\theta}(s_t, a_t) = E_{\pi_\theta}[G_t] \tag{2.20}$$

We can state:

Theorem 1 ***Policy Gradient Theorem****: The weight update of the policy π distribution towards maximization of the objective function $J(\pi)$(that is, maximization of the expected reward) is given by the expectation of the gradient of the product of the log probabilities of the actions performed given the state $log(\pi(a_t|s_t))$ and the perceived value of those actions through the reward obtained approximated through a Q-value $q_{\pi_\theta}(s_t, a_t)$ for the state s_t given a_t and the policy π_θ. That means:*

$$\nabla_\theta J(\pi_\theta) = E_{\pi_\theta}[\sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t))q_{\pi_\theta}(s_t, a_t)] \tag{2.21}$$

Which means that we will increase or decrease the probabilities of each action for each state according to the value rewards associated with those actions in each state.

We now focus on the REINFORCE([33]) method. It estimates $q_{\pi_\theta}(s_t, a_t)$ by performing an episode, calculating its discounted rewards to associate a reward to each action and updating the policy according to the expression:

$$\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t))G_t \tag{2.22}$$

Through the approximation:

$$q_{\pi_\theta}(s_t, a_t) \approx G_{t,\pi_\theta} \tag{2.23}$$

Where G_{t,π_θ} is the discounted rewards sum the policy π_θ would obtain after running the episode from the timestep t on and $q_{\pi_\theta}(s_t, a_t)$ the expected reward for the state s_t given a_t. The algorithm for this method is then:

Algorithm 1 REINFORCE

Input N number of iterations, α learning rate, frequency of the weight's update n_θ
Initialize θ weights and the environment:
Get first state $s_0 \in S$ from the environment
for episode=0,...,N-1 **do**:
Sample a trajectory τ using the policy π_θ
Calculate the discounted returns $G_{\pi_\theta,t} = \sum_{s=t+1}^{T} \gamma^{s-t-1} r_s$
Update θ through:

$$\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta log(\pi(a_t|s_t)) G_{t,\pi_\theta} \tag{2.24}$$

end for

A typically used technique to diminish the instability of the training is subtracting a baseline value b, dependent on the state, from the discounted returns G_t. To the result of the subtraction of the episodic rewards G_t by the value of each state given the current policy π $v_\pi(s_t)$, we call the advantage function.

$$\mathbb{A}_{\pi_\theta}(s_t, a_t) = q_{\pi_\theta}(s_t, a_t) - v_{\pi_\theta}(s_t) \tag{2.25}$$

And we approximate it through:

$$q_{\pi_\theta}(s_t, a_t) - v_{\pi_\theta}(s_t) \approx G_{t,\pi_\theta} - v_{\pi_\theta}(s_t) \tag{2.26}$$

The advantage function computes the difference between the reward the agent gets by choosing an action in a given state versus the average reward you get across all the possible actions in that state.

This is the basis of the Actor-Critic([17]) methods. The Actor-Critic framework consists in having a policy model to which we call actor and a critic model responsible for mapping states to values. The critic "critics"the actor by computing the reward it should have achieved given the states where it passed. The actor's policy update is then given by the expression:

$$\theta_{actor} = \theta_{actor} + \alpha \sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t)) \mathbb{A}_{\pi_\theta}(s_t, a_t) \tag{2.27}$$

That we approximate through:

$$\theta_{actor} = \theta_{actor} + \alpha_{actor} \sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t)) [G_{t,\pi_\theta} - v_{\pi_\theta}(s_t)] \tag{2.28}$$

In parallel, the critic also has to be trained to be able to give an accurate estimation of the values of the states through the temporal difference error technique as in (2.9):

$$L_i(\theta_{critic,i}) = \mathbb{E}_{s,a,r,s'} \left[(y_i - v(s; \theta_{critic,i}))^2 \right] \text{ where } y_i = r + \gamma \max_{a'} v(s'; \theta_{critic,i-1}) \tag{2.29}$$

Where s' represents the state following the state s and a' the action chosen by the agent for the state s'. The algorithm for this method then:

Algorithm 2 Episodic Actor-Critic

Input N number of iterations, α learning rate($\alpha = \alpha_{actor} = \alpha_{critic}$)
Initialize θ_{actor} and θ_{critic} weights and the environment:
Get first state $s_0 \in S$ from the environment
for episode=0,...,N-1 **do**:
Sample a trajectory τ using the policy π_θ
Calculate the discounted returns $G_t = \sum_{s=t+1}^{T} \gamma^{s-t-1} r_s$
Calculate the advantage function $\mathbb{A}_{\pi_\theta}(s_t, a_t) = G_{t,\pi_\theta} - v_{\pi_\theta}(s_t)$
Update θ_{actor} through:

$$\theta_{actor} = \theta_{actor} + \alpha_{actor} \sum_{t=0}^{T-1} \nabla_\theta log(\pi_\theta(a_t|s_t))[G_{t,\pi_\theta} - v_{\pi_\theta}(s_t)] \tag{2.30}$$

Update θ_{critic} through minimizing:

$$L_i(\theta_{critic,i}) = \mathbb{E}_{s,a,r,s'}\left[(y_i - v(s;\theta_{critic,i}))^2\right] \tag{2.31}$$

end for

Trust Region Policy Optimization ([26]) tried to improve this technique by using the KL-divergence measure to limit the distance possible between the updated policy(θ) and old policy(θ') distributions so that the update becomes:

$$\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} \mathbb{A}_{\pi_\theta}(s_t, a_t) \tag{2.32}$$

Subject to:

$$KL[\pi_\theta(a_t|s_t), \pi_{\theta'}(a_t|s_t)] \leq \sigma \tag{2.33}$$

Where σ is chosen according to the problem. The constraint makes sure that the new policy doesn't move too far away from the old policy, forcing the training to be less unstable. The problem is that this constraint adds complexity and slows down the training process, since the constraint is exterior to the objective function, disabling the model from properly integrating it into its learning.

Proximal Policy Optimization(PPO)([25]) tried to solve this problem. We start with defining the ratio between policies:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta'}(a_t|s_t)} \tag{2.34}$$

And now we present the main objective function:

$$J_t^{CLIP}(\theta) = E_t[min(r_t(\theta)\mathbb{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\mathbb{A}_t)] \tag{2.35}$$

And in it, we have the minimum of two terms: $r_t(\theta)\mathbb{A}_t$, that is the normal Trust Region Policy Optimization objective function, and $clip(r_t(\theta), 1-\epsilon, \epsilon)\mathbb{A}_t$, its clipped version.

Let's first look at the clipped version, and remember that the advantage function can have positive and negative values. That is why the clipping includes a superior limit and an inferior limit for the values that $r_t(\theta)$ can have because the distance between distributions can be too big either if we increase the log probabilities too much or diminish them too much. If the distance between the old and new policy does not surpass ϵ, then the minimum operator will select the normal $\mathbb{A}$ objective function term. But there is more. The full objective function is:

$$J_t^{PPO}(\theta) = E_t[J^{CLIP}(\theta) - c_1 J_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{2.36}$$

The first additional term is responsible for updating the critic, exactly like it was in Trust Region Policy Optimization and Actor-Critic. The second term is responsible for incentivizing exploration during training. Inserting the constraint into the objective function so that it is differentiable and included in the gradient greatly improved training. But it still creates a certain vulnerability to converge into local maximums, given the limitation of the update. This term tries to ensure that enough exploration happens during training so that the model is less likely to converge into local maximums.

Finally, the algorithm for the method:

Algorithm 3 Episodic PPO-Clip

Input N number of iterations, α learning rate($\alpha = \alpha_{actor} = \alpha_{critic}$)
Initialize θ_{actor} and θ_{critic} weights and the environment:
Get first state $s_0 \in S$ from the environment
for episode=0,...,N-1 **do**:
Sample a trajectory τ using the policy π_θ
Calculate the discounted returns $G_{t,\theta} = \sum_{s=t+1}^{T} \gamma^{s-t-1} r_s$
Calculate the advantage function $\mathbb{A}_{\pi_\theta}(s_t, a_t) = G_{t,\theta} - v_{\pi_\theta}(s_t)$
Update θ_{actor} through:

$$\theta_{actor} = \theta_{actor} + \alpha_{actor} \sum_{t=0}^{T-1} \nabla_\theta min(r_t(\theta)\mathbb{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\mathbb{A}_t) + S[\pi_\theta](s_t) \tag{2.37}$$

Update θ_{critic} through minimizing:

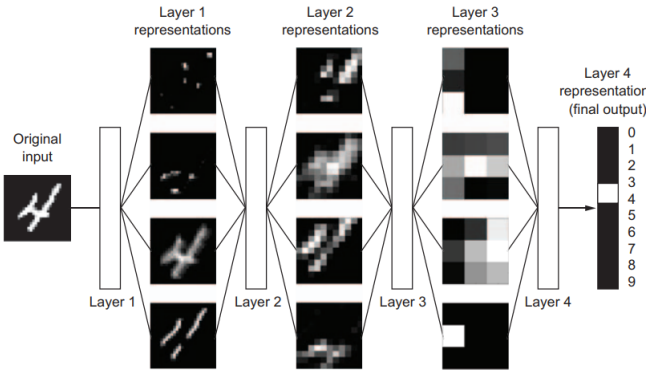
$$L_i(\theta_{critic,i}) = \mathbb{E}_{s,a,r,s'}\left[(y_i - v(s; \theta_{critic,i}))^2\right] \tag{2.38}$$

end for

2.5 Deep Learning Theory

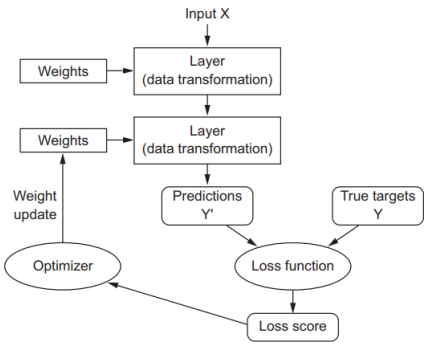
Machine learning is a field dedicated to designing models that can extract patterns from a dataset and recognize them in datasets not yet seen by the model. It does it through data transformations that will result in meaningful representations of data to the output we are approximating. Deep Learning is a subfield of ML that transforms data into increasingly meaningful representations through the use of successive layers of data transformation. Each layer is composed of neurons. Each neuron performs a transformation on the data. Depending on how useful that representation is and on the specific architecture of the neural network, it will be shared with the neurons in the next layer that will perform their on transformations on top of it. These are called feature extraction layers. The last layer is responsible for taking the features considered important for the target output and using them to predict it.

Figure 2.1 An example of the learning of deep representations for Optical Character Recognition(OCR)



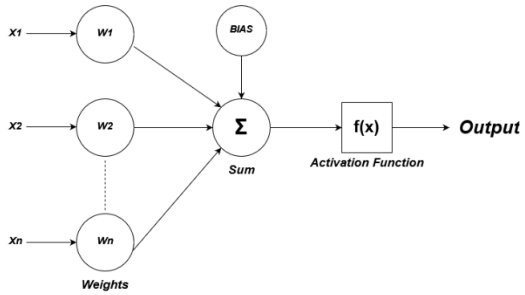
The difference between the predicted value and the real value will be used to inform the neurons on how to change their data transformations to achieve more meaningful representations that will give us more accurate results. This update process is represented in the scheme below.

Figure 2.2 Update step algorithm in neural networks



More specifically, the transformation that each neuron does to the data is defined by its weights θ. The learning process can be summarized as finding the right weight values for the task at hand. In the neuron, the weights are multiplied by the input and summed with added bias values. An activation function is applied to the result. Without an activation function, the neurons could just perform linear operations, being that we are only performing a product and a sum to it. Activation functions introduce the possibility of applying non-linear transformation on the data, enabling them to approximate a much wider range of functions.

Figure 2.3 The neural network's neuron's scheme



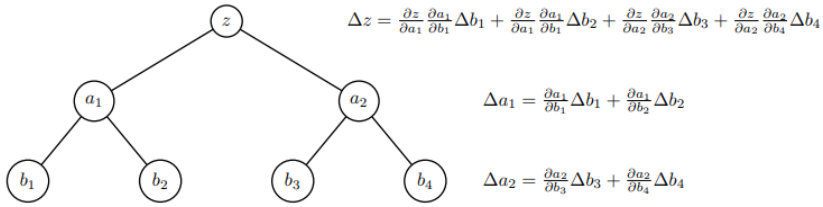
The output of the neuron will be then shared it to a neuron of the next layer that will apply its transformation according to its weights. We call this the forward pass.

Algorithm 4 Forward Pass

Input: $X^{(0)}$(input with zero transformations), N layers, activation function f, loss function L, W and b weights and biases per layer matrixes
for n = 1,...,N **do**:
$Z^{(n)} = b^{(n)} + W^{(n)} \cdot X^{(n-1)}$
$X^{(n)} = f(Z^{(n)})$
end for
$\hat{y} = X^N$
$L = L(\hat{y}, y)$

At the end of the process, predictions will be made and compared with the correct values associated with the input the neural network received. A loss function $L(\hat{y}, y)$ will be used to measure the distance between the predictions and the target values. Then it's time for the backward step, where, through the use of the chain rule($[f(g(x))]' = f'(g(x))g'(x)$), we understand how much a change in each weight $\Delta\theta_i$ means for the variation of the loss function $\Delta L(\hat{y}, y)$. This is called the backward pass.

Figure 2.4 Using the chain rule to relate the change in weights throughout the neural network



Finally, we use an optimizer to obtain the direction upon which we have to change the weights to minimize the loss score (hopefully converging into a global minimum) and make a step in that direction with a size proportional to our chosen learning rate. There are many kinds of optimizers (Adam, RMSprop, SGD). The right one to choose varies on the problem, but most of the ones used today are based on the concept of mini-batch gradient descent. In this approach, instead of using the entire dataset or one experience at a time, we compute the gradient for small batches of data (the ideal size of which varies with the situation), enabling more stable and efficient learning.

The quality of the learning process depends on our choice of a set of values, to which we call hyperparameters. They are adjustable settings or parameters that cannot be learned from the data, but must be set manually before training a machine learning model, such as learning rate, number of neurons per layer, batch size and the type of optimizer and its characteristics. The process of finding the best hyperparameters for a given training process is called tuning.

Relative to this process, Yann LeCun wrote([10]):

> *Backpropagation is very popular because it is conceptually simple, computationally efficient, and because it often works. However, getting it to work well, and sometimes to work at all, can seem more of an art than a science. Designing and training a network using backpropagation requires making many seemingly arbitrary choices such as the number and types of neurons, layers, learning rates, training and test sets, and so forth. These choices can be critical, yet there is no foolproof recipe for deciding them because they are largely problem and data dependent.*

(Yann LeCun (1998))

3

Methods and Results

Reinforcement learning is perhaps the simplest and most general way of learning, but it is also the most difficult to get right.

(Rich Sutton (1998))

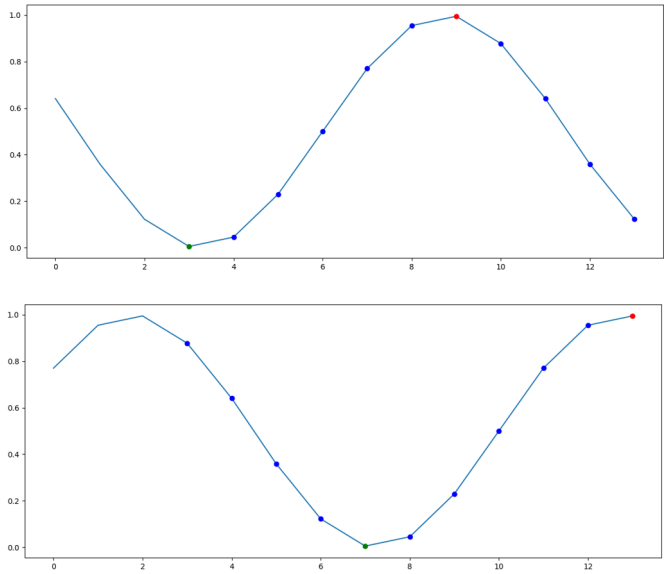
This chapter will present the price patterns used, the environments implemented, and the results obtained. The first section is dedicated to reveal the different variants of sinusoids used to simulate prices during training. Afterward, each section is associated with an environment: starting with its characterization and to which the presentation of the results we acquired with it follows. We will characterize the environments through their observation space, action space, action consequences, and reward function. The presentation of results will consist of graphics of the evolution of the episodic reward throughout the training and examples of episodes obtained with the trained weights for each task. The tuning configuration to which each of the results refers will is also presented. For the environments II and III, we will also present the evolution of the profit and the number of times the agent chooses a forbidden action per episode throughout training.

3.1 Price Patterns

There are four price patterns with which we intended to challenge the agent in each environment. In each image shown below, each point is either green, blue, or red, a reference to the agent's chosen action at that point: green means the agent decided to *buy*, red means the agent decided to *sell*, and blue is associated with the *hold* action (the agent neither buys nor sells). In the environments II and III, each action will include a volume value relative to how many units are being traded at each point. We associate a negative value to the movement of stock units from the wallet, positive volume values into the wallet (respectively selling and buying), and a "err" signal when the agent chooses a forbidden action and is therefore forced to hold. The action space and action consequences are specific for each environment and are explained in the sections below. The first points without a dot are part of the first observation of the agent.

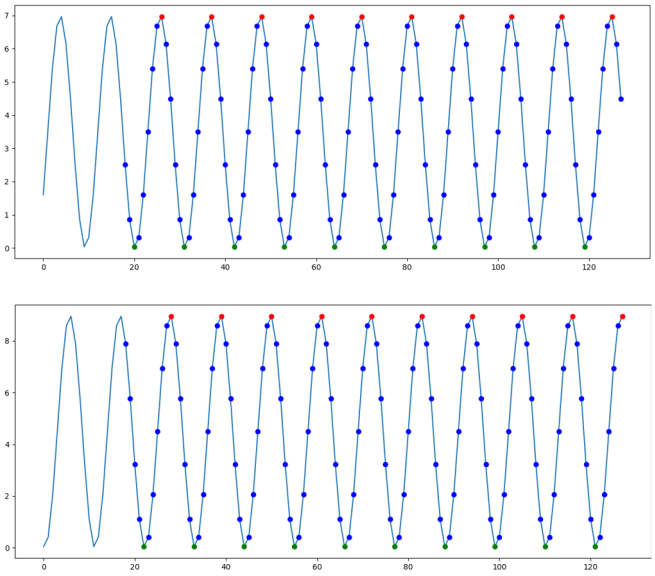
- **Fixed Sinusoid**: Points given by a sine function that are fixed throughout a training session. The agent trains repeatedly on the same episode.

Figure 3.1 Examples of episodes relative to a training session with a fixed sinusoid (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



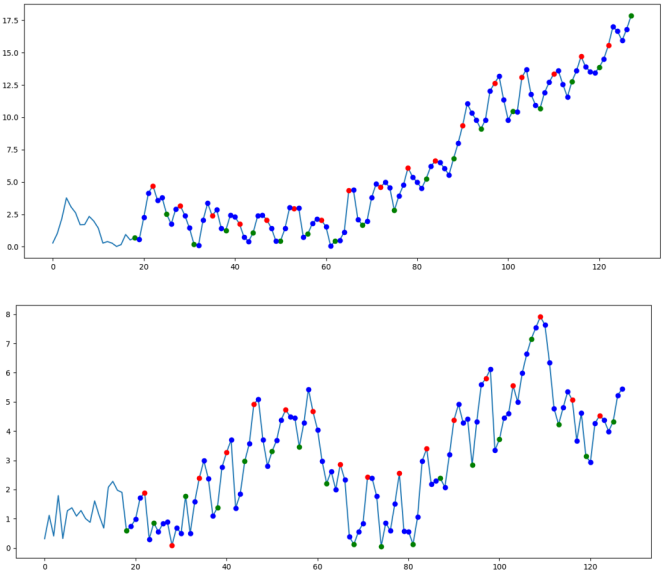
- **Changing Sinusoid**: Points are given by a sine function starting at a different phase and multiplied by a constant (integer between 1 and 10) at each episode.

Figure 3.2 Examples of episodes relative to a training session with a changing sinusoid (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



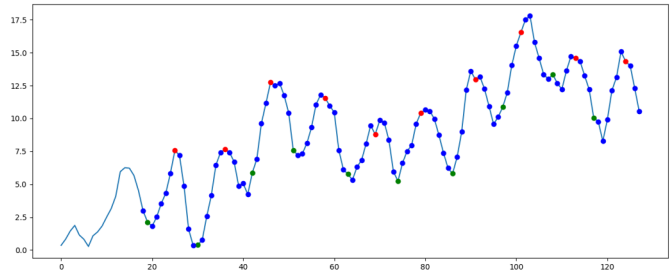
- **Brownian Path added to a Fixed Sinusoid**: A brownian path, changing throughout the training, to which points given by a sine function, fixed throughout a training session, is added.

Figure 3.3 Examples of episodes relative to a training session with brownian path added to a fixed sinusoid (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



- **Brownian Path added to a Changing Sinusoid**: A brownian path, changing throughout the training, to which points given by a sine function, starting at a different phase and multiplied by a random constant (float sampled from a uniform distribution between 0 and 2) for each episode, is added.

Figure 3.4 Example of an episode with brownian path with an added changing sinusoid (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



3.2 Environment I

- **Observation Space**: At each time step, the agent will look at a certain number of prices and the differences between prices from the past. We call to the number of time steps the agent looks back to obtain its data window size. Formally this means:

$$o_t = [[p_{t-ws}, \Delta_{t-ws}], ..., [p_t, \Delta_t]], \tag{3.1}$$

$$p_i \in \mathbf{R},\ \Delta_i \in \mathbf{R},\ ws \in \mathbf{Z},\ t \in \mathbf{Z},\ t < l$$

  Where p_i is the price of the stock at the time step i, Δ_i is $p_i - p_{i-1}$, ws is the window size of the observation, t is the current time step of the agent, and l is the episode's length, fixed for every trained session.

- **Action Space**: At each timesteps two actions are available: *Buy* and *Sell*

- **Action's Consequences**: The consequence of an action will depend on a variable we will call action state. It has two possible states: 0 and 1. This variable disables the agent from buying or selling repeatedly without doing the other possible action in between. That means if the agent tries to sell while *action state* = 0 or buy while *action state* = 1 it will be forced to hold. On the other hand, if the agent selects the action *Sell* while *action state* = 1 or buy while *action state* = 0, then the agent will effectively *Sell*/*Buy*, and the action state will be changed into the other possible value. The action state starts the episode with a value of 0.

- **Rewards**: Upon buying, the price of the stock at the time will be collected. When selling, the agent will receive a reward equivalent to the difference between the price at which the agent bought the stock and the price at which it was sold. Except for episodes containing only a fixed sinusoid throughout training, the reward of the episode will additionally be divided by the episode's average price.

3.2.1 Results

3.2.1.1 Results from training sessions with a fixed sinusoid price pattern

Results from training sessions with episodes with 11 timesteps

The tuning obtained that achieved the best performance for this case was (consider that a neural network with an architecture [x, y, z] has three layers, x neurons on the first layer, y on the second and z on the third):

Window Size	3
Adam's ϵ parameter	1e-4
Learning Rate	0.004
Policy Network Architecture	[32,32]
Value Network Architecture	[32,32]
Batch Size	32

The agent with this tuning was able to converge at the global maximum of the episode. Below is an episode where the agent was able to achieve the maximum reward and the reward along the number of episodes completed throughout the training.

Figure 3.5 Episode where the agent was able to get the maximum reward possible (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

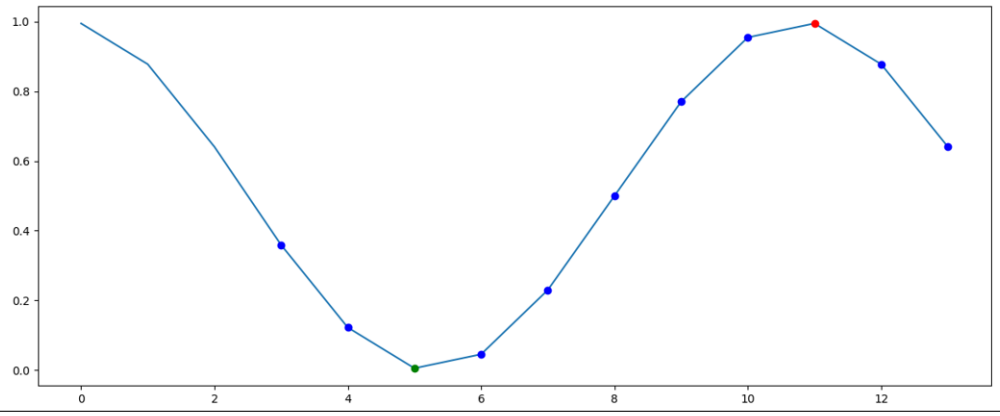
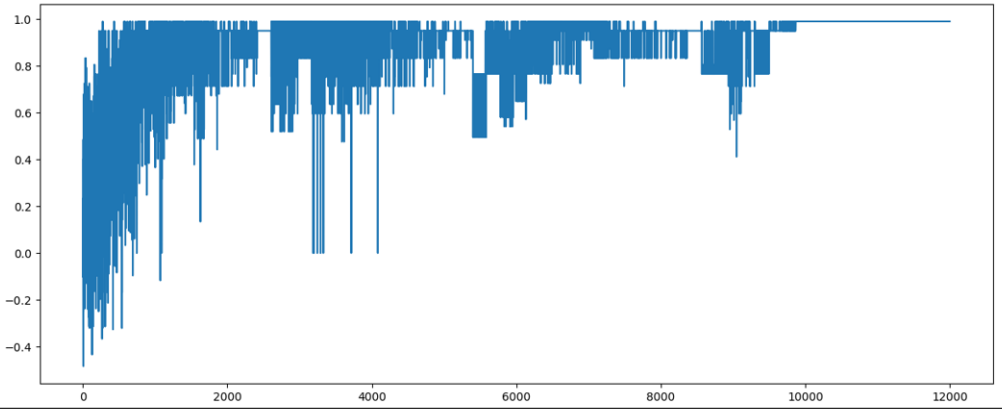
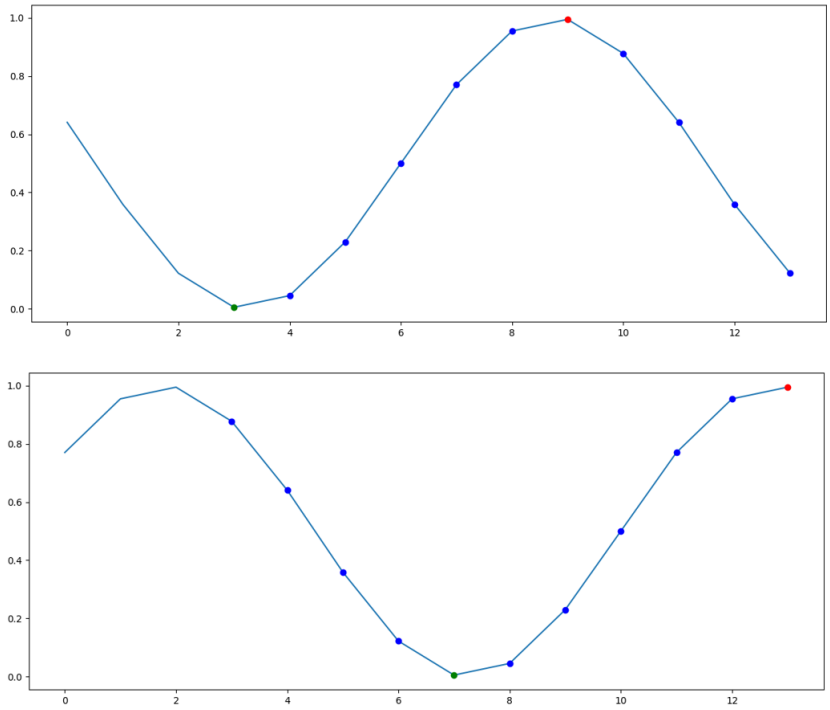


Figure 3.6 Evolution of the episodic reward obtained by the agent throughout training (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)



The obtained agent also showed to be able to generalize its performance for testing episodes' sinusoids with different starting phases.

Figure 3.7 Two examples of episodes with price patterns starting at different phases from the fixed sinusoid used to train the weights of the model used in these episodes (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



Results from training sessions with episodes with 55 timesteps

The tuning obtained that achieved the best performance for this case was:

Window Size	6
Adam's ϵ parameter	1e-4
Learning Rate	0.004
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

The agent was here again able to converge at the training episode's maximum. The reward throughout training and a training episode where the agent performed optimally are shown below:

Figure 3.8 Evolution of the episodic reward obtained by the agent throughout training (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)

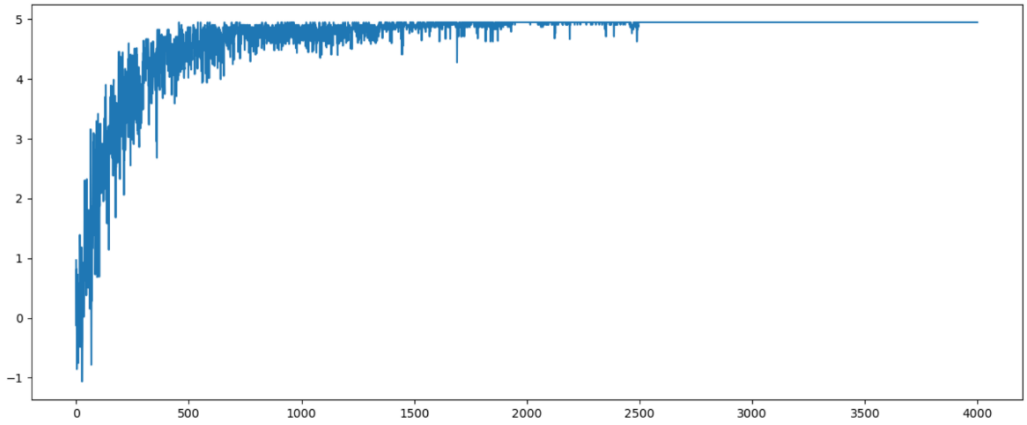
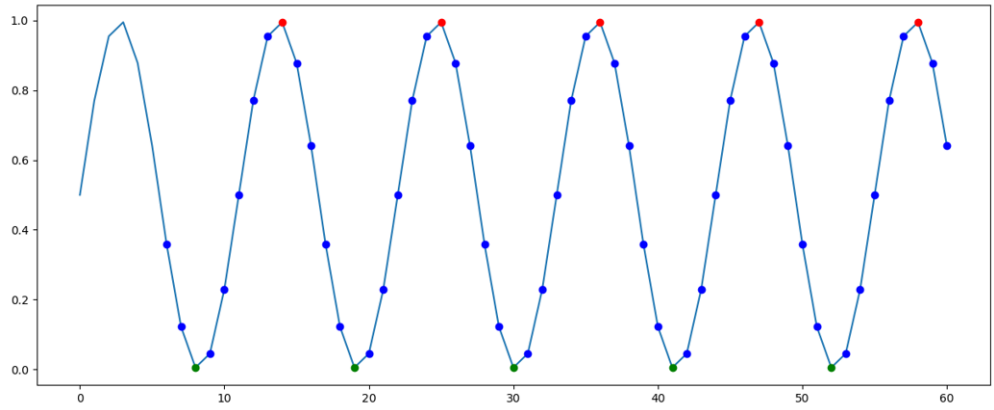
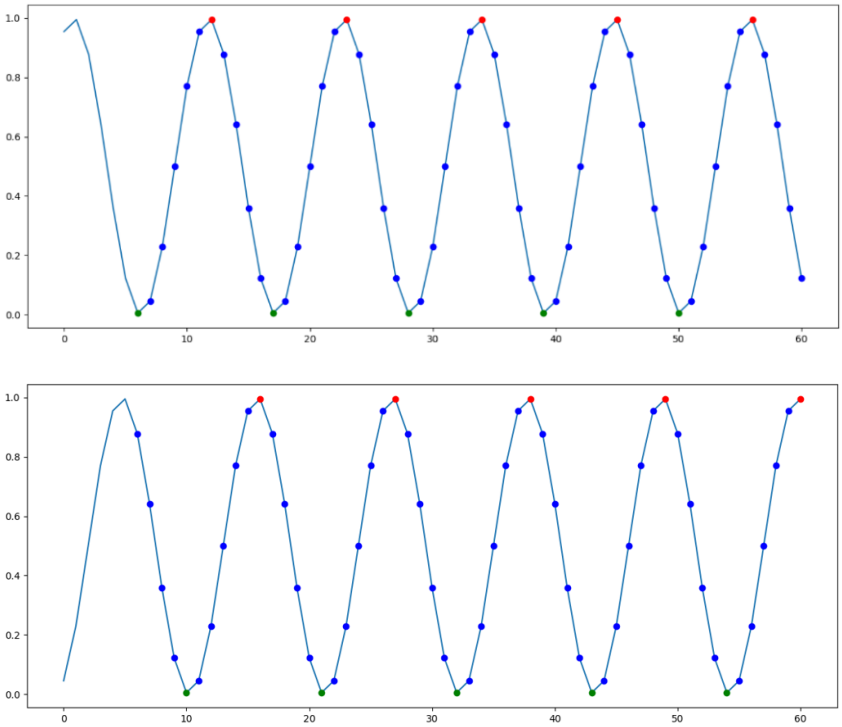


Figure 3.9 Episode where the agent was able to get the maximum reward possible (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



Analogous to the previous case, the obtained agent also showed to be able to generalize its performance for testing episodes' sinusoids with different starting phases.

Figure 3.10 Two examples of episodes with price patterns starting at different phases from the fixed sinusoid used to train the weights of the model used in these episodes (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



Results from training sessions with episodes with 110 timesteps

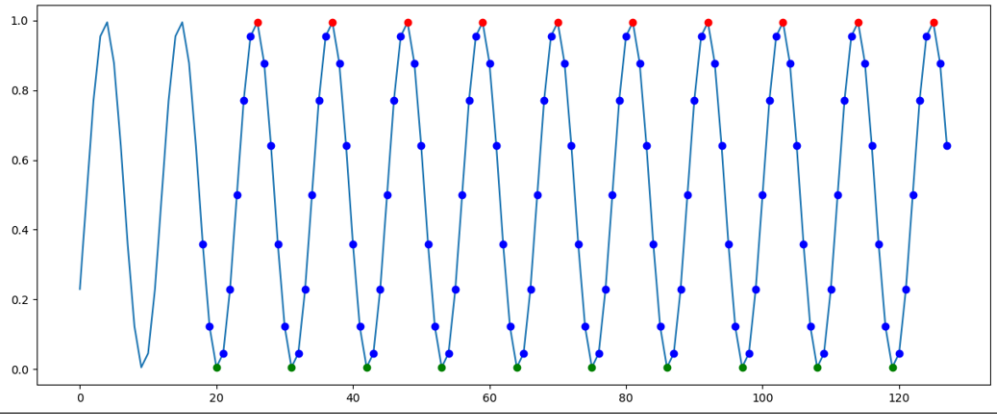
The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.006
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

Again the agent was able to converge on the policy capable of consistently achieving the maximum reward in the training episodes. Below is the training episode with the sequence of optimal actions chosen by the converged policy and reward throughout training.

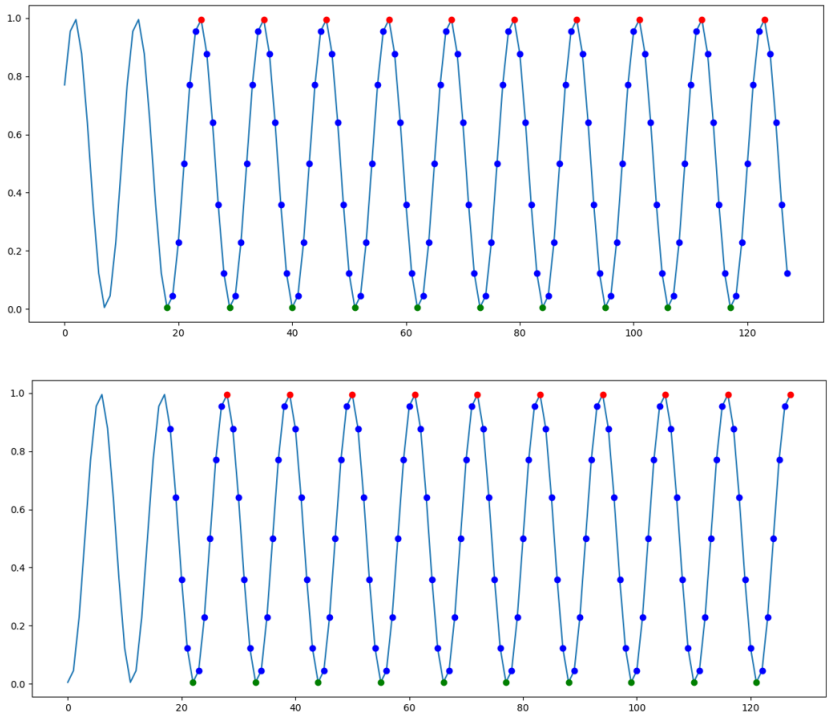
Figure 3.11 Evolution of the episodic reward obtained by the agent throughout training (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)

Figure 3.12 Episode where the agent was able to get the maximum reward possible (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



Again, the obtained agent also showed to be able to generalize its performance for testing episodes' sinusoids with different starting phases.

Figure 3.13 Two examples of episodes with price patterns starting at different phases from the fixed sinusoid used to train the weights of the model used in these episodes (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



3.2.1.2 Results from training sessions with a changing sinusoid price pattern

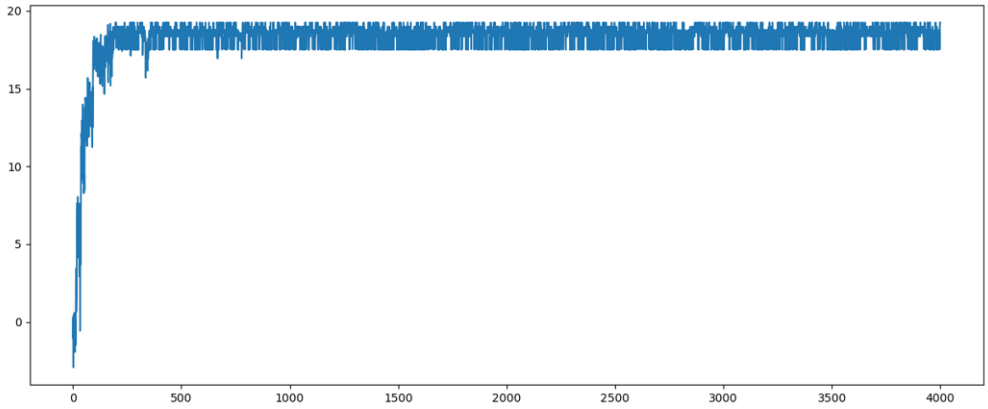
Results from training sessions with episodes with 110 timesteps

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.006
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

The episodic reward obtained throughout the training is shown below.

Figure 3.14 Evolution of the episodic reward obtained by the agent throughout training (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)

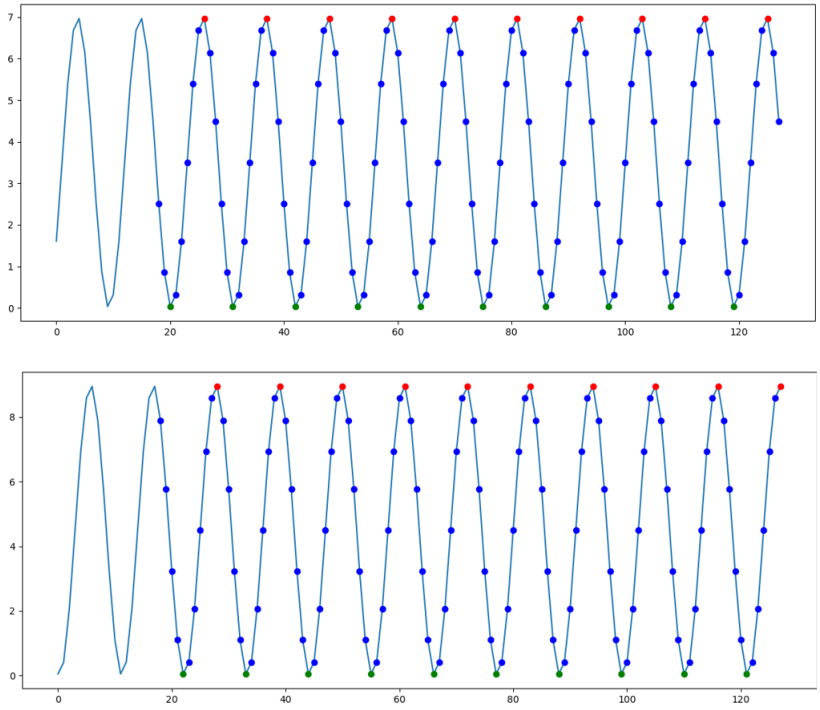


Better results were obtained by using the model trained for a fixed sinusoid with 110 timesteps. It showed to consistently be able to achieve optimal scores in the episodes to which it was presented. Below is a table comparing the average reward(μ), reward's standard deviation(σ), and median during 1000 test episodes of the agent with the trained weights for episodes with a fixed sinusoid and the agent trained with the best found tuning for episodes with a changing sinusoid, both with 110 timesteps.

	μ	σ	median
Policy trained on a fixed sinusoid	19.1	0.568	19.4
Policy trained on a changing sinusoid	18.5	0.571	18.7

Table 3.1: Table comparing the performance of two policies trained on episodes from the first environment with 110 timesteps, one with a fixed sinusoid price pattern and another with a changing sinusoid price pattern on test episodes from the first environment with 110 timesteps and changing sinusoid price patterns (the μ column refers to the average of the episodic reward, the σ to its standard deviation and the third column to its median obtained after running 1000 test episodes with 110 time steps and prices following the changing sinusoid pattern)

Figure 3.15 Two examples of test episodes with a changing sinusoid between episodes while using the model trained with a fixed sinusoid throughout the training (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



3.2.1.3 Results from training sessions with a brownian path with an added fixed sinusoid price pattern

Results from training sessions with episodes with 110 timesteps

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.004
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

Below is the episodic training reward for the given tuning at this task, and an example of a testing episode the agent completed after training.

Figure 3.16 Evolution of the episodic reward obtained by the agent throughout training (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)

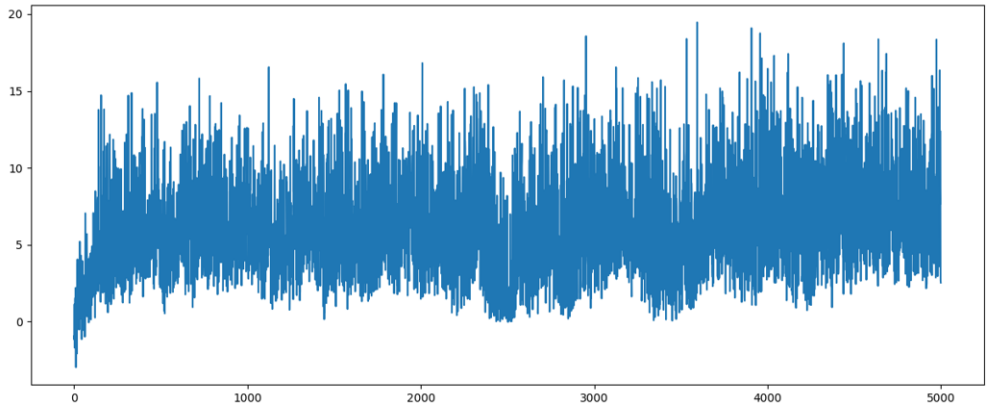
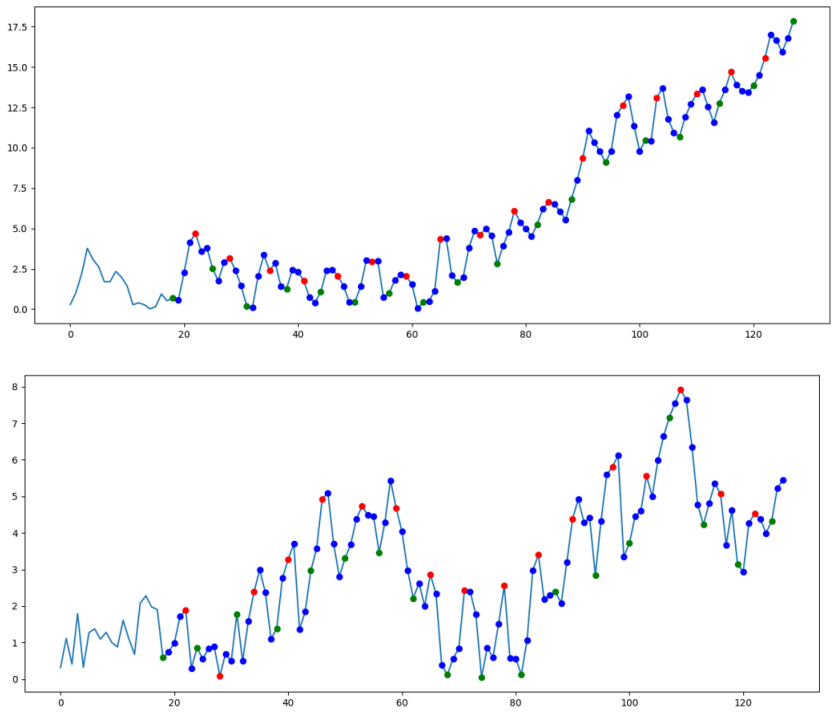


Figure 3.17 Two examples of test episodes with brownian path added to a fixed sinusoid (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



We ran 1000 testing episodes to collect the mean reward of the resulting trained agent and compare its performance with a random policy. Below are the results. We also compared it with the performance of the policy trained on episodes with a fixed sinusoid price pattern and 110 timesteps.

	μ	σ	median
Policy trained on a fixed sinusoid	0.563	1.23	0.492
Random policy	0.859	1.46	0.834
Policy trained on brownian paths added to a fixed sinusoid	7.42	3.29	6.95

Table 3.2: Table comparing the performance of two policies trained on episodes from the first environment with 110 timesteps (one trained on a fixed sinusoid price pattern and another on a brownian path added to a fixed sinusoid price pattern) and a random policy on test episodes from the first environment with 110 timesteps and a brownian path added to a fixed sinusoid price pattern (the μ column refers to the average of the episodic reward, the σ to its standard deviation and the third column to its median obtained after running 1000 test episodes with 110 time steps and prices following the changing sinusoid pattern)

3.2.1.4 Results from training sessions with a brownian path with an added changing sinusoid price pattern

Results from training sessions with episodes with 110 timesteps

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0004
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

Below is the episodic training reward for the given tuning at this task, and an example of a testing episode the agent completed after training.

Figure 3.18 Evolution of the episodic reward obtained by the agent throughout training for a brownian path added to a changing sinusoid episode with 110 timesteps (the vertical axis represents the episodic reward, the horizontal axis the number of training episodes completed by the agent)

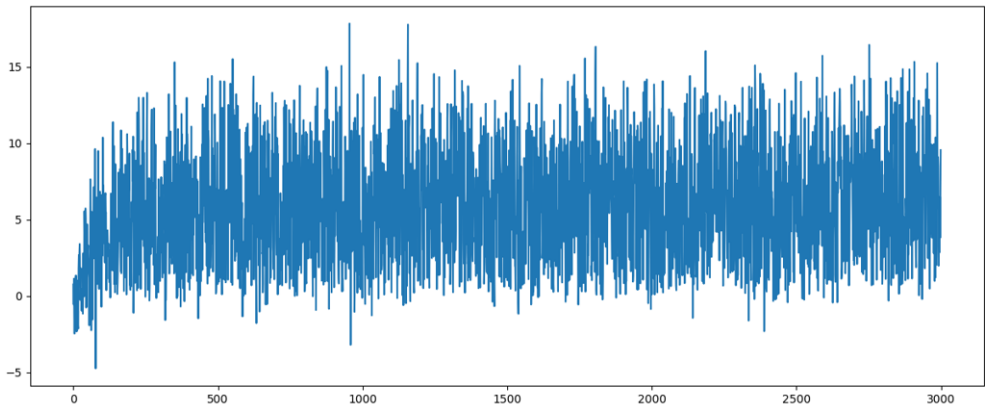
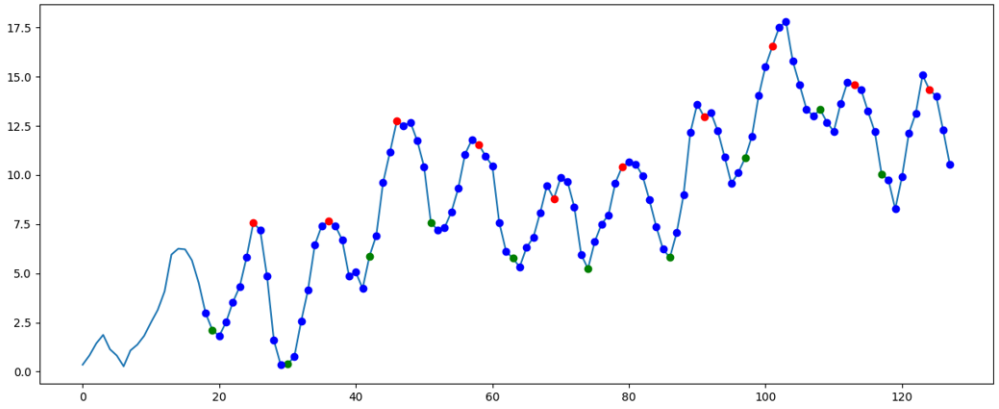


Figure 3.19 Example of a test episode with a brownian path added to a changing sinusoid episode with 110 timesteps (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)



We ran 1000 testing episodes to collect the mean reward of the resulting trained agent and compare its performance with a random policy.

	μ	σ	median
Random policy	0.311	1.31	0.366
Policy trained on brownian paths added to a changing sinusoid	6.12	3.75	5.87

Table 3.3: Table comparing the performance of one policy trained on episodes from the first environment with 110 timesteps and a brownian path added to a changing sinusoid price pattern and a random policy on test episodes from the first environment with 110 timesteps and a brownian path added to a changing sinusoid price pattern (the μ column refers to the average of the episodic reward, the σ to its standard deviation and the third column to its median obtained after running 1000 test episodes)

3.3 Environment II

- **Observation Space**: At each time step, the agent will look at a certain number of prices and the differences between prices from the past. It will also include the amount of cash and the number of units of stock available to sell at the time step of the observation. Formally this means:

$$o_t = [[p_{t-ws}, \Delta_{t-ws}], ..., [p_t, \Delta_t], [c_t, n_t]], \tag{3.2}$$

$$p_i \in \mathbf{R},\ \Delta_i \in \mathbf{R},\ ws \in \mathbf{Z},\ t \in \mathbf{Z},\ t < l,\ c \in \mathbf{R},\ n_i \in \mathbf{Z}$$

  Where p_i is the price of the stock at the time step i, Δ_i is $p_i - p_{i-1}$, ws is the window size of the observation, t is the current time step of the agent and l is the episode's length (fixed for every trained session), c_i and n_i are the budget available and the number of stocks available at the timestep i.

- **Action Space**: At each time step, each action takes the form:

$$a_t = [act_t, v_t], \tag{3.3}$$

$$act_i \in \{Buy, Sell\},\ v_i \in \{0, 1, ..., MT-1, MT\},\ MT \in \mathbf{Z}$$

  Where the first element of the action act_t is the type of the action chosen at the time step t, v_t the volume included in that action, and MT the maximum number of units the agent is allowed to move at each time step.

- **Action's Consequences**: The consequence of the action will depend on the current cash and the number of stock units available to sell. When the agent tries to sell more than what it has or buy more than the current cash allows, it is forced to hold. Hold also happens when the agent chooses action with volume 0. When the agent chooses to buy an allowed volume for that time step, it will save the price and the number of units bought. When the agent buys again, it will check if it already has stored units bought with the same price. If it has, the volume is added to that stored element. If it doesn't, a new element is created with the bought volume. When the agent selects an authorized sale, it will look for the element with the highest associated buying price to take units from it first.

- **Rewards**: Upon buying, the price of the stock at the time will be collected. When selling, the agent will receive a reward equivalent to the difference between the price at which the stock was sold and the price at which it was bought multiplied by the volume of units sold. Except for episodes containing only a fixed sinusoid throughout training, the reward of the episode will additionally be divided by the episode's average price. A penalty will also be added each time the agent chooses a forbidden action.

3.3.1 Results

All episodes in the section are going to have 110 timesteps. That episode characteristic will be omitted from now on. Additionally, the plots representing the episodes will, at each point, include the volume of the transaction effectuated, with a minus sign representing a volume being sold and a positive value volume being bought.

3.3.1.1 Results from training sessions with a fixed sinusoid price pattern

Results from training sessions where the agent starts the episodes with 5 units of cash, and is able to trade a maximum of 5 units of stock per timestep

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.21 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

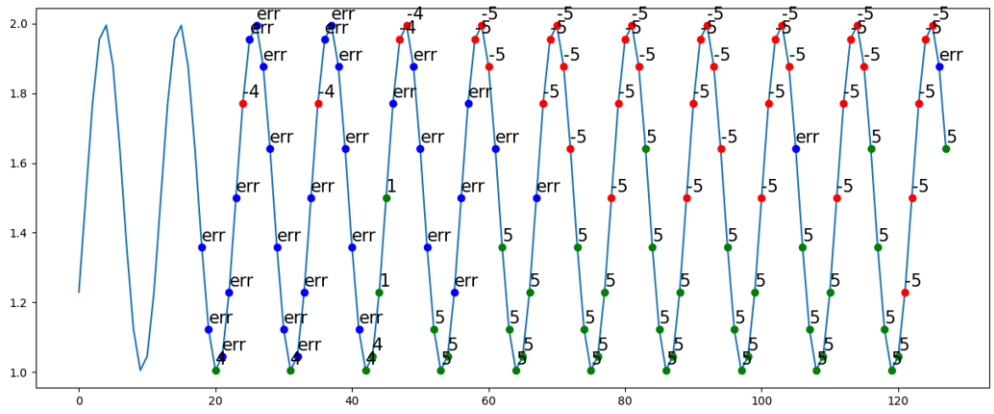
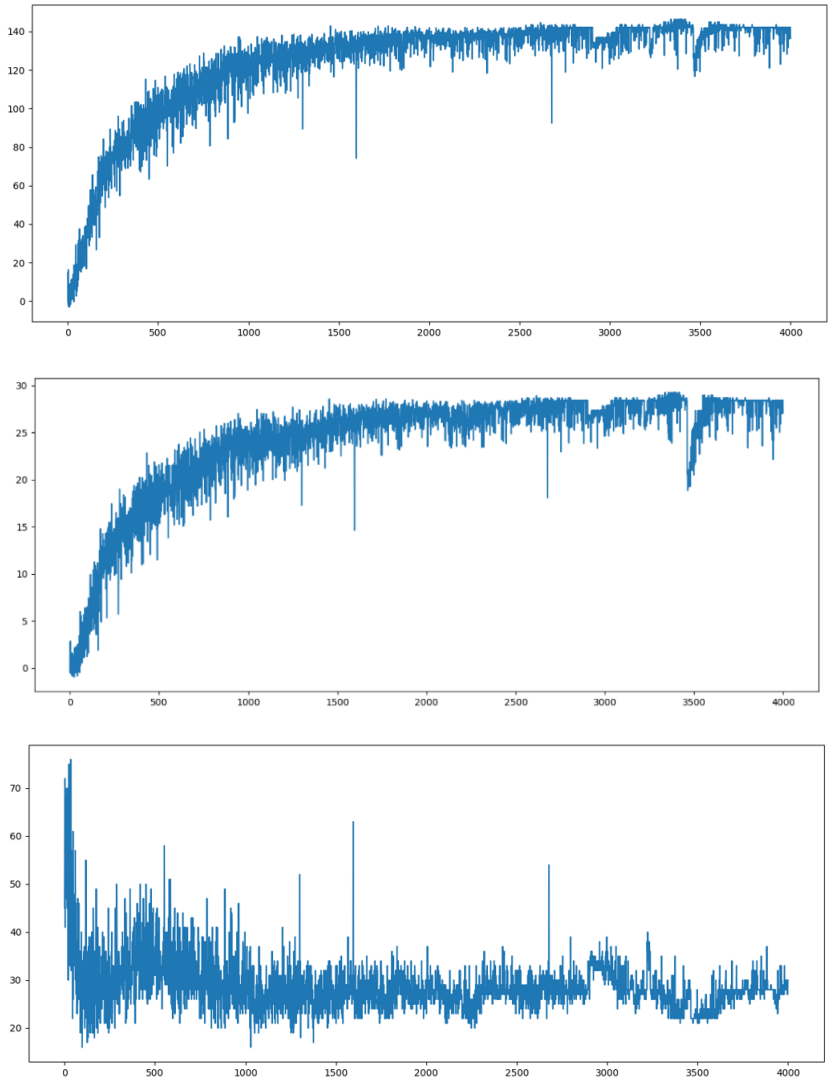


Figure 3.20 The episodic reward, profit, number of error obtained throughout training.(the vertical axis represents the episodic reward, the percentage of the extra cash the agent has given the initial budget, and the number of times the agent chooses forbidden actions per episode, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 5 units of cash, and is able to trade a maximum of 10 units of stock per timestep

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.22 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

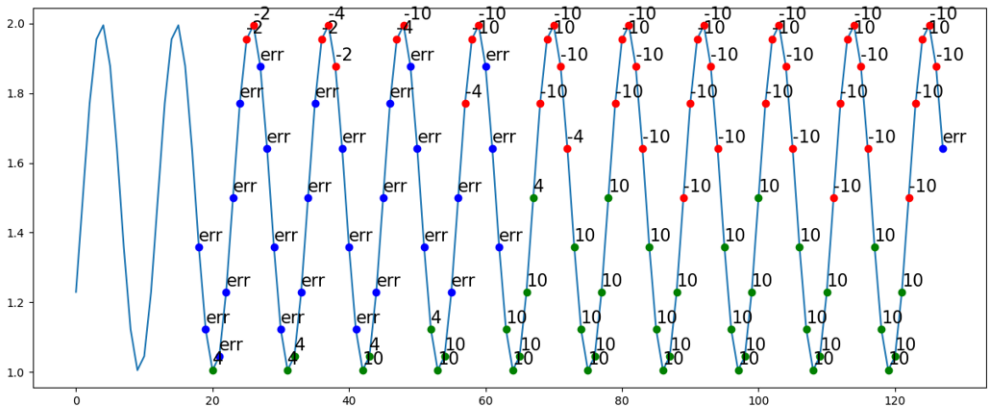
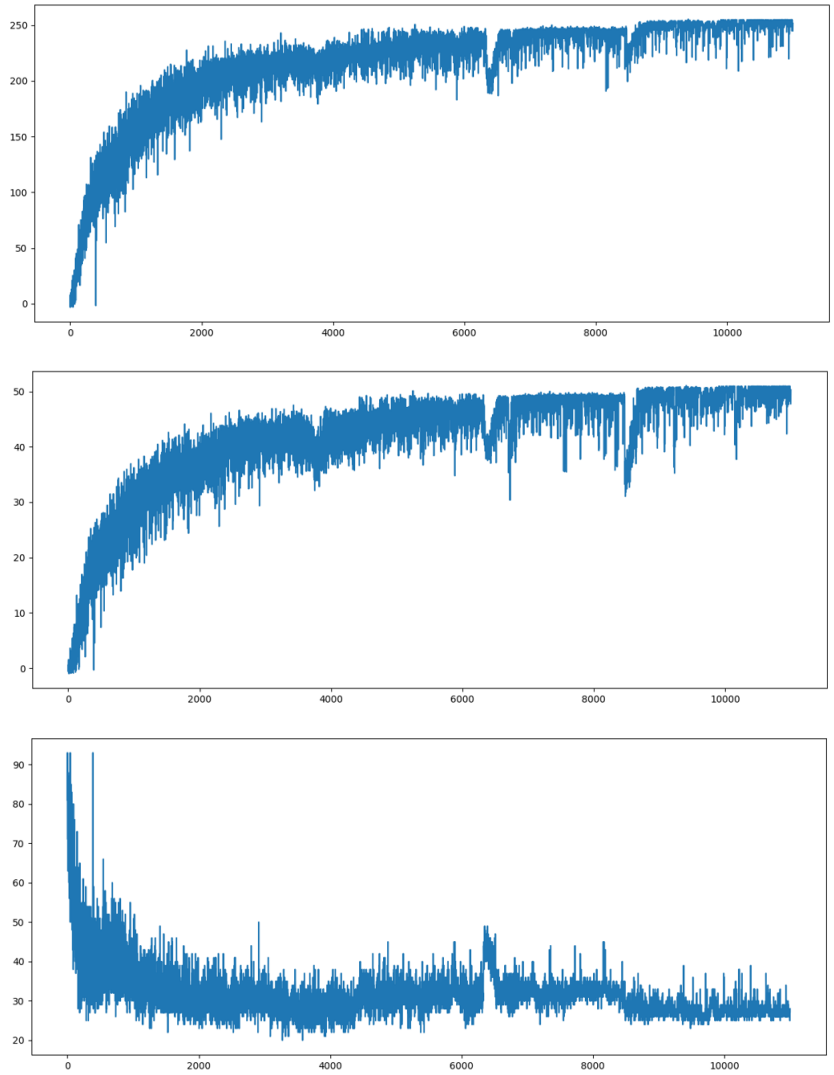


Figure 3.23 The episodic reward, profit, number of error obtained throughout training.(the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 5 units of cash, is able to trade a maximum of 10 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	0.605

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.24 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

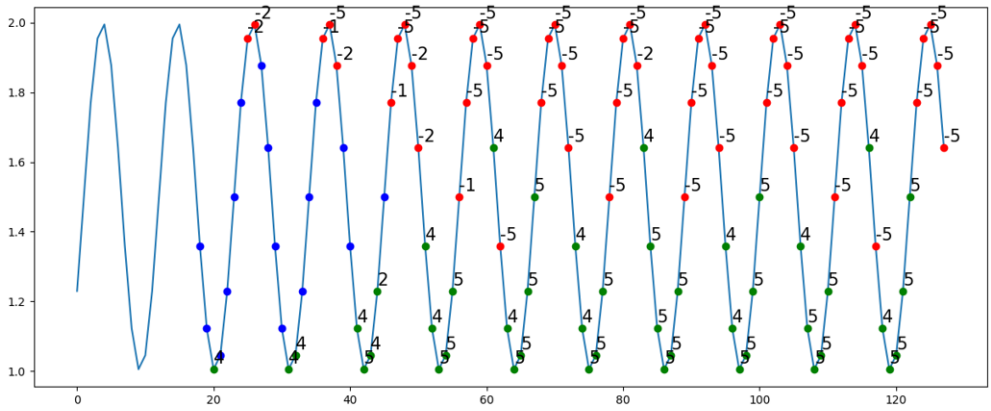
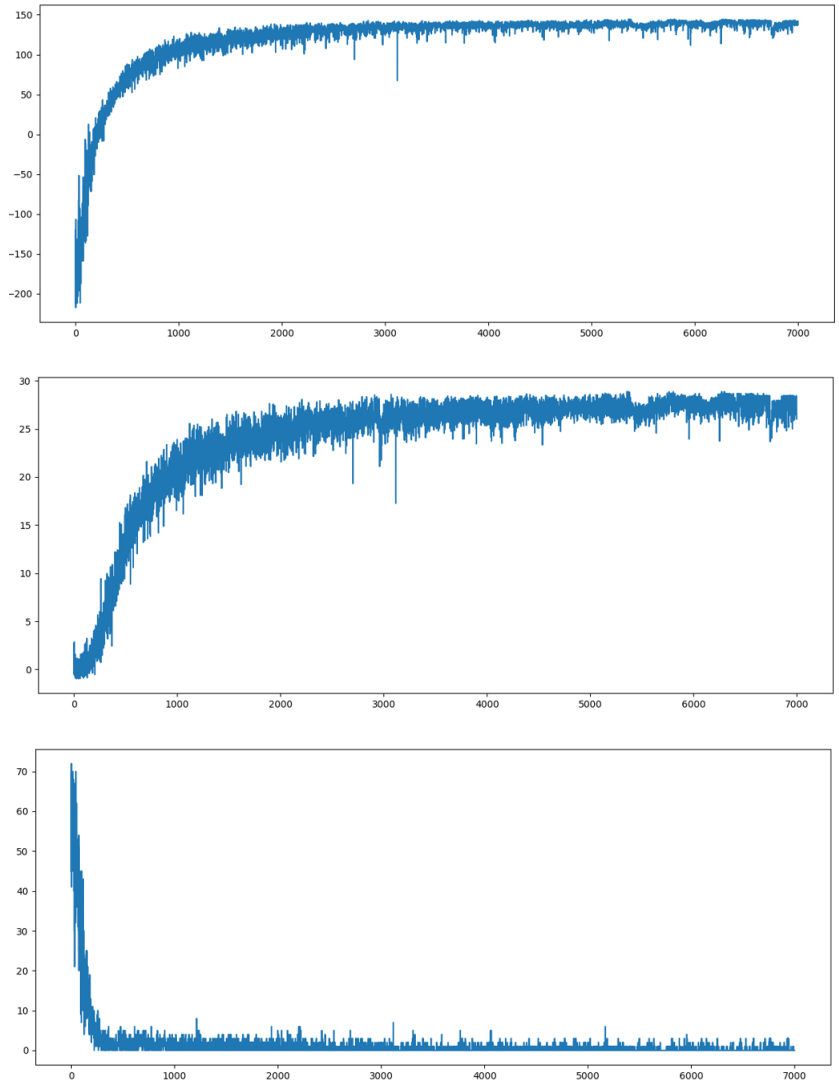


Figure 3.25 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 10 units of cash, is able to trade a maximum of 10 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	0.605

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.26 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

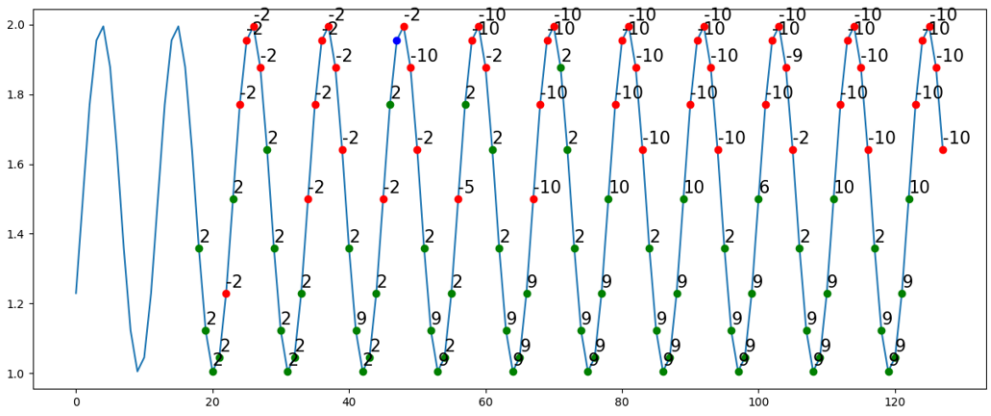
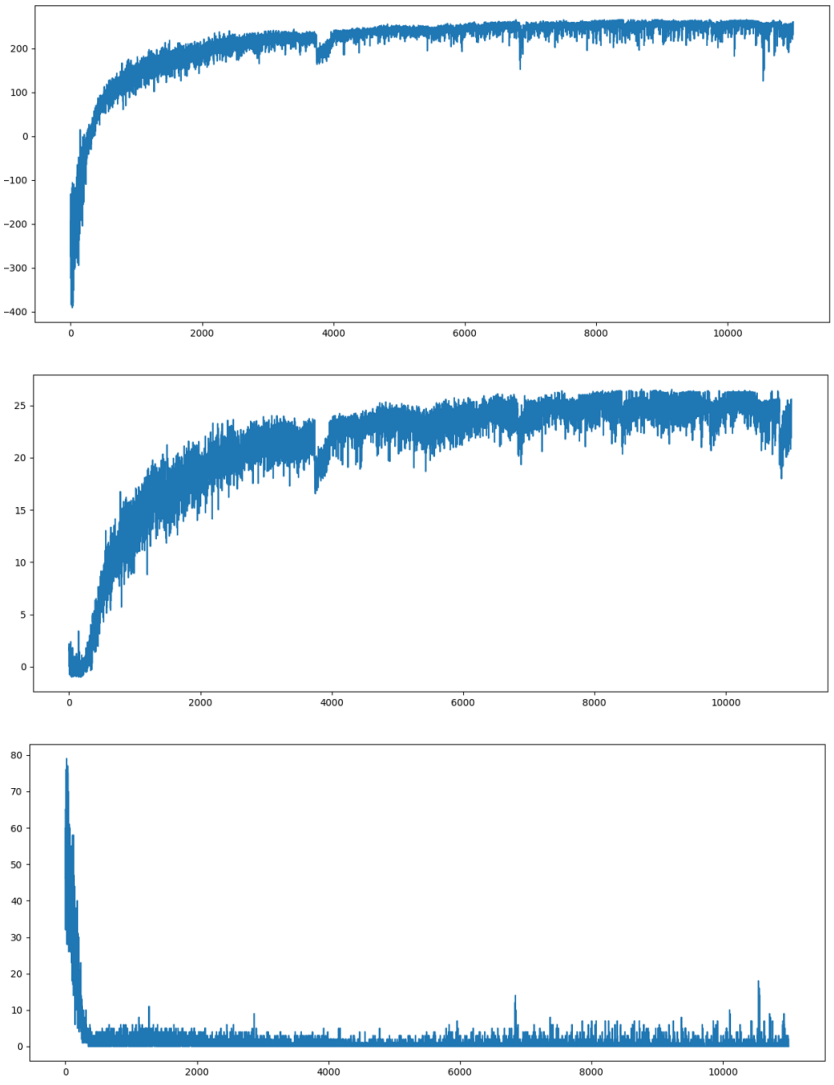


Figure 3.27 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 20 units of cash, is able to trade a maximum of 20 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	0.605

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.28 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

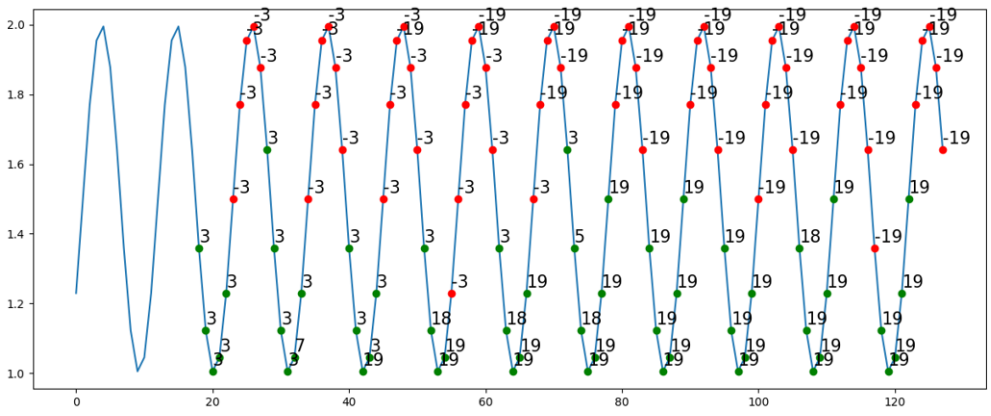
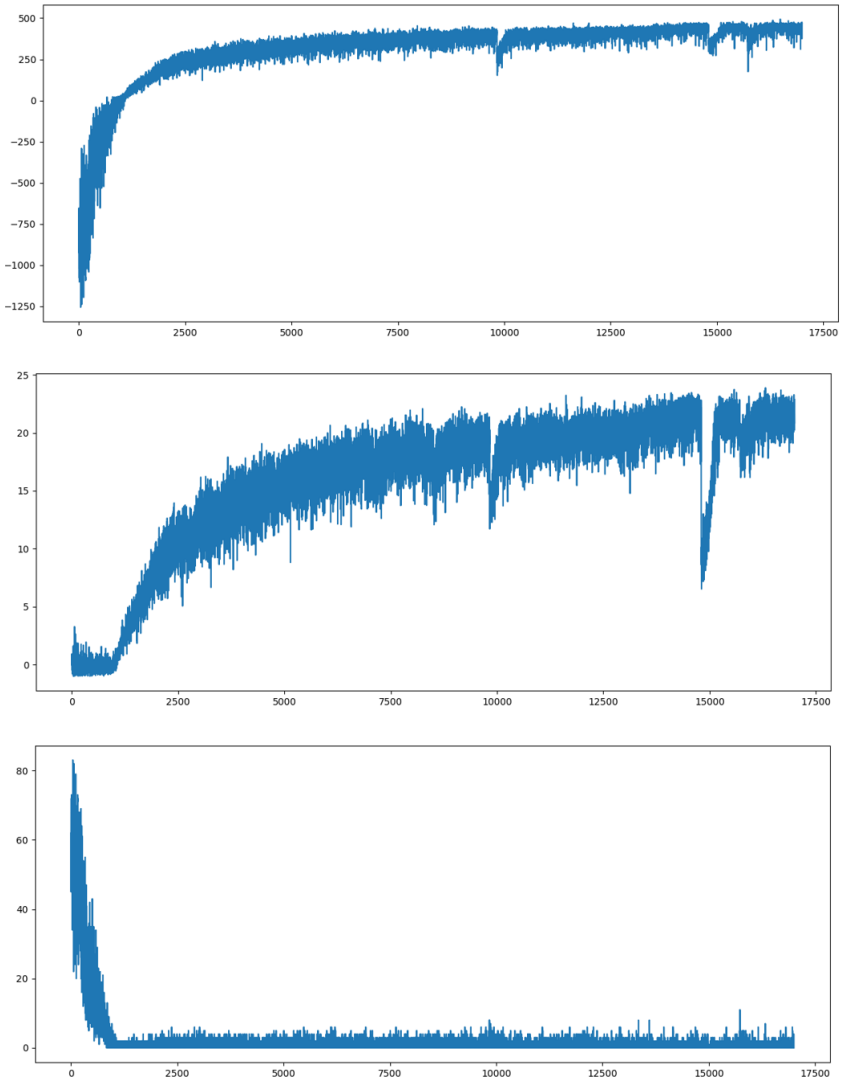


Figure 3.29 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



3.3.1.2 Results from training sessions with a changing sinusoid price pattern

Results from training sessions where the agent starts the episodes with 5 units of cash, is able to trade a maximum of 5 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	15

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.30 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

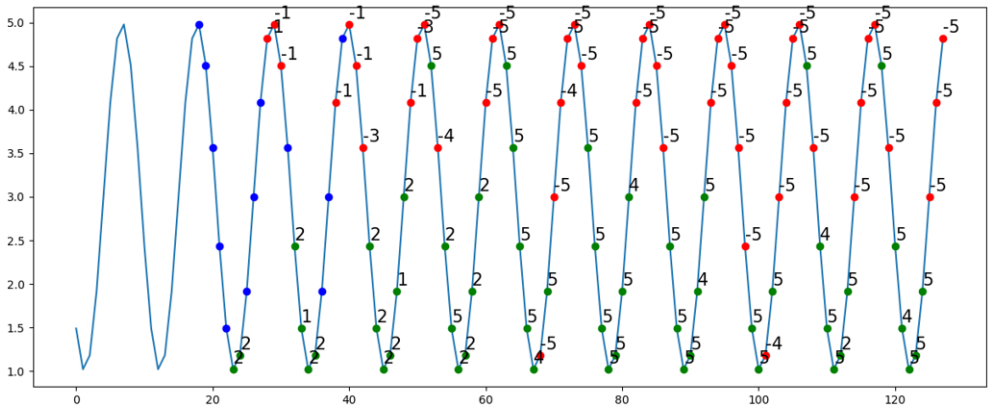
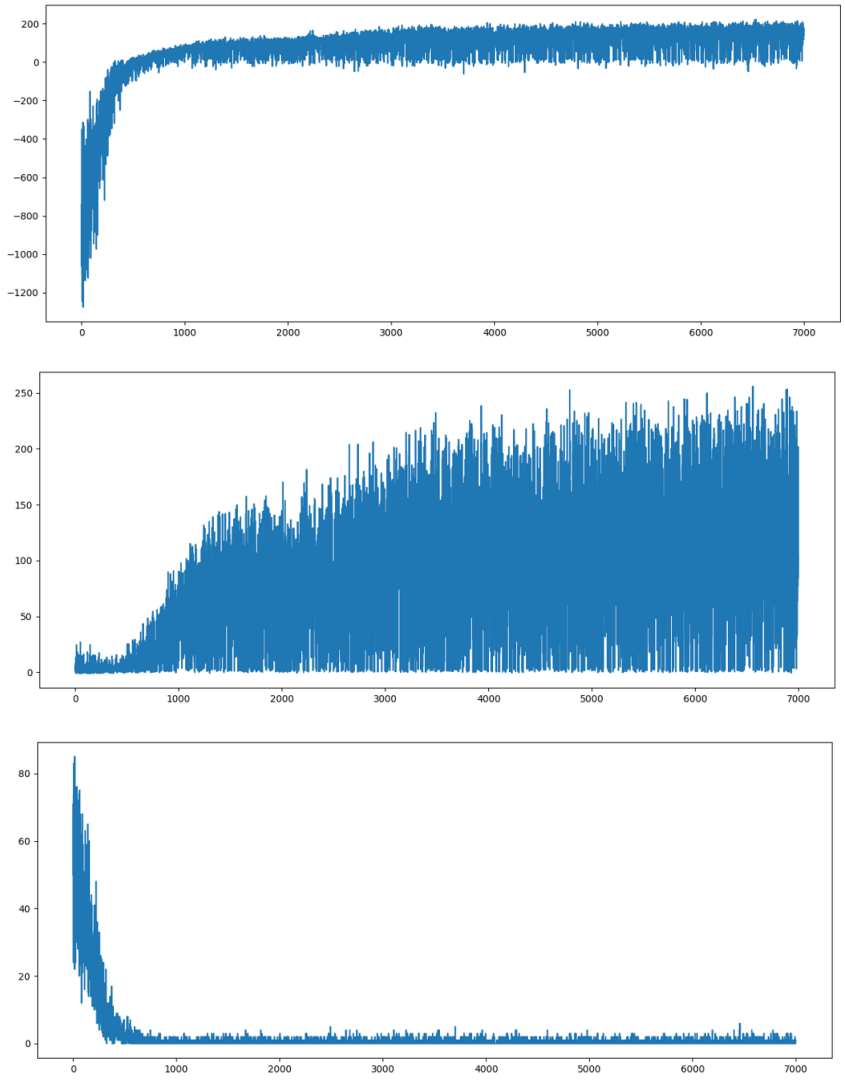


Figure 3.31 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 10 units of cash, is able to trade a maximum of 10 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	20

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.32 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

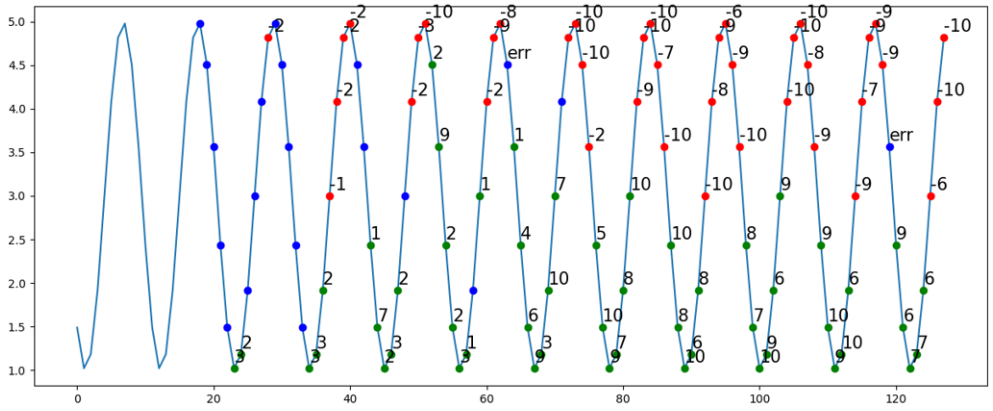
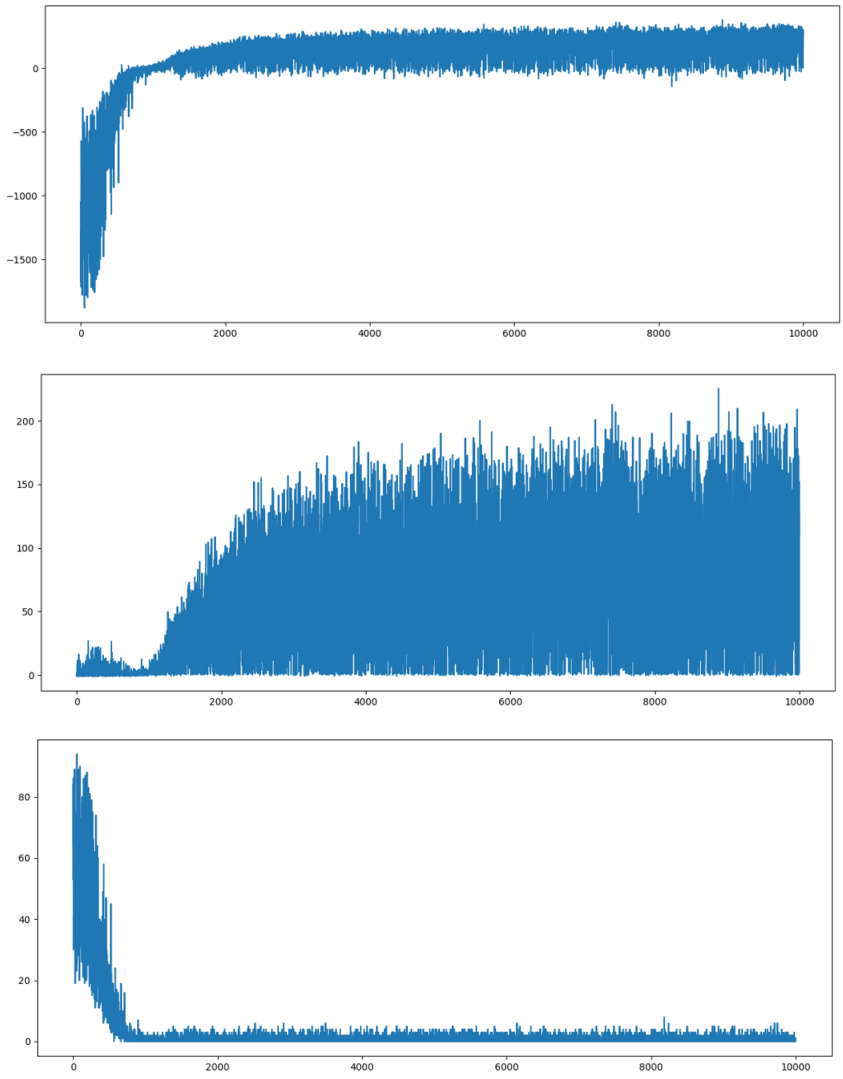


Figure 3.33 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions where the agent starts the episodes with 20 units of cash, is able to trade a maximum of 20 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	80

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.34 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

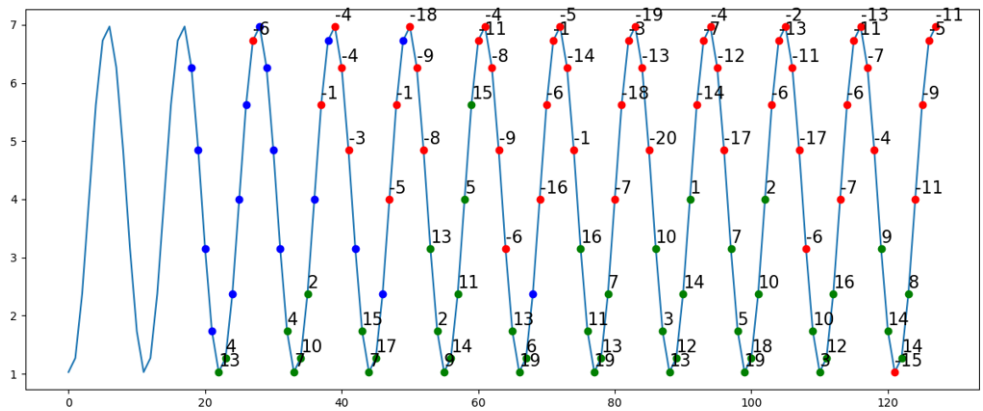
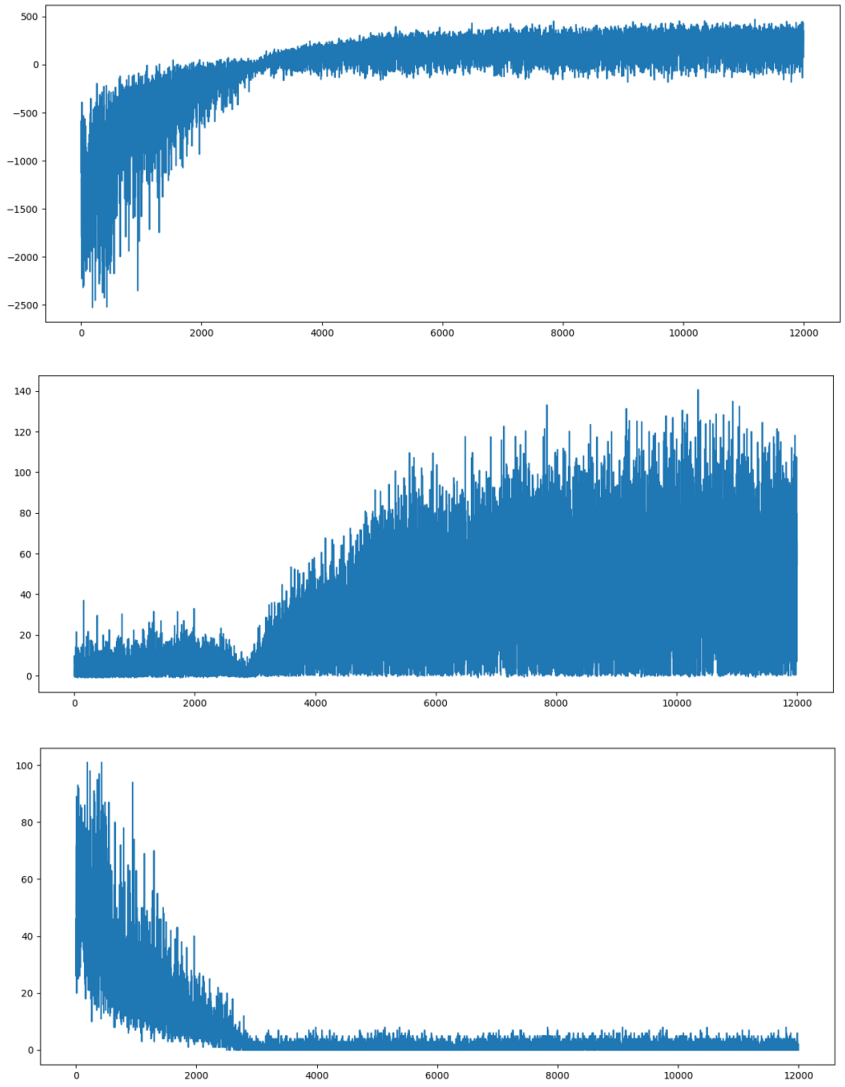


Figure 3.35 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



3.3.1.3 Results from training sessions with a brownian path with an added fixed sinusoid price pattern

Results from training sessions where the agent starts the episodes with 20 units of cash, is able to trade a maximum of 20 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	10

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.36 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

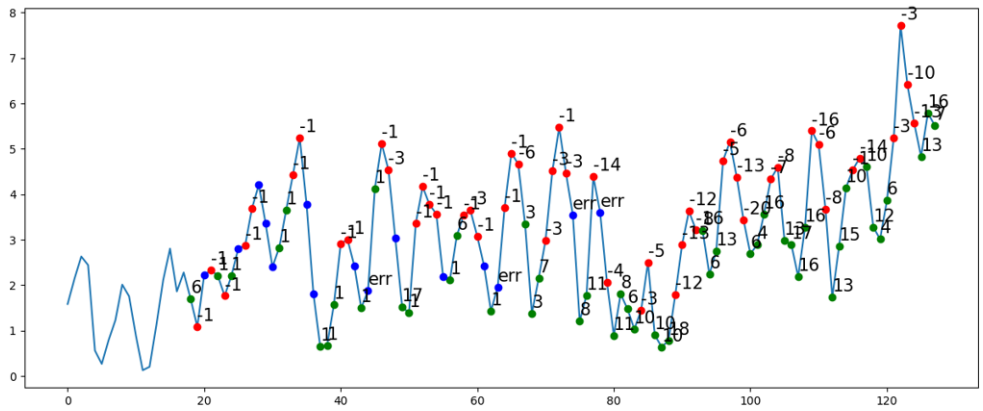
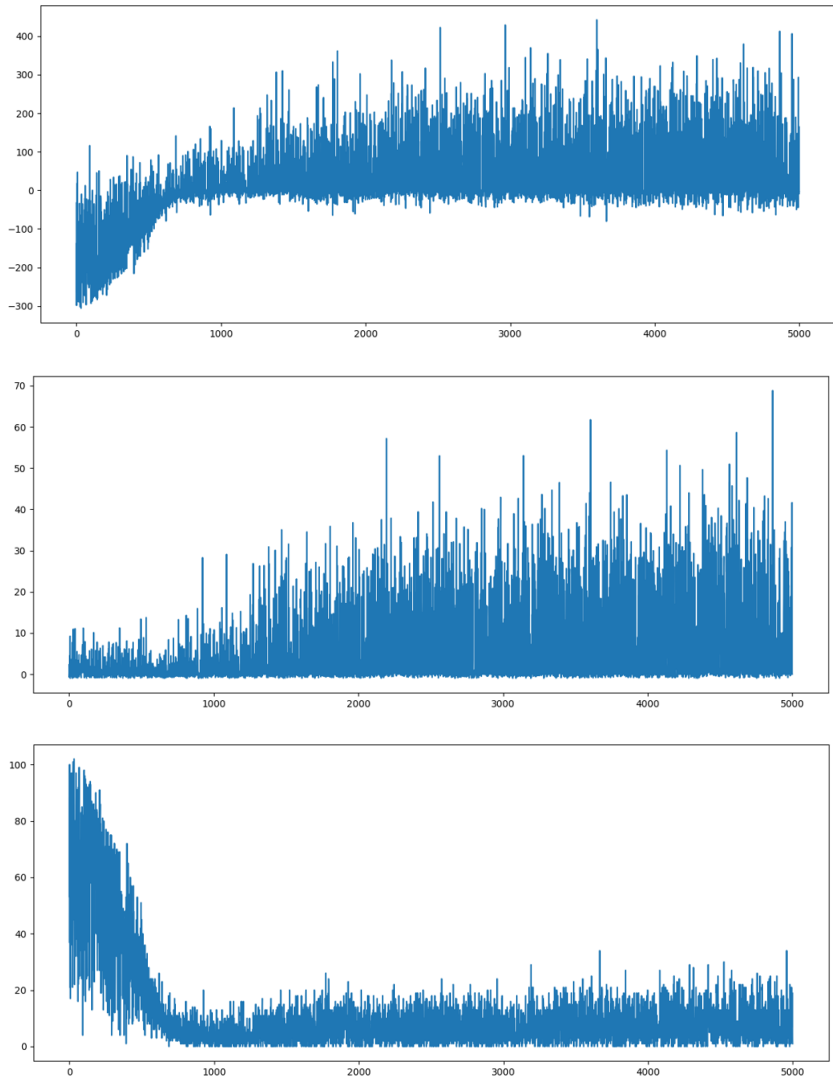


Figure 3.37 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



We ran 1000 testing episodes to collect the mean reward of the resulting trained agent and compared its performance with a random policy. Below are the results.

		μ	σ	median
Random policy	rew.	-682	270	-701
	errors	68.4	24.7	71.5
	profit	1.89	3.54	0.59
Policy trained on brownian path with added fixed sinusoid	rew.	14.2	107	-10.0
	errors	7.91	5.64	7.00
	profit	10.1	11.2	7.00

Table 3.4: Table comparing the performance of one policy trained on episodes from the second environment with a brownian path added to a fixed sinusoid price pattern and a random policy on test episodes from the first environment with 110 timesteps and a brownian path added to a fixed sinusoid price pattern (the μ column refers to the average, the σ to the standard deviation and the third column to the median of the episodic reward, profit and number of errors obtained after running 1000 test episodes with prices following a brownian path added to a fixed sinusoid price pattern)

3.3.1.4 Results from training sessions with a brownian path with an added changing sinusoid price pattern

Results from training sessions where the agent starts the episodes with 20 units of cash, is able to trade a maximum of 20 units of stock per timestep and has a non-zero penalty associated with forbidden actions

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	10

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.38 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

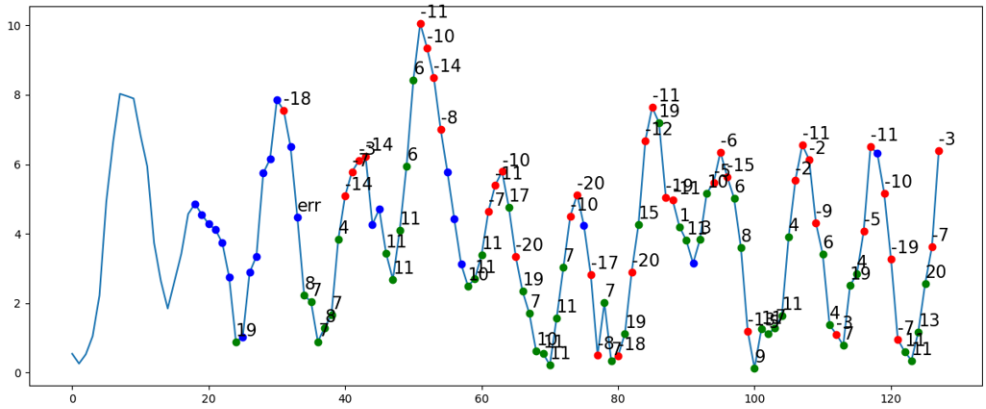
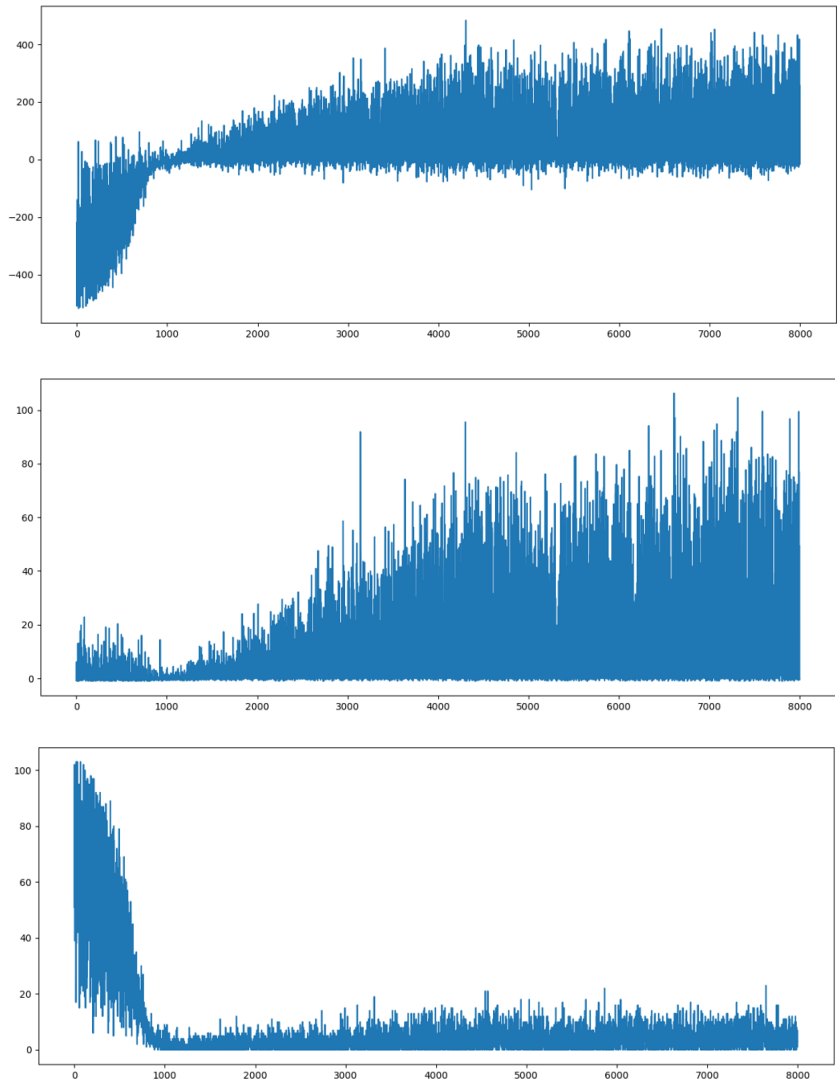


Figure 3.39 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



We ran 1000 testing episodes to collect the mean reward of the resulting trained agent and compare its performance with a random policy. Below are the results.

		μ	σ	median
Random policy	rew.	-667	266	-704
	errors	70.0	24.2	72.0
	profit	2.63	5.08	0.683
Policy trained on brownian path with added changed sinusoid	rew.	78.3	126	22.2
	errors	3.96	3.36	3.00
	profit	18.9	22.5	7.74

Table 3.5: Table comparing the performance of one policy trained on episodes from the second environment with a brownian path added to a changing sinusoid price pattern and a random policy on test episodes from the first environment with 110 timesteps and a brownian path added to a fixed sinusoid price pattern (the μ column refers to the average, the σ to the standard deviation and the third column to the median of the episodic reward, profit and number of errors obtained after running 1000 test episodes with prices following a brownian path added to a changing sinusoid price pattern)

3.4 Environment III

- **Observation Space**: At each time step, the agent will look at a certain number of prices and the differences between prices from the past. It will also include the amount of cash and the number of units of stock available to sell at the time step of the observation. It will include this information for each stock. Formally this means:

$$o_t = [[[p_{1,t-ws}, \Delta_{1,t-ws}], ..., [p_{1t}, \Delta_{1,t}], [c_t, n_{1,i}]], ..., [[p_{N,t-ws}, \Delta_{N,t-ws}], ..., [p_{N,t}, \Delta_{N,t}], [c_t, n_{N,t}]]], \tag{3.4}$$

$$p_{j,i} \in \mathbf{R},\ \Delta_{i,j} \in \mathbf{R},\ ws \in \mathbf{Z},\ t \in \mathbf{Z}, t < l,\ c \in \mathbf{R},\ n_{j,i} \in \mathbf{Z}, N \in \mathbf{Z}$$

  Where $p_{j,i}$ is the price of the stock j at the time step i, $\Delta_{j,i}$ is $p_{j,i} - p_{j,i-1}$, ws is the window size of the observation, t is the current time step of the agent and l is the episode's length (fixed for every trained session), c_i is the budget available at the time step i, $n_{j,i}$ is the number of units of the stock j available at the timestep i and N is the number of stocks included in the episode.

- **Action Space**: At each time step, each action takes the form:

$$a_t = [[act_{0,t}, v_{0,t}], ..., [act_{N,t}, v_{N,t}]], \tag{3.5}$$

$$act_j \in \{Buy, Sell\},\ v_j \in \{0, 1, ..., MT-1, MT\},\ MT \in \mathbf{Z},\ N \in \mathbf{Z}$$

  Where the first element of the action $act_{j,i}$ and $v_{j,i}$ are the type action and the volume included in that action for the stock j at the time step i and MT the maximum number of units the agent is allowed to move at each time step.

- **Action's Consequences**: The consequence of an action will depend on the current cash and the number of units of each stock available to sell. The consequences will be specific for each stock. When the agent chooses a forbidden action for a given stock it holds at that time step for that stock. The elements stored upon buying are specific for each stock. When the agent selects selling for one given stock, the agent will check the stored elements for that stock.

- **Rewards**: Upon buying, the price of the stock at the time will be collected. When selling, the agent will receive a reward equivalent to the difference between the price at which the stock was sold and the price at which it was bought multiplied by the volume of units sold, for each stock sold at that time step. Except for episodes containing only a fixed sinusoid throughout training, the reward of the episode will additionally be divided by the episode's average price of the stock with which it is associated. A penalty will also be added each time the agent chooses a forbidden action.

3.4.1 Results

All episodes in the section are going to have 20 of initial cash and 20 maximum volume for units traded of each stock at each time step. These episode characteristics will be omitted from now on.

3.4.1.1 Results from training sessions with a fixed sinusoid price pattern

Results from training sessions with 2 stocks
The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	80

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.40 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

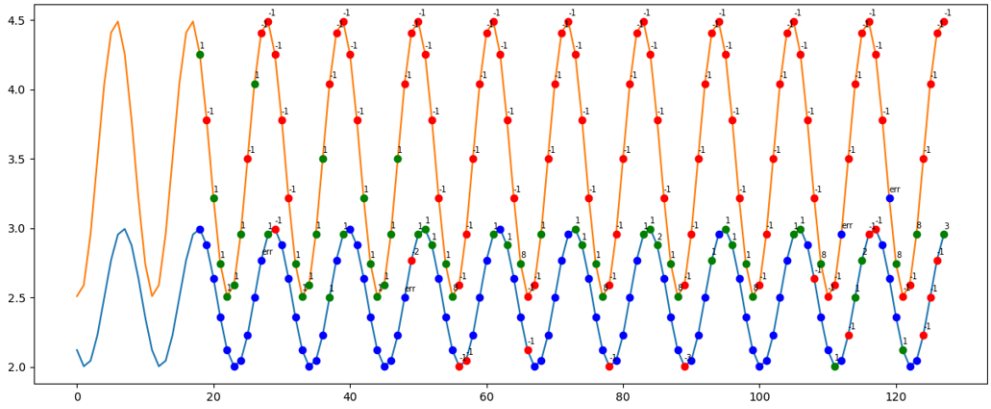
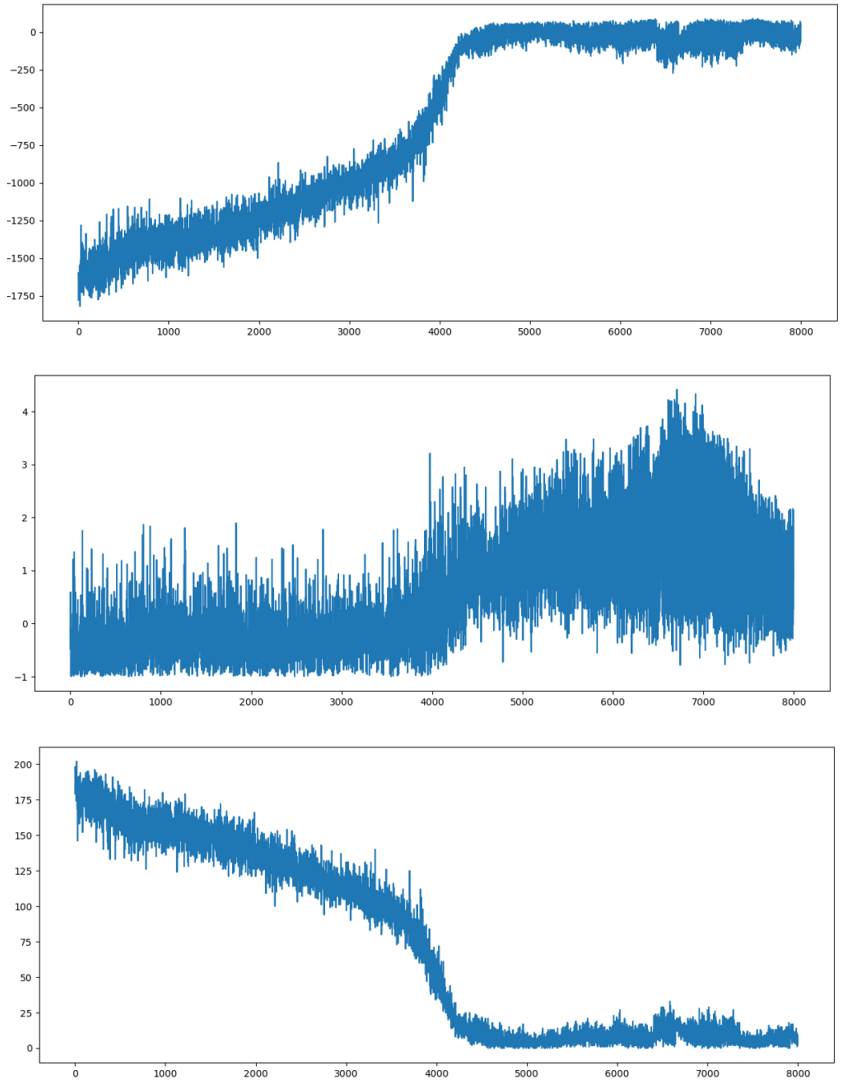


Figure 3.41 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions with 3 stocks

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.0002
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	80

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.42 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

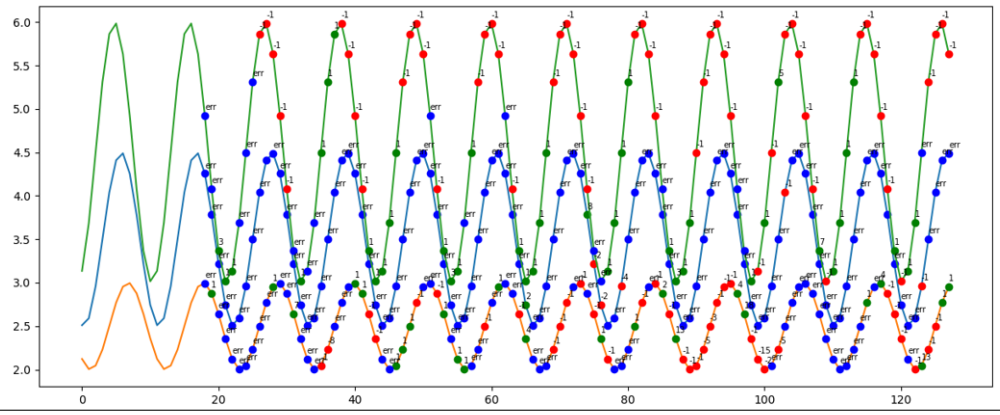
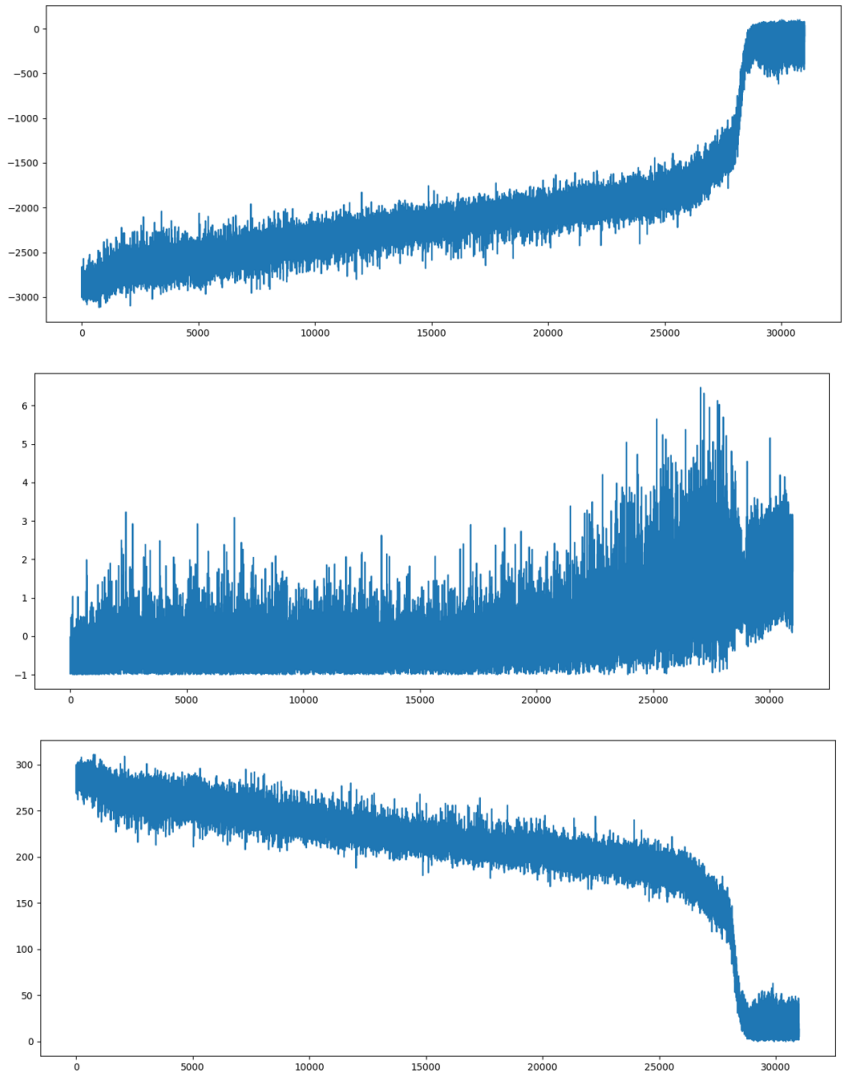


Figure 3.43 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



3.4.1.2 Results from training sessions with a changing sinusoid price pattern

Results from training sessions with 2 stocks

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.004
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	30

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.44 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

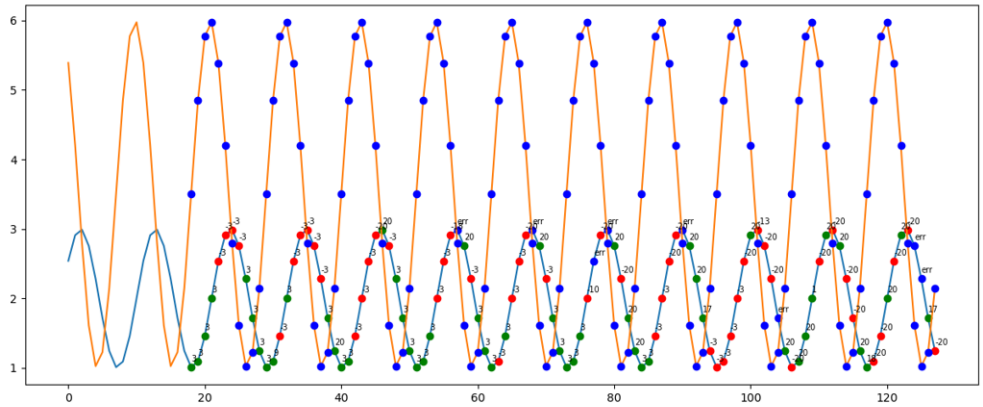
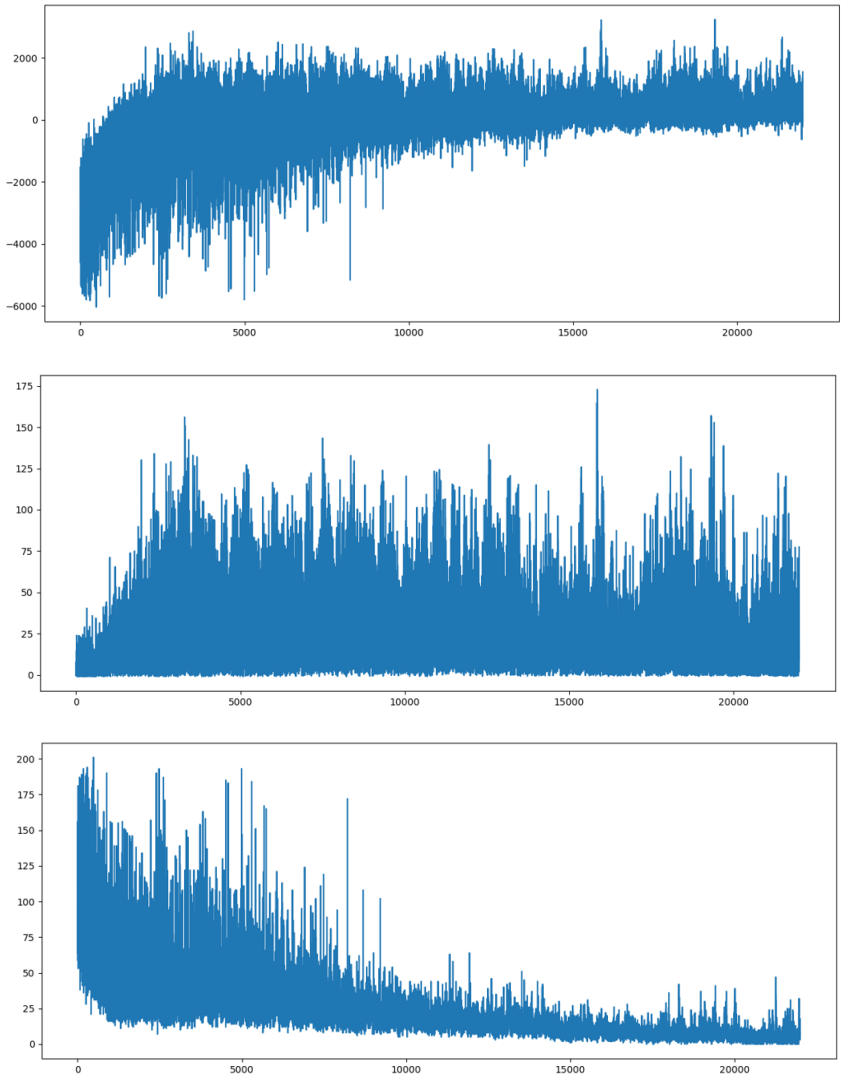


Figure 3.45 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



Results from training sessions with 3 stocks

The tuning obtained that achieved the best performance for this case was:

Window Size	18
Adam's ϵ parameter	1e-4
Learning Rate	0.005
Policy Network Architecture	[128,128,128]
Value Network Architecture	[128,128,128]
Batch Size	128
Penalty	40

The episodic reward, profit, number of errors per episode obtained throughout training, and an example of a test episode obtained with the trained agent are shown in the plots below.

Figure 3.46 Example of an episode completed by the trained agent (the vertical axis represents stock prices, the horizontal axis the timesteps of the episode)

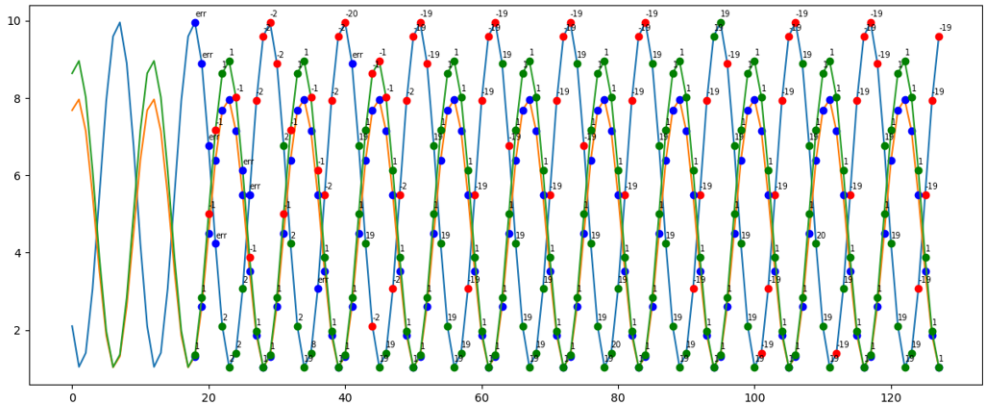
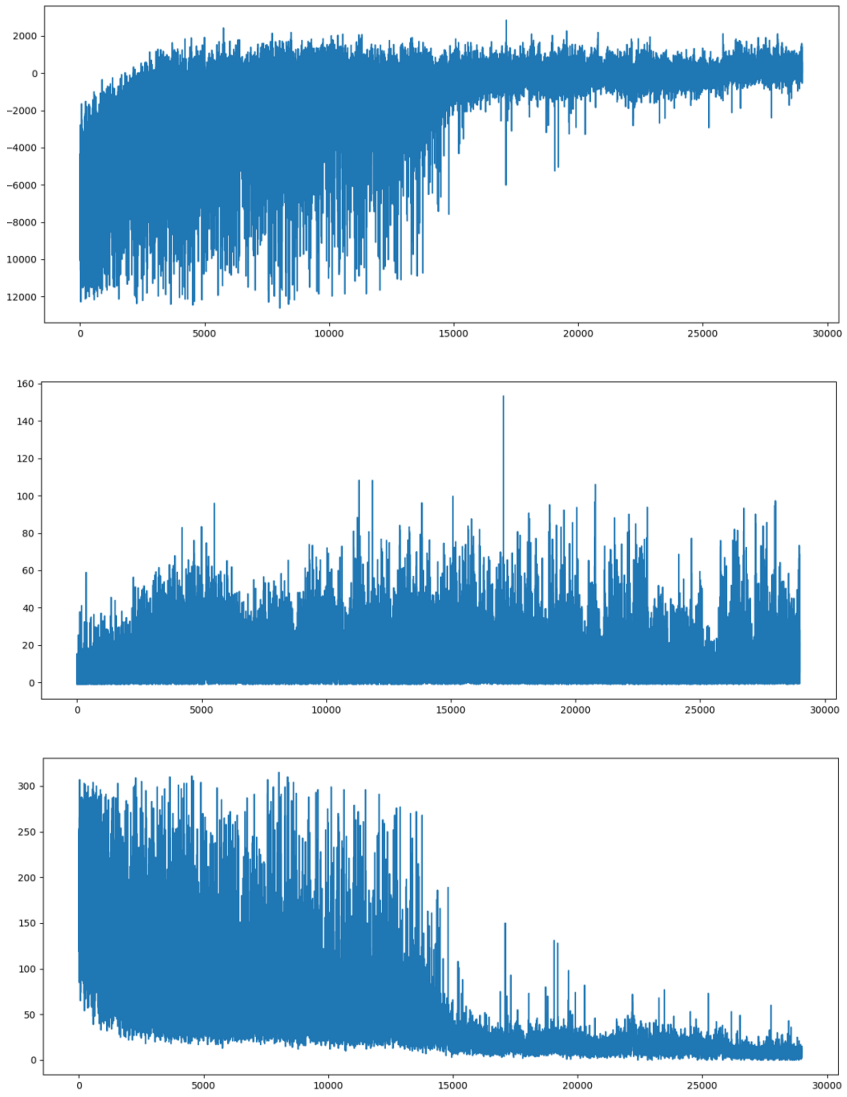


Figure 3.47 The episodic reward, profit, number of error obtained throughout training (the vertical axis represents the episodic reward, the percentage of the extra cash the agent has at the end of each episode given the initial budget, and the number of times the agent chooses forbidden actions per episode respectively, the horizontal axis the number of training episodes completed by the agent)



4

Conclusion

This thesis focused on building simple and effective training environments and training an agent on each of them so that we can better control the inherent complexity of the training process to obtain an agent capable of performing in realistic environments. The lack of availability of resources appropriate for training deep learning models extended the process of finding adequate tunings for each task and training the agents beyond what can be considered practical. Future research could expand the environment evolution to reach environments more analogous to reality and investigate how training in simpler versions helps to achieve successful training in more complex environments. Additionally, the use of hybridization of reinforcement learning techniques with financial models is an interesting potential framework to further simplify agents' interactions with the environment and give an initial trading intuition.

Bibliography

[1] A. Aboussalah, Z. Xu, and C.-G. Lee. "What is the Value of the Cross-Sectional Approach to Deep Reinforcement Learning?" In: *SSRN Electronic Journal* (2020-01). DOI: 10.2139/ssrn.3748130 (cit. on p. 5).

[2] S. Basak and G. Chabakauri. "Dynamic Mean-Variance Asset Allocation". In: *The Review of Financial Studies* 23.7 (2010), pp. 2970–3016 (cit. on p. 4).

[3] F. Black and R. Litterman. "Global portfolio optimization". In: *Financial Analysts Journal* 48.5 (1992), pp. 28–43 (cit. on p. 4).

[4] X. Du, J. Zhai, and K. Lv. "Algorithm trading using Q-learning and recurrent reinforcement learning". In: *Positions* 1 (2016) (cit. on p. 5).

[5] E. F. Fama and K. R. French. "Common risk factors in the returns on stocks and bonds". In: *Journal of Financial Economics* 33.1 (1993), pp. 3–56. DOI: 10.1016/0304-405X(93)90023-5 (cit. on p. 4).

[6] E. F. Fama and K. R. French. "Multifactor explanations of asset pricing anomalies". In: *The Journal of Finance* 51.1 (1996), pp. 55–84. DOI: 10.2307/2329302 (cit. on p. 4).

[7] A. Georgantas, M. Doumpos, and C. Zopounidis. "Robust optimization approaches for portfolio selection: a comparative analysis". In: *Annals of Operations Research* (2021), pp. 1–17. DOI: 10.1007/s10479-021-04177-y (cit. on p. 4).

[8] I. Goodfellow et al. "Generative adversarial nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680 (cit. on p. 5).

[9] M. J. Hausknecht and P. Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *AAAI Fall Symposia*. 2015 (cit. on p. 5).

[10] Y. Lecun et al. "Efficient BackProp". In: (2000-08) (cit. on p. 16).

[11] D. Li and W.-L. Ng. "Optimal Dynamic Portfolio Selection: Multiperiod Mean-Variance Formulation". In: *Mathematical Finance* 10 (2000-07), pp. 387–406. DOI: 10.1111/1467-9965.00100 (cit. on p. 4).

[12] Z. Liang et al. "Adversarial Deep Reinforcement Learning in Portfolio Management". In: *arXiv: Portfolio Management* (2018) (cit. on p. 5).

[13] T. P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015) (cit. on p. 5).

[14] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[15] S. Maillard, T. Roncalli, and J. Teïletche. "The properties of equally weighted risk contribution portfolios". In: *Journal of Portfolio Management* 34.4 (2008), pp. 40–51 (cit. on p. 4).

[16] H. M. Markowitz. "Portfolio selection". In: *Journal of Finance* 7 (1952), pp. 77–91 (cit. on p. 4).

[17] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: (2016-02) (cit. on p. 11).

[18] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013-12) (cit. on p. 8).

[19] J. E. Moody and L. Wu. "Optimization of trading systems and portfolios". In: *Proceedings of the IEEE/IAFE 1997 Computational Intelligence for Financial Engineering (CIFEr)* (1997), pp. 300–307 (cit. on pp. 4, 5).

[20] J. E. Moody et al. "Performance functions and reinforcement learning for trading systems and portfolios". In: *Journal of Forecasting* 17 (1998), pp. 441–470 (cit. on p. 4).

[21] H. Park, M. K. Sim, and D. G. Choi. "An intelligent financial portfolio trading strategy using deep Q-learning". In: *Expert Systems with Applications* 158 (2020), p. 113573. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113573. URL: https://www.sciencedirect.com/science/article/pii/S0957417420303973 (cit. on p. 5).

[22] J. L. Pedersen and G. Peskir. "Optimal mean-variance portfolio selection". In: *Mathematics and Financial Economics* 11.2 (2017), pp. 137–160 (cit. on p. 4).

[23] P. C. Pendharkar and P. Cusatis. "Trading financial indices with reinforcement learning agents". In: *Expert Systems with Applications* 103 (2018), pp. 1–13. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.02.032. URL: https://www.sciencedirect.com/science/article/pii/S0957417418301209 (cit. on p. 5).

[24] G. A. Rummery and M. Niranjan. "On-line Q-learning using connectionist systems". In: *Advances in neural information processing systems*. 1994, pp. 1039–1045 (cit. on p. 5).

[25] J. Schulman et al. "Proximal policy optimization algorithms". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3191–3199 (cit. on pp. 5, 12).

[26] J. Schulman et al. "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by F. Bach and D. Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015-07–09 Jul, pp. 1889–1897. URL: https://proceedings.mlr.press/v37/schulman15.html (cit. on p. 12).

[27] W. F. Sharpe. "Capital asset prices: A theory of market equilibrium under conditions of risk". In: *The Journal of Finance* 19.3 (1964), pp. 425–442 (cit. on p. 4).

[28] R. S. Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3.1 (1988), pp. 9–44 (cit. on p. 5).

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018 (cit. on p. 7).

[30] R. S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (2000), pp. 1057–1063 (cit. on p. 5).

[31] E. Vigna. "On time consistency for mean-variance portfolio selection". In: *Collegio Carlo Alberto Notebook* 476 (2016) (cit. on p. 4).

[32] C. J. Watkins and P. Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292 (cit. on p. 5).

[33] R. J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (1992-05), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (cit. on p. 10).

[34] P. Yu et al. "Model-based Deep Reinforcement Learning for Dynamic Portfolio Optimization". In: *ArXiv* abs/1901.08740 (2019) (cit. on p. 5).

[35] X. Zhou and D. Li. "Continuous-Time Mean-Variance Portfolio Selection: A Stochastic LQ Framework". In: *Applied Mathematics & Optimization* 42.1 (2000), pp. 19–33 (cit. on p. 4).