7CCSMPRJ

Deep Reinforcement Learning for Optimal Portfolio Management

Final Project Report

Author: Arman Mann

Supervisor: Dr. Bart de Keijzer

Student ID: 20009893

August 31, 2021

Abstract

Over the last decade, Deep Reinforcement Learning has seen various breakthroughs and a considerable increase in adoption due to advances in Deep Learning and the general availability of computational power. A particular area where there is rising interest is in its application in financial portfolio management. In this dissertation, we provide a framework and tool for the application of the Deep Q-Network (DQN) and Deep Deterministic Policy Gradient (DDPG) algorithms for dynamic portfolio optimisation. For this, the RL agents are trained to maximise returns by rebalancing the portfolio weights on each trading day for a 4 asset portfolio. We show that this automatically makes the agents risk averse, allowing them to make risk-sensitive trades. The agents also substantially outperform various benchmarks and traditional portfolio selection strategies on metrics such as Cumulative/Annualised returns, Sharpe ratio and Maximum Drawdown, amongst others. We further show that our agents, with the same network weights, are able to generalise well by identifying universal trade signals in the price data for the assets tested. In addition to this, the choice of network architecture and hyperparameters used by us seems to have a high learning capacity and allows for training agents on arbitrary portfolios. Finally, the source code we provide is easily extended and enables tremendous customisation, making it straightforward to improve and build upon our work.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Arman Mann

August 31, 2021

Word count: **14658**

Acknowledgements

I would like to extend a special thank you to Dr. Bart de Keijzer for his unwavering support and encouragement - right from the early days of this project - which provided me with the confidence and the much needed motivation to go through with such a challenging research topic.

In addition, I convey my gratitude to the many researchers, scientists, academics and students whose work, prior to mine, made this dissertation possible.

Finally, I would like to acknowledge my dearest Rebecca for her constant love and understanding throughout a very difficult pandemic year.

Dedicated to my parents, for always standing by me.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Aims and Objectives

The primary objective of this dissertation is to investigate the application of Deep Reinforcement Learning (DRL) for the purpose of portfolio allocation in the financial domain. Our research will focus on two widely applied model-free DRL algorithms, namely the Deep Q-Network (DQN) (Mnih et al., 2013; Mnih et al., 2015) and the Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016). We will also perform a comparative analysis of the results obtained against various benchmarks and traditional portfolio allocation strategies.

Further, by examining the existing research on the use of Reinforcement Learning (RL) for this task and by implementing the above mentioned algorithms, we wish to:

- Develop a deep understanding of the entire domain of Reinforcement Learning, its components, and the differences between various methodologies and concepts such as model-free and model-based, value iteration and policy iteration, discrete and continuous action spaces, etc.
- Gain hands-on experience and understanding of the technical implementations of the DQN and DDPG algorithms, simultaneously becoming adept at using the tools and frameworks for building these DRL agents and defining Reinforcement Learning environments in order to interact with the agents.
- Evaluate the performance of the DQN and DDPG algorithms at the task of portfolio management using well defined metrics and thereby demonstrate their usefulness in trading ap-

plications.

- Perform an analysis of the trained agents to infer their learning capacity and generalisation ability.
- Understand the broader use cases for the application of RL in finance, and the challenges associated with the task of portfolio allocation.

1.2 Problem Statement

Based on the above aims and objectives, our problem statement, formally stated, is:

> *To build Deep Reinforcement Learning agents that dynamically optimise the allocation of wealth towards assets in a portfolio in order to maximise returns for a given level of risk.*

1.3 Motivation

The motivations for attempting to solve such a problem are varied. With the exponential growth in the power of computers, each day there are more problems that, previously unsolvable, may be solved (or at least attempted) by applying Artificial Intelligence (AI). The financial markets, due to the large quantities of data generated, provides a prime example for this, where more and more tasks are being automated and modelled using Machine Learning. Portfolio allocation is another such problem. Though there has been considerable work in this area which involves AI, it is only recently that we have seen an increasing amount of interest in using Deep Reinforcement Learning to trade assets in a portfolio. Owing to this, our problem statement is one which is ripe for further experimentation and exploration in order to take the work done in this field even further.

Personally, and perhaps more importantly, the motivation to move forward with this project arises from a deep interest in finance and Reinforcement Learning including more abstract and higher order goals of learning about and mastering these deeply complex topics. Although, when first we proposed this problem for our dissertation, we had only a surface level understanding of the inner workings of RL, and probably knew even less about portfolio management. Since then we've come a long way and wish to make this dissertation an example of how any challenging problem may be solved with time and dedicated effort.

1.4 Report Structure

This report is subdivided into chapters, each of which tackles a unique area related to the problem of Reinforcement Learning for portfolio optimisation.

In the next chapter we provide a comprehensive background of the theory that enables solving this problem. We take a deep look at the fundamentals of RL and of the algorithms we will be working with, before objectively outlining the task of portfolio management and providing a brief overview of how RL may be used for the same. In addition, we go over numerous metrics, benchmarks and traditional portfolio allocation strategies that enable us to evaluate the agents we train.

In Chapter 3, we survey the state of the art and other historically significant implementations of the DQN and DDPG algorithms for portfolio management. We further compare the design decisions made by each paper's authors to inform our own choices when selecting reward functions, metrics, benchmarks, etc. for this project.

Thereafter, in Chapter 4, we lay out the relevant details of our implementation along with the tools used for the same, which if followed correctly, should allow replicating our results.

Chapter 5 presents our evaluation and comments on the performance of the trading agents built by us. We provide a detailed comparison with traditional strategies used for portfolio management and show our agents can beat all but one. We further test our agents on their generalisation ability and substantiate our choice of neural network architectures and model hyperparameters used by demonstrating their ability to learn to trade arbitrary portfolios.

Finally, in Chapter 6, we conclude the report by summarising our contributions and discuss ways to expand upon our work in the future.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning for experience-driven autonomous learning (Sutton & Barto, 2018). It may also be considered as a mechanism to direct supervised learning methods towards maximizing some long-term or delayed reward (Kolm & Ritter, 2019). Here an agent learns through positive reinforcement by interacting with its environment (or a model of the environment), and performs actions in order to maximise a numerical reward signal over time. (Sutton & Barto, 2018, p. 3). By evaluating the success and failure of these actions the agent learns to make optimal choices for a given problem.

The remainder of this section dives into the key concepts associated with the field of reinforcement learning along with select algorithms important to the research and experiments we conduct in the later chapters. Unless otherwise stated, we make use of Sutton and Barto (2018) for reference and apply the same notation as the book except for a few instances where minor modifications have been made for greater clarity.

2.1.1 Key Concepts

Markov Decision Process

Markov Decision Processes (MDPs) are mathematical frameworks to model sequential decision making problems in stochastic (or deterministic) settings. They provide an abstraction to the problem of goal-directed behaviour, which makes them the perfect tool for formulating reinforcement

learning problems.

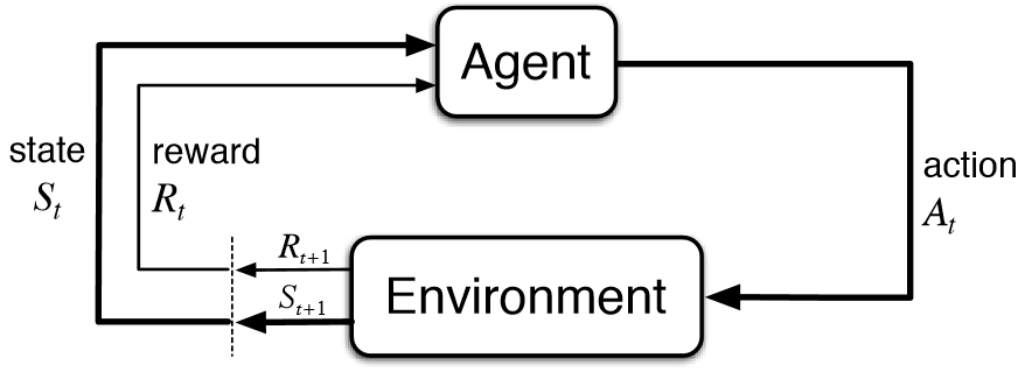

Figure 2.1: Representation of a Markov Decision Process (Sutton & Barto, 2018)

As we shall see, the dynamics of a Markov Decision Process, and thereby of any goal-directed learning, can be completely characterised by the interactions between an agent and its environment through the agent's choices, its goals, and the basis for those choices (Figure 2.1). This is formalised in the form of the probability $p(s', r \mid s, a)$ of observing state $S_{t+1} = s'$ and reward $R_{t+1} = r$ when an action $A_t = a$ is taken in state $S_t = s$. Here t refers to the current time step. It is therefore required that state S_t contains all past information required by the agent to make a decision at the current time. This is known as the Markov property and may be condensed into the phrase *"the future is independent of the past, given the present"* (Feldman & Valdez-Flores, 2010).

Agent

In the field of reinforcement learning, the decision making algorithm which learns by interacting continuously with the environment is known as the agent.

Environment

The environment is everything outside the agent which the agent interacts with but cannot arbitrarily change. It may either be fully or partially observable. The observations received by the agent from the environment allow it to learn to respond to novel situations by selecting actions that maximise reward.

State and Observation

In an MDP, the agent receives a representation of the environment state, S_t, at each time step which forms the basis for the agent to make some decision. This is known as the observation or agent state (or simply the state). In case of a fully observable MDP, the agent state is equivalent to the environment state, and could include anything that may be useful to the agent in making decisions.

Action

An action A_t represents the decision made by the agent which may or may not change the state of the environment and lead to some immediate reward. More broadly it refers to the decisions we want the agent to learn to make.

Reward

Rewards are scalar feedback signals returned from the environment to the agent in response to the action taken at each time step. They indicate the efficacy of the action taken in a given state. The reward at time step t is denoted by R_t.

Return

The goal for reinforcement learning agents, in general, is to maximise the cumulative long-term reward, also known as the return G_t. This is typically represented as the sum of discounted rewards,

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.1}$$

where $\gamma \in (0, 1)$ is known as the *discount rate* or *discount factor* and is used to balance immediate reward with long-term reward. An agent with $\gamma = 0$ is considered myopic, attempting to maximise only immediate rewards. As γ is increased towards 1 it becomes more and more farsighted and places a greater importance on delayed rewards.

Episode

An episode is a single independent subsequence of agent-environment interactions which consists of a definite start and terminal states.

Policy

A policy, π, is a stochastic rule and can be seen as a mapping from agent states to probability distributions of selecting possible actions. It thereby defines an agent's behaviour. The probability of selecting an action a by an agent currently in state s and following policy π is denoted by $\pi(a \mid s)$.

Value Function

Value functions are functions of states (or state-action pairs) that determine the value of a state (or of selecting an action in a state) in terms of the expected return. They are often defined with respect to the policy being followed when acting.

The state value function of policy π in state s, $v_\pi(s)$, is defined as the expected return on starting in state s and following policy π. More formally this can be written as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.2}$$

Similarly, the state-action value (known historically as the Q-value, or simply action value) function of policy π, $Q_\pi(s, a)$, is given as the expected value of taking action a in state s and following π thereafter.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{2.3}$$

The two value functions are related to each other as follows:

$$v_\pi(s) = \sum_a \pi(a \mid s) Q_\pi(s, a) = \mathbb{E}_\pi[Q_\pi(s, a) \mid S_t = s] \tag{2.4}$$

Optimality

For any given RL task, there must exist an *optimal policy*, denoted as π_*, which is better than or equally as good as all other policies, ie. $\pi_* \geq \pi \; \forall \; \pi \neq \pi_*$. A policy π is defined to be better than a policy π' if and only if $v_\pi(s) > v_{\pi'}(s)$ for all states s. All optimal policies achieve the optimal state and action values. As such, the optimal state and action value functions for the optimal policy produce the largest expected return obtainable by any policy, and are given by

$$v_*(s) = v_{\pi_*}(s) = \max_\pi v_\pi(s) \tag{2.5}$$

$$Q_*(s, a) = Q_{\pi_*}(s, a) = \max_\pi Q_\pi(s, a) \tag{2.6}$$

These are related to each other through

$$v_*(s) = \max_a Q_*(s, a) \tag{2.7}$$

or alternatively

$$Q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{2.8}$$

The above optimal value functions denote the best possible performance outcome for the MDP. Therefore, finding the optimal value functions gives us the optimal policy. Here, the optimal state-value gives the value after a *one-step lookahead* by selecting the best action at the next step. Although this looks at the one-step reward, by virtue of being the state value, it encodes information about all future long-term rewards from the state. The optimal action-value, on the other hand, gives the value after taking a predetermined action a and thereafter selecting the best action at each time step. In this case, the optimal policy can be determined simply by selecting a such that it maximises $Q_*(s, a)$. This, once again, is due to the fact the action value for the optimal action incorporates the information from future states.

In view of this, it can be said that the optimal policy is the greedy policy which performs the optimal actions at each time-step determined by the optimal value functions. This allows the evaluation of short term information to yield optimal actions which maximise long-term returns without having to know the values of future states and actions.

The Bellman Equations

One way to solve for the value function equations defined earlier is to use Bellman equations (Bellman, 1957). These were named after their creator, Richard E. Bellman, who in the 1950s, developed an approach solve optimal control problems by defining an equation using system states and value functions. This approach came to be known as Dynamic Programming (DP) and the equations used to solve such problems, the Bellman equations.

The Bellman equations allow us to rewrite the value functions recursively in terms of the future state/action values. This, as we will see, forms the basis of multiple approaches of approximating the state and action value functions in reinforcement learning.

There are four Bellman equations as shown below. Of these, the first two are sometimes known as the *Bellman expectation equations* while the last two, approximating the optimal value functions, are known as the *Bellman optimality equations*.

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big]
\end{aligned} \tag{2.9}
$$

$$
\begin{aligned}
q_\pi(s, a) &= \mathbb{E}[G_t \mid S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\
&= \sum_r \sum_s p(s', r \mid s, a) \Big[r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a') \Big]
\end{aligned} \tag{2.10}
$$

$$
\begin{aligned}
v_*(s) &= \max_a q_{\pi_*}(s, a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_*(s') \Big]
\end{aligned} \tag{2.11}
$$

$$
\begin{aligned}
q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma \max_{a'} q_*(s', a') \Big]
\end{aligned} \tag{2.12}
$$

The four Bellman equations are summarised below for reference.

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
q_\pi(s, a) &= \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \\
q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a]
\end{aligned}
$$

Curse of Dimensionality

The curse of dimensionality was a phrase coined by Bellman (1957) in the context of dynamic programming and refers to the fact that the number of states, and therefore the computational requirements, increase exponentially as the degrees of freedom (the unique state variables) increase. It is also relevant with respect to more recent developments in Reinforcement Learning, and is the primary reason why the Bellman equations cannot be solved for problems with large state spaces. We therefore often resort to function approximation.

Function Approximation

Explicitly solving the Bellman optimality equations would lead to the optimal policy. In practice this is not always possible since the optimal solution is generally associated with extreme computational costs, and therefore is intractable. In addition, the optimal solution is predicated on the Markov assumption and accurately knowing the environment dynamics. However, even though it is modelled as one, the environment may not actually be Markov or the transition probabilities may be unknown.

The field of reinforcement learning, therefore, mainly deals with building agents that *approximately* solve the Bellman optimality equations. In many cases this is done by using actual experienced state transitions as a proxy for knowing the environment dynamics. This brings the added benefit that the agent may now only learn to make good decisions in states that are experienced often and hence deemed important to the problem. Taking suboptimal actions in states encountered with low probabilities in unlikely to affect the cumulative reward and therefore the agent's behaviour. Additionally, the approximate models can now generalise between similar states and therefore reduce the computational burden of learning.

Exploration and Exploitation

As we saw earlier, choosing the greedy action with respect to the action values yields the optimal policy. But for this to be true it is assumed the action values are accurately known. In general, though, we only have estimates of action values. This leads to one of the critical problems in reinforcement learning - balancing exploration and exploitation.

At any time step, in a given state, one of the action value estimates is going to be higher than or equal to the rest. Picking this greedy action implies we are *exploiting* the knowledge present in the action value estimates to maximise the expected return for the current step. Alternatively, due to the uncertainty in the action values, picking a non-greedy action allows the estimate for the selected action to be improved. This is known as *exploration* which may lead to discovering an action that is actually better than the greedy action and thereby provides a greater reward in the long run.

There are various sophisticated means of optimising the amount of exploration and exploitation. For the purpose of this dissertation we consider the ϵ-greedy strategy and action noise. In ϵ-greedy, the agent picks a random action (or explores) with a probability of ϵ at each time step, while in the remaining cases - with probability $1 - \epsilon$ - the greedy action is exploited. Moreover, as learning progresses and the value estimates converge to their real values, we would like to increasingly exploit the greedy actions. Hence the value of ϵ can be decayed or annealed over time to gradually reduce the amount of exploration done, and this is known as the "Annealing ϵ-greedy mechanism."

In case of continuous action space algorithms which suffer from inefficient exploration, an undirected exploration policy, μ', can be applied by adding noise, $\mathcal{N}$, to the action policy to boost exploration.

$$\mu'(s_t) = \mu(s_t \mid \theta_t^\mu) + \mathcal{N} \tag{2.13}$$

For this, the original DDPG implementation made use of time-correlated noise generated from the Ornstein-Uhlenbeck Process (Uhlenbeck & Ornstein, 1930). The correlated noise is dependent on the previous values and prevents the noise from cancelling out over some time period, such as it would in a completely random process. However, future work since then has found that *"noise drawn from the Ornstein-Uhlenbeck process offered no performance benefits"* (Fujimoto et al., 2018) and has opted to use uncorrelated Gaussian noise.

Taxonomy

A first-level categorisation of reinforcement learning algorithms can be made on the basis of the kind of learners. Here we have three distinct categories.

- **Value Based** agents explicitly model the value function. This implies that the agent internally has an approximate value function which it uses to compute the state or action values. In this case, the policy is not learnt explicitly but is constructed implicitly from the value. Examples of value based algorithms are DQN, C51 (Bellemare et al., 2017) and HER (Andrychowicz et al., 2017).

- **Policy Based** agents directly learn the policy without having to learn the values. Examples of this include A2C/A3C (Mnih et al., 2016) and PPO (Schulman et al., 2017).

- **Actor-Critic** systems are agents which have an explicit representation of the policy and value function, and learn both. Here, the policy is defined by the actor and the value function by the critic. Some examples of actor-critic methods are the DDPG, TD3 (Fujimoto et al., 2018) and SAC (Haarnoja et al., 2018).

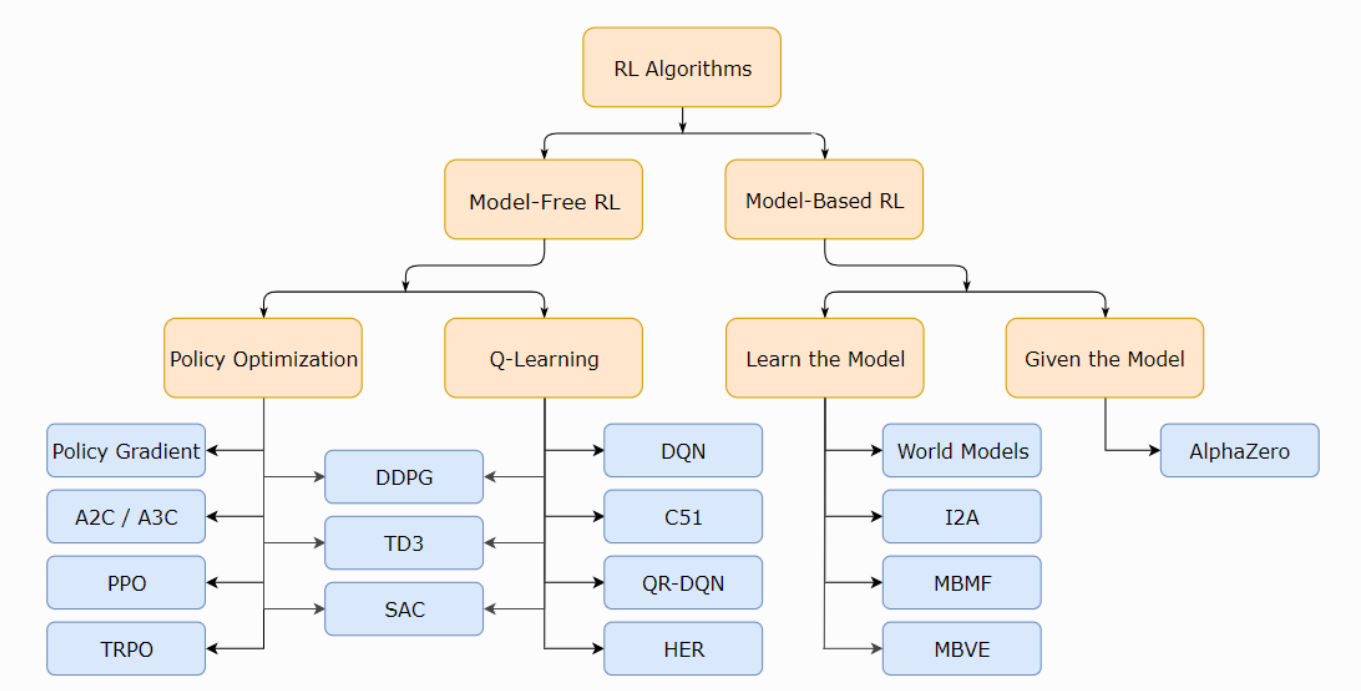

Figure 2.2: Categorisation of RL algorithms
https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

A secondary classification can be made based on whether or not the above learners use a model. A model, in this case, represents a system that can be used to infer the behaviour or the *dynamics* of the environment. It is therefore applied to predict what the environment will do next. Reinforcement learning methods that make use of models to predict future occurrences before actually experiencing them are know as *model-based*. Conversely, methods that use actual experiences for trial and error do not need a model, and are termed *model-free*. This dissertation only deals with the latter. Figure 2.2 illustrates the two levels of categorisation along with a few examples of the RL algorithms in each category.

2.1.2 Q-learning

Q-learning (Watkins & Dayan, 1992) is an off-policy temporal difference (TD) control algorithm widely applied in RL. Opposed to TD prediction which simply estimates state/action value function for a given policy based on experience data, a TD control algorithm is one that finds the optimal policy. In the case of Q-learning this is done by bootstrapping on the future action values to evaluate the greedy policy by picking the optimal action in the next state. The action value of the

current state is then adjusted according to this estimate.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.14}$$

Assuming each state–action pair is visited infinitely often, Q-learning will converge to the optimal action value function, and thereby the optimal policy. This is analogous to solving the fourth Bellman equation.

Since it is an off-policy algorithm, it means we can learn about some target policy, in this case the optimal policy, while following a different behaviour policy, for example an exploratory policy. So the agent may perform exploration to update the state-action values for all state-action pairs but eventually acts greedily according to the behaviour policy. Alternatively, an on-policy algorithm is one where the policy being learnt must be the behaviour policy.

Algorithm 1 Q-learning (Sutton & Barto, 2018)

Input: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialise $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}$, arbitrarily except that $Q(terminal, \cdot) = 0$
for episode $= 1 \dots M$ **do**
  Initialise S
  for $t = 1 \dots T$ **do**
    Choose A from S using policy derived from Q (eg. ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  end for
end for

2.1.3 Policy Gradient

Policy Gradient methods are a class of algorithms part of the policy based learning methods in reinforcement learning which explicitly learn a parameterised policy instead of approximating a value function to determine behaviour. Advantages of this are that policy based methods can easily be extended into high dimensional or continuous action spaces and directly serve the goal of finding "how to act" without needing to bootstrap off of a value function. Moreover, approximating the policy may often be simpler, and therefore faster, than approximating the value function.

The goal here, given some policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ with parameters $\boldsymbol{\theta}$, is to find the best $\boldsymbol{\theta}$. This is usually

done by performing gradient ascent to maximise a chosen performance measure, $J(\boldsymbol{\theta})$.

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t) \tag{2.15}$$

In the episodic case, that is where there are distinct terminal states, $J(\boldsymbol{\theta})$ can be chosen as the state value of a (non-random) start state s_0 under the parameterised policy.

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{2.16}$$

In the non-terminating or continuous case, this can be replaced with the average reward rate.

$$J(\boldsymbol{\theta}) = \sum_s \mu(s) \sum_a \pi_{\boldsymbol{\theta}}(a \mid s) \sum_{s',r} p(s', r \mid s, a) r \tag{2.17}$$

where $\mu(s) = p(S_t = s \mid \pi_{\boldsymbol{\theta}})$ is the probability of being in state s in the long run under policy $\pi_{\boldsymbol{\theta}}$.

Determining the effect of changing the policy on action selection in a particular state is relatively easy. However, as we can see, performance also depends on the state distribution μ in which those actions are selected and calculating the impact of changing the policy parameters on the state distribution is quite challenging as it is dependent on the environment. Fortunately, we can make use of the *policy gradient theorem* which, for any differentiable policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ and any performance measure $J(\boldsymbol{\theta})$, provides the gradient for performance in terms of the policy - without requiring the state distribution.

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[Q_{\pi_{\boldsymbol{\theta}}}(S_t, A_t) \frac{\nabla \pi_{\boldsymbol{\theta}}(A_t \mid S_t)}{\pi_{\boldsymbol{\theta}}(A_t \mid S_t)} \right] \tag{2.18}$$

Additionally, since this gives an expectation, it is implied that the term on the right-hand side can be sampled at each time step and therefore used for stochastic gradient ascent in Equation 2.15 to provide an estimate proportional to $\nabla J(\boldsymbol{\theta})$.

2.1.4 Deep Reinforcement Learning

In recent years, rapid advances in the field of Deep Learning have allowed Reinforcement Learning to exploit the ability of Deep Neural Networks (DNNs) to solve dynamic stochastic control problems that have high-dimensional state and action spaces. This has led to the establishment of the field of Deep Reinforcement Learning which has seen great successes in areas such as self-driving, robotics, and gaming.

The key idea here is to use Neural Networks as function approximators to find approximate value functions for the value based methods or, using gradient ascent, find the best policy parameters in the case of policy based methods. As mentioned earlier, the two model-free algorithms we will

study are the Deep Q-Network (DQN), which is built on the foundations of Q-learning, and the Deep Deterministic Policy Gradient (DDPG), which is an actor-critic method and applies concepts from both Q-learning and policy gradients.

Deep Q-Network

The standard DQN is an off-policy, value based algorithm, which uses Deep Neural Networks (typically a Convolutional Neural Network (CNN)) to approximate the optimal action value function using stochastic gradient descent (Mnih et al., 2015). This provides a massive advantage over traditional Q-learning which can quickly become intractable when dealing with problems having large state or action space due to the "curse of dimensionality".

Algorithm 2 Deep Q-learning with Experience Replay (Mnih et al., 2013)

Initialise replay memory $\mathcal{D}$ with capacity N
Initialise action-value function Q with random weights
for episode $= 1 \dots M$ **do**
  Initialise sequence $s_1 = \{x\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  (Instead of histories of arbitrary length, Q-learning works with fixed length representations of histories produced by function ϕ)
  for $t = 1 \dots T$ **do**
    With probability ϵ select a random action a_t
    Otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_{t+1} and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_{t+1}, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_{j+1}, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_{j+1} & \text{for terminal } \phi_{j+1} \\ r_{j+1} + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non terminal } \phi_{j+1} \end{cases}$
    Perform gradient descent step for minibatch on sample loss $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

However, on using a non-linear function approximators such as Neural Network to represent the action value, learning is found to become unstable and even diverge in some cases (Tsitsiklis & Van Roy, 1997). This occurs due to the correlations present in consecutive observations in case of sequential data and the fact that, due to the use of neural networks, small changes to the value function can significantly alter the policy.

To counteract this, Mnih et al. (2015) propose two key ideas. First, the DQN uses Experience Replay (Lin, 1992), wherein the weights of the network are updated using randomly selected mini-batches of past experiences ($S_t, A_t, R_{t+1}, S_{t+1}$ quadruples) from a "replay buffer". This eliminates the correlations between consecutive samples leading to better convergence as well as allowing the agent to train on less data. Secondly, a separate target network is proposed which is updated only periodically to further increase stability while training. Learning occurs through an ϵ-greedy strategy where the agent randomly picks an action with a probability of ϵ to balance exploitation with exploration.

Deep Deterministic Policy Gradient

The original DQN can only handle low-dimensional discrete action spaces. While this problem may be alleviated to an extent by discretisation of the action space, this soon causes the number of actions to explode exponentially with the number of variables. The DDPG is an off-policy, model-free, actor-critic algorithm which improves upon the policy based Deterministic Policy Gradient (DPG) by incorporating learnings adopted from the DQN (Lillicrap et al., 2016). It is also considered the continuous action space variant of Deep Q-learning as it is able to learn in high-dimensional, continuous action spaces. It thereby offers a solution to the problem of discrete action spaces faced by DQNs.

The DPG algorithm on which the DDPG is based is a policy gradient method. It too is trained off-policy through a distinct actor-critic mechanism, learning deterministic target policy while following an exploratory stochastic behaviour policy (Silver et al., 2014). Here, the actor maintains a deterministic mapping between states and actions using feedback from the critic, while the critic learns the Q-function using off-policy data. The deterministic policy is calculated as the expectation of the gradient of the action-value function, allowing it to be sampled and thereby estimated efficiently. At the same time the stochastic policy provides adequate exploration while learning.

Algorithm 3 Deep Deterministic Policy Gradient (Lillicrap et al., 2016)

Randomly initialise critic network $Q(s, a \mid \theta^Q)$ and actor $\mu(s \mid \theta^\mu)$ and weights θ^Q and θ^μ.
Initialise target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer R
for episode $= 1 \dots M$ **do**
  Initialise a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1 \dots T$ **do**
    Select action $a_t = \mu(s_t \mid \theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_{t+1} and observe new state s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in R
    Sample a random minibatch of N transitions $(s_i, a_i, r_{i+1}, s_{i+1})$ from R
    Set $y_i = r_{i+1} + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$
    Update critic by minimising the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a \mid \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu)|_{s_i}$$

    Update the target networks (Polyak update):

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau)\theta^{\mu'}$$

  end for
end for

The DDPG makes modifications to the DPG by including some of the innovations pioneered by the DQN. Similar to the DQN, DDPG introduces Experience Replay and separate target networks for the actor and critic to stabilise learning of the policy and Q-function. Furthermore, instead of periodically updating the target networks with a copy of the weights like in the DQN, the DDPG performs "soft updates" using the Polyak update coefficient $\tau \ll 1$.

$$\theta' \leftarrow \tau \theta + (1 - \tau)\theta' \tag{2.19}$$

This constrains the target weights to evolve slowly, improving the stability of the network greatly. Together these modifications allow for the use of Deep Neural Networks as function approximators.

In addition, for better exploration, the exploration policy is constructed by adding action noise $\mathcal{N}$ (Equation 2.13).

2.2 Reinforcement Learning in Finance

Similar to other fields, there is a giant body of work which looks to exploit the decision making ability of Reinforcement Learning algorithms in financial domains to make investment and trade decisions. These include decisions related to the pricing of assets, buying and selling of securities, automated or optimal order execution, dynamic balancing and allocation of portfolios, hedging risks and liability management, etc. This has allowed the field to move forward from the traditional (and more popular) Supervised Learning techniques that have found widespread applications over the last few decades (Fischer, 2018) to experimenting with the latest Reinforcement Learning techniques on tasks that attempt to maximise payoffs at some point in the future. Typically this is done by formulating the given task into a multi-period optimal control problem which can be modelled as an MDP and thereby solved with Reinforcement Learning. Thereafter, the agent interacts sequentially with the environment to determine the best action at each time-step which would potentially lead to maximising profits over the selected period of time. The inclination to use Reinforcement Learning to do so seems obvious given how it may be thought of as the most natural way of learning, with clear associations to the manner in which humans and animals learn.

2.3 Portfolio Management

This dissertation deals with problem of *optimal portfolio management*. This is defined as the dynamic and inter-temporal process of determining optimal weights for a portfolio of multiple assets in order to maximise expected returns for a given level of risk (Sato, 2019), or conversely to minimise the risk associated with a desired level of expected return. Here, weights refer to the proportion of wealth allocated to each asset in the portfolio. Not only is this a challenging undertaking in itself, it can be further complicated in case the portfolio weights are to be revised at each time step, and especially in case transaction costs, or commission fees, are to be considered for executing each trade (X. Li et al., 2019).

Several frameworks in economics exist which formalise the problem of portfolio management into strategies that define how to select assets for a portfolio and how wealth should be allocated amongst them. We will be looking closely at two such frameworks.

2.3.1 Modern Portfolio Theory

Modern Portfolio Theory (MPT), was proposed by Markowitz (1952) as a mathematical framework to assemble portfolios that maximise *expected return* for a given level of risk. Alternatively, it can also be used to construct portfolios to minimise risk for a given level of expected return. Such a portfolio is said to be *efficient*.

MPT assumes that an investor prefers a less risky portfolio to a riskier one for the same rate of return. Using variance and asset correlation to lay the statistical grounding for the framework, it argues that an individual instrument's risk and return should not be considered independently, but evaluated by how the investment impacts the overall risk and return of the portfolio. A key insight of MPT is the idea of diversification - that an investor should invest in multiple asset classes that are uncorrelated to reduce the portfolio's variance.

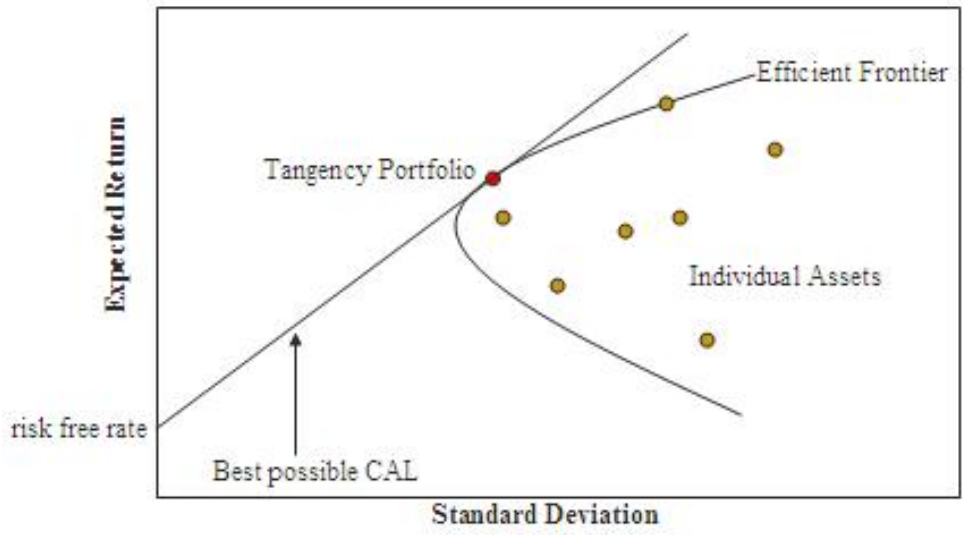

Figure 2.3: Efficient Frontier
https://en.wikipedia.org/wiki/Efficient_frontier

Markowitz Efficient Frontier

The efficiency of a portfolio can be measured using a graph which plots the portfolio's risk against the expected return. On this graph, an upward sloping curve can be used to connect the set of optimal portfolios. This curve, depicted in Figure 2.3, is known as the efficient frontier, and forms a cornerstone of Modern Portfolio Theory. Usually, the portfolio standard deviation is used as a proxy for risk. The straight line through the graph denotes the best possible Capital Allocation Line (CAL), which determines the optimal allotment of funds between risk-free and risky assets.

The intercept point of the CAL and the efficient frontier gives the most efficient portfolio, known as the *tangency portfolio*.

A portfolio that lies underneath the curve is undesirable because it does not provide a sufficient return for the level of risk. Similarly a portfolio to the right of the efficient fronter will no longer be efficient either as it takes on a higher risk for any given return.

2.3.2 Kelly Capital Growth Criterion

The Kelly Criterion (Kelly, 1956), also known as the growth-optimal investment strategy or Capital Growth Theory (Hakansson & Ziemba, 1995), deals with directly maximising the long-term growth rate of a portfolio or trading strategy. In contrast to the Modern Portfolio Theory, investments based on the Kelly Criterion are aimed at growing capital rather than managing risk. This is done by carefully selecting the optimal value for the amount of wealth put into each trade position, k, based on the strategy's past performance.

$$k = P - \frac{1 - P}{R} \tag{2.20}$$

Here P refers to the historical win percentage for the strategy considered and R is the historical win/loss ratio. In theory, trading with the Kelly Criterion also minimises the expected time taken to reach a specific wealth target, however since it takes no consideration of the risk associated with trades, it often results in over-concentrated and therefore non-diversified portfolios which can incur considerable losses in the short-term (Sato, 2019). Along with MPT, this makes up the two major schools of portfolio management. A key difference between the two is that while MPT focuses on single-period asset selection, the Kelly Criterion involves multi-period or sequential portfolio selection (B. Li & Hoi, 2014). This naturally allows the Kelly Criterion to lend itself as a foundational basis to the several online portfolio selection strategies we will discuss in Section 2.3.5 and use thereafter as baselines.

2.3.3 Reinforcement Learning for Portfolio Management

The process of sequentially optimizing portfolio weights to maximise returns for a given level of risk may be represented as a Markov Decision Process (MDP), making portfolio optimization a stochastic optimal control problem which can be solved using the Bellman Optimality Equation. In case there exists a perfect model, along with state-transition probabilities and a reward function which can represent the environment accurately as an MDP, this may be solved with dynamic programming. However, in portfolio optimization, a model of the environment is not available and the underlying dynamics of the system are not known. As such, we may apply model-free Reinforcement Learning to find an approximate solution to the Bellman equation and thereby

compute the optimal policy entirely from the available sample data. The RL policy, in this case, directly outputs the portfolio weights, integrating, and thereby circumventing, the two step iterative cycle of asset price forecasting and portfolio weight allocation from traditional portfolio management methods. This enables accounting for various constraints such as transaction costs and investor risk aversion entirely by modifying the reward function to incorporate the required signals (Fischer, 2018).

2.3.4 Performance Metrics

When creating a trading model, there is an imperative for a measure of performance. These *metrics*, in addition to providing a means of analysing the model, also present a standard to compare multiple models against each other. In the world of finance, there are many such metrics ranging from the simple to the intricate and complex. Here we pick four which are relatively straightforward as well as widely applied in various financial contexts.

- Cumulative Return (CR): The cumulative return of a portfolio is the total change - gain or loss - in the amount invested over a certain time period. Measured as a percentage it is calculated as follows,

$$CR = \frac{P_{current} - P_{invested}}{P_{invested}} \tag{2.21}$$

  where $P_{current}$ and $P_{invested}$ are the current and originally invested values for the portfolio respectively.

- Annualised Return (AR): The annualised return is the compounded average change in value of the investment each year for a given period of time. When the investment horizon is greater than a year it is also known as the Compounded Annual Growth Rate (CAGR).

$$AR = (1 + CR)^{365/\text{Days Held}} - 1 \tag{2.22}$$

- Sharpe Ratio (SR) (Sharpe, 1994): The Sharpe ratio is defined as the average return on an investment over the risk-free rate of return - per unit of measured risk. As such, it is used to compare the return on an investment associated specifically with the additional risk taken. The risk-free rate here is approximated with any investment with zero or close to zero risk, eg. US Treasury Bills, while the standard deviation of returns is used to infer the risk associated

with the investment.

$$SR = \frac{R_p - R_f}{\sigma_p} \tag{2.23}$$

Here R_p and R_f denote the portfolio return and risk-free rate respectively, whereas σ_p is the standard deviation of portfolio's excess returns.

Please note, in our work we assume a risk free rate of 0%. This is okay because the Sharpe ratios for all strategies shall be evaluated with the same risk-free rate.

- Maximum Drawdown (MDD) (Magdon-Ismail et al., 2004): Although the Sharpe ratio considers investment risk, it treats both, upward and downward swings in prices the same. But clearly, upward movements are more beneficial in terms of accrued wealth. In order to measure the downside risk-resistance of a portfolio, we use Maximum Drawdown. MDD is defined as the maximum peak to trough decline experienced by the investment over a certain time period, before a new peak is seen.

$$MDD = \frac{\text{Peak Value} - \text{Trough Value}}{\text{Peak Value}} \tag{2.24}$$

2.3.5 Online Portfolio Selection Strategies

In order to evaluate the efficacy of the models we train and their potential for real-world deployment, we compare their performance with 4 benchmarks and 9 baselines portfolio management strategies. The benchmarks we have selected are simple portfolio allocation strategies commonly used in the industry.

- Buy and Hold (BAH): The simplest and perhaps the most common trading strategy is the Buy and Hold strategy. In BAH, the initial wealth is invested equally in each asset in the portfolio after which the portfolio is held with no rebalancing or trading until the end of the time period considered.

- Uniform Constant Rebalanced Portfolio (UCRP) (Cover, 1991): In BAH the portfolio weights following the initial investment change with market movements. The UCRP is a type of Constant Rebalanced Portfolio (CRP) strategy which continuously rebalances the changing asset weights to be the same each period. That is, if there are m assets in the portfolio, the asset weights on each time period will be rebalanced to $\boldsymbol{b} = \left(\frac{1}{m}, ..., \frac{1}{m}\right)$.

- Best Constant Rebalanced Portfolio (BCRP) (Cover, 1991): The BCRP too is a special case of CRP which selects the optimal weights for a portfolio that maximise total returns. The

asset weights in BCRP are kept constant in time, however, it is calculated in hindsight, ie. it can't actually be traded and is used as a target to benchmark other trading strategies.

- Minimum Variance Portfolio: The minimum variance portfolio follows from the tenets of the Modern Portfolio Theory presented earlier in Section 2.3.1. As the name suggests, it is the leftmost portfolio lying on the efficient frontier with the minimum associated risk (or variance) not allowing short-selling. The level of risk is achieved through diversification where assets may individually be volatile but, due to being uncorrelated, provide stability.

For the baselines, we select a set of traditional portfolio management algorithms. These may be categorised as one of "Follow-the-Winner", "Follow-the-Loser", "Pattern Matching", or "Meta-Learning" (B. Li & Hoi, 2014). A brief explanation of each type of method and the specific algorithms from that category - which will be used to evaluate the agents we train - is provided below.

Follow-the-Winner

Follow-the-winner methods attempt to achieve the same returns as an optimal strategy evaluated over historical assets prices. This is done by increasing the weights of the best performing assets in the portfolio. Often the optimal strategy tracked is the BCRP with the aim of achieving performance that approaches that of BCRP.

- Universal Portfolios (UP) (Cover, 1991)
- Exponential Gradient (EG) (Helmbold et al., 1996)
- Online Newton Step (ONS) (Agarwal et al., 2006)

Follow-the-Loser

These methods are based on the understanding that over or under-performing assets will eventually return to some kind of equilibrium. Therefore, when trading with one of these methods, wealth is transferred from over-performing assets to those under-performing over a predefined time period. As such, each strategy in this category is some kind of mean-reverting strategy. Even though following the losers may seem counter-intuitive, in practice, these strategies often perform much better than winner followers.

- Anti Correlation (Anticor) (Borodin et al., 2004)
- Passive Aggressive Mean Reversion (PAMR) (B. Li et al., 2012)

- Online Moving Average Reversion (OLMAR) (B. Li & Hoi, 2012)
- Robust Mean Reversion (RMR) (D. Huang et al., 2016)
- Confidence Weighted Mean Reversion (CWMR) (B. Li, Hoi, Zhao, et al., 2011)
- Weighted Moving Average Mean Reversion (WMAMR) (L. Gao & Zhang, 2013)

Pattern Matching

While the first two categories make use of prior models, Pattern Matching algorithms typically include non-parametric sequential investment strategies which employ both winners and losers, and attempt to predict future distributions of the market based on samples of similar historical patterns in the data.

- Non-parametric Nearest Neighbour Log-optimal Strategy (BNN) (Györfi et al., 2008)
- Correlation-driven Non-parametric Learning Strategy (CORN) (B. Li, Hoi, & Gopalkrishnan, 2011)

Meta-Learning

A final class of online portfolio selection algorithms, known as Meta-Learning methods, resembles expert learning where a number of strategies from multiple classes may be combined to predict the final portfolio value for the next trading period. Although there are various Meta-Learning methods applied to portfolio selection (B. Li & Hoi, 2014), we do not make use of these for the purpose of this dissertation.

Chapter 3

Literature Review

Recent years have seen an increasing amount of research which deals with the application of Reinforcement Learning to portfolio management. This, in itself, seems to be in the early days of becoming a major subfield of RL as well as of finance. In this chapter we look at the most relevant work in the setting of portfolio management and optimization. Since our study will be analysing the DQN and DDPG algorithms, we have decided to primarily showcase prior state of the art work making use of these algorithms. However, we also include certain research which we found to be fundamental to the field or the problem statement in general. For consistency while reading, we begin with analysing literature related to the DQN algorithm and then move to the DDPG.

One of the earliest studies conducted to compare value based and policy based Reinforcement Learning for portfolio optimization was undertaken by Du et al. (2009). They applied Q-learning (without a neural network), which is the basis of current state of the art value based DRL methods, and policy based Recurrent Reinforcement Learning (RRL) to optimise a stock portfolio. Compared to Q-learning, the policy based algorithm was found to be more robust for stochastic optimal control and for dealing with noisy data. This was owing to the fact that policy based methods can produce real valued actions and do not suffer from Bellman's "curse of dimensionality" due to discretised state or action spaces.

With the explosion in availability of computation and therefore in Deep Learning applications, Jin and El-Saawy (2016) made the first attempt to apply Deep Q-learning to optimise a model portfolio which consisted of one high-volatility stock and one low-volatility stock, using a multilayer perceptron (MLP) (instead of a CNN) for approximating the Q-value function. Similar to the standard DQN, ϵ-greedy strategy was adopted to balance exploration and exploitation in combination

Author	**Year**	**Algorithms**	**Reward Signals**	**Metrics**	**Benchmarks and baselines**
Du et al.	2009	Q-learning	Interval profit, Sharpe ratio, Derivative Sharpe ratio	Cumulative profit	S&P 500
Jin and El-Saawy	2016	DQN	Volatility penalised additive profit, Sharpe ratio	Avg Sharpe ratio, Avg returns, Avg std dev.	BAH, UCRP
Jiang and Liang	2017	DPG	Avg log returns	Cumulative return, Sharpe ratio, MDD, Std dev.	Best stock, BAH, UCRP, UP, ONS, PAMR
Jiang et al.	2017	EIIE (DDPG)	Avg log returns	Cumulative return, Sharpe ratio, MDD	Best stock, BAH, UCRP, Anticor, OLMAR, PAMR, WMAMR, CWMR, RMR, ONS, UP, EG, B^K, CORN, M0
Liang et al.	2018	DDPG, PPO, PG	Risk-adjusted portfolio value	Avg daily return, Sharpe ratio, MDD	UCRP, Follow-the-Winner, Follow-the-Loser
Xiong et al.	2018	DDPG	Additive profit	Cumulative returns, Annualised returns, Annualised std error, Sharpe ratio	DJIA, Min variance
X. Li et al.	2019	Adaptive DDPG	Daily returns	Cumulative returns, Annualised returns, Annualised std error, Sharpe ratio	Vanilla DDPG, DJIA, Min variance, Mean variance
Lucarelli and Borrotti	2020	DQN, D-DQN, DD-DQN	Sum of nominal net returns, linear combination of Sharpe ratio and net return	Daily returns, Sharpe ratio	BAH, Genetic Algorithm selection
Z. Zhang et al.	2020	DQN, PG, A2C	Volatility scaled additive profit	Annualised returns, Annualised std dev. of returns, Downside deviation, Sharpe ratio, Sortino ratio, MDD, Calmar ratio, % +ve returns, Profit factor	BAH, Time series momentum (Sign(R)), Moving Average Convergence Divergence (MACD)
Z. Gao et al.	2020	DQN, D-DQN, DD-DQN	Additive profit	Accumulative returns, Sharpe ratio, MDD	BAH, UCRP, Anticor, OLMAR, PAMR, CWMR, RMR, ONS, UP, EG
G. Huang et al.	2020	DDPG	Daily log returns	Avg return, Avg log return, Annual Sharpe ratio, Log annual Sharpe ratio, Annual Sortino ratio, Log annual Sortino ratio, MDD	CSI300, BAH, Min variance
H. Zhang et al.	2021	DDPG	Additive profit	Annualised return, Sharpe ratio, MDD	BAH, UCRP, Anticor, OLMAR, PAMR, UP, EG

Table 3.1: Summary of surveyed literature

with Experience Replay. As there was no need for discretisation of state space, the input features consisted of the historical stock prices along with auxiliary non-price features such as outstanding shares and portfolio value. Transaction costs were also considered and the agent's action space was discretised into seven percentages which determined the amount of stocks to be bought or sold. Two kinds of reward functions were compared, and surprisingly, the models trained using a volatility penalised reward consistently obtained the highest average Sharpe ratio in spite of the second reward function being the Sharpe ratio itself. Their models were able to perform on par with and exceed variations of the "do-nothing" (BAH) and the rebalance (UCRP) benchmarks used by the authors.

Closer to the current state of the art implementations, we have Lucarelli and Borrotti (2020), who treat the binary principles of "problem decomposition" and "financial interaction identification" at macro and micro levels as the basis for portfolio optimization. These principles are incorporated into their framework in the form of two kinds of agents: a set of local agents that model individual asset behaviors and a global agent that defines the cumulative reward. The approach is applied to optimise a cryptocurrency portfolio comprising of four assets using the three classical variations of Deep Q-Learning (Deep Q-Network (DQN), Double Deep Q-Network (D-DQN) and Duelling Double Deep Q-Network (DD-DQN)). Compared on daily returns obtained by agents trained on 2 different reward functions, all produce positive results in most test periods. However the DQN model by far achieves the best performance, obtaining the highest rate of daily return and performing better when compared on the Sharpe ratio with equally-weighted (BAH) and genetic algorithm based portfolio selection strategies.

Z. Zhang et al. (2020) design three Long Short-Term Memory (LSTM) based RL models - DQN, Policy Gradient (PG), Advantage Actor-Critic (A2C) - to directly produce trade positions based on market conditions, and thereby bypass the need to make predictions based on price forecasts. This overcomes the difficulty in having to accurately predict price movements or designing models to decide "what and how much" to buy or sell based on the prices. The LSTM networks are employed for their memory capabilities, with the understanding that they will be better able to model time-series data, and therefore price movements, than a CNN, which is most commonly used. The models trade 50 liquid futures contracts where each model is trained separately using data from a different asset class. They also apply volatility scaling to scale trades based on asset volatility, and compare their results on 9 metrics against baselines such as BAH and classical time series momentum strategies. Amongst the models trained, the DQN achieves the best overall performance largely outperforming the baseline strategies.

Similarly, Z. Gao et al. (2020) too propose a DQN framework for portfolio selection in discretised action space where there is no need to predict future market prices. Furthermore, their approach

is specifically designed for multi-asset portfolios. Instead of the traditional Experience Replay with a random replay buffer, they propose using Prioritised Experience Replay (Schaul et al., 2016) which weighs the selection probability of samples in the memory pool proportionally based on the difference between predicted and real Q-values. This allows for more valuable experience and better adaptation to extreme price movements. Compared against 10 baseline portfolio management strategies, the DQN agent outperforms all on Accumulated Rate of Return (ARR), Sharpe ratio and Maximum Drawdown.

In spite of the breakthrough results obtained in the research conducted above, the issue with finite or discretised action space models arising from the limitations in the number of action combinations remains. Due to this drawback, they often converge to local or non-optimal solutions. Therefore, we now look at equally compelling work relating to and preceding the actor-critic DDPG model for portfolio management in continuous action spaces.

Work done by Jiang and Liang (2017) was the "first empirical study" to employ actor-only Reinforcement Learning to manage portfolios of multiple assets. Citing the instability often caused by training separate actor and critic networks, the authors decided to forgo the critic network and make use of a simple Deterministic Policy Gradient model (leaving out the Q-value estimation) to obtain optimal trading strategies for a cryptocurrency portfolio. Without incorporating any priors into the design, their framework makes use of a CNN to accept historical asset prices as inputs and directly outputs trading actions (or portfolio weight vectors), once again eliminating the need to predict asset prices. The proposed framework when compared against 6 baselines was able to outperform all on cumulative returns, other than PAMR.

Jiang et al. (2017) later proposed another model-free Deep Reinforcement Learning trading framework where they apply a "fully exploiting" DDPG-like solution to directly output portfolio weights for optimizing cumulative returns from a portfolio of 11 cryptocurrencies and one risk-free asset. Their model exploits the Ensemble of Identical Independent Evaluators (EIIE) topology which inspects the history of an asset and evaluates its growth potential for the future. Candlestick features (open, high, low, close prices) are used as input. The authors propose Portfolio-vector memory (PVM) which records portfolio weights from the previous trading period for each observation. This works to prevent gradient vanishing problems which plagues some other forms of memory by allowing the network to be trained parallelly against samples in a minibatch. Moreover, PVM is also used to evaluate the effects of transaction costs (due to changing portfolio weights) on the portfolio performance. The authors also propose an Online Stochastic Batch Learning (OSBL) scheme to analyse incoming market information more efficiently. The framework is tested with three different neural networks, a CNN, an RNN (Recurrent Neural Network) and an LSTM. The results show that the designed models easily outperform 3 benchmarks and 12 traditional portfolio management

strategies, reporting *at least* 4-fold returns over 50 day periods.

Liang et al. (2018) adapt 3 different RL algorithms - DDPG, Proximal Policy Optimization (PPO) and PG - to optimise stock portfolios in continuous action-space. The DDPG here too makes use of the EIIE topology described above, however the CNN is replaced by a Deep Residual Network (ResNet; He et al., 2016) to better cope with vanishing and exploding gradients. Portfolios are constructed randomly consisting of Chinese stocks and are periodically optimised after penalizing the agent in case of volatility in the prices for the underlying assets. The study also proposes a novel Adversarial Learning approach which adds random noise to the input features for greater training robustness and risk sensitivity. This improves performance, however it also leads to greater downside risk measured by the Maximum Drawdown. Only results for the PG model are reported as the other two algorithms fail to learn a useful policy. Compared with the UCRP benchmark and back-tested on the Chinese asset market, the PG is seen to perform better.

Xiong et al. (2018) also apply the DDPG algorithm to find an optimal stock trading strategy to maximise investment return. The DDPG is chosen, once again, due to it's ability to handle continuous action spaces in contrast with the DQN algorithm. Results demonstrate that the model is able to make calibrated trades to manage the risk sensitivity of the portfolio. It therefore significantly outperforms the Dow Jones Industrial Average (DJIA) and the Min. variance portfolio strategy on the Sharpe Ratio, in addition to Annualised returns.

X. Li et al. (2019) propose a novel Adaptive Deep Deterministic Reinforcement Learning (Adaptive DDPG) framework for finding the optimal strategy for stock portfolio allocation. The model is an application of optimistic and pessimistic Deep RL which allows dynamic updates to the reward signals by distinguishing between positive and negative environment feedback. As such, the model is able to learn differently based on positive and negative forecasting errors and achieves higher returns when compared to the vanilla DDPG, the DJIA, and other traditional portfolio allocation strategies such as Min. and Mean variance.

G. Huang et al. (2020) present a DDPG framework which enables shorting in continuous action space, allowing them to execute arbitrage trades against the market. They also showcase further optimizations such as redesigning the activation function and the neural network used by the agent. In addition to the candlestick data (open, high, low, close), their model also incorporates 3 non-price features - P/E, P/B and Turnover factors - as well as the CSI300 index price. Working with a portfolio consisting of the CSI300 price to represent the market's rate of return and stocks from the CSI500 index, their design is able to consistently achieve higher returns than the market even when the portfolio is created from assets selected randomly.

More recently, H. Zhang et al. (2021) developed a novel DDPG framework for multi-asset trading in

continuous action space using LSTMs and Fully Connected Neural Networks to model their critic and actors respectively. Here, instead of a simple application of action noise, which is the norm for exploration in continuous action spaces, the author's use the ϵ-greedy strategy to control when exploration using action noise is undertaken. The framework is used to compare performance with and without transaction costs, however arrives at a counter-intuitive result with findings showing the model incorporating transaction costs to be more robust and profitable. This is explained by interpreting the transaction cost as a penalty which trains the agent to be more cautious and thereby choose better actions in order to be profitable. Both proposed strategies, however, outperform the 7 traditional strategies used for comparison on the Compound Annual Return Rate (CARR/CAGR), Maximum Drawdown, and risk-stability measured using the Sharpe ratio.

The literature surveyed in this chapter makes a compelling case for the successful application of Reinforcement Learning at the tasks of portfolio allocation and optimization. It also stands as evidence in order to justify our own undertaking of the same. In the next chapter, we go over our methodology in detail.

Chapter 4

Methodology

4.1 Assumptions

We model the portfolio allocation problem as an MDP, taking into consideration the following assumptions to simplify the environment and trading dynamics.

- Assumption 1: Trades made by the agent cannot have any impact on the market prices.
- Assumption 2: The agent can only trade between the the pre-selected instruments in the portfolio and the risk free asset (cash).
- Assumption 3: Buy and sell orders can be executed at the instrument's closing price on each trading day.
- Assumption 4: The liquidity of the traded instruments is high enough that there is no slippage, and trades can be executed immediately at the price of the order placed.

4.2 Asset Pre-selection

The models are trained on daily price data for assets selected from the Dow Jones Industrial Average (DJIA). For our experiments, we select the top 4 instruments based on market capitalisations (as on July 1, 2021), Apple Inc. (AAPL), Microsoft (MSFT), Visa (V) and JPMorgan Chase (JPM). Although the idea is to build models that can generalise or be retrained on a portfolio comprising of any unique combination of assets, fixing the assets for the trials at the start goes a long way in

reducing the time taken during the hyperparameter tuning stages as it provides a fixed standard to compare model performance against. In the next chapter, however, we will be making use of other assets to present a deeper analysis of the generalisation and learning capacities of our models. Moreover, selecting assets with the maximum market-caps also has the benefit of playing into our assumptions of sufficiently high liquidity (Assumption 4) as these instruments are some of the most traded on the markets and therefore are also less likely to influence market prices (Assumption 1).

Additionally, selecting 4 assets for the portfolio balances the complexity of a multi-stock portfolio with the interpretability of actions taken by the agent. It also works out well in case of the DQN, which requires discretisation of the action space, and would cause the number of discrete actions to explode exponentially as the number of assets are increased. Along with the risk-free asset, cash, this makes the size of the portfolio $m + 1 = 5$.

Dataset partitions	**Start Date**	**End Date**
Training Set	19-03-2008	21-03-2018
Validation Set	22-03-2018	22-11-2019
Test Set	25-11-2019	30-07-2021

Table 4.1: Training, Validation and Test partitions of the data

Price data for the selected instruments is downloaded using the Python *yfinance* library (Aroussi, 2017), which retrieves it from Yahoo Finance (Yahoo, 1997). For our experiments, we use prices within the range of dates starting from 19-03-2008, up to 30-07-2021. The start date here is the date of listing for Visa (V), which was the last company to list on the stock exchange amongst the pre-selected assets in our portfolio, and hence has no earlier data. The dataset is then partitioned into 3 disjoint - training, validation and test - sets, as shows in the Table 4.1. This is also illustrated in the form of a time-series plot in Figure 4.1 which contains each asset's closing prices for each trading day during the selected period. From the figure, we can see that the validation and test sets are significantly different from the training data in terms of their volatility and returns. Choosing such a partition will therefore make for a good test of our model's generalisation ability.

4.3 Environment Design

The trading environment is created using OpenAI Gym (OpenAI, 2016), a Python tool for building reinforcement learning environments as per a standardised interface. The key components of the environment are the action and observation spaces, and the step function. The environment also contains methods to reset it's state and render the actions or outputs received from the agent. These methods are used to revert to the start state and create plots for evaluating model performance after the completion on an episode.

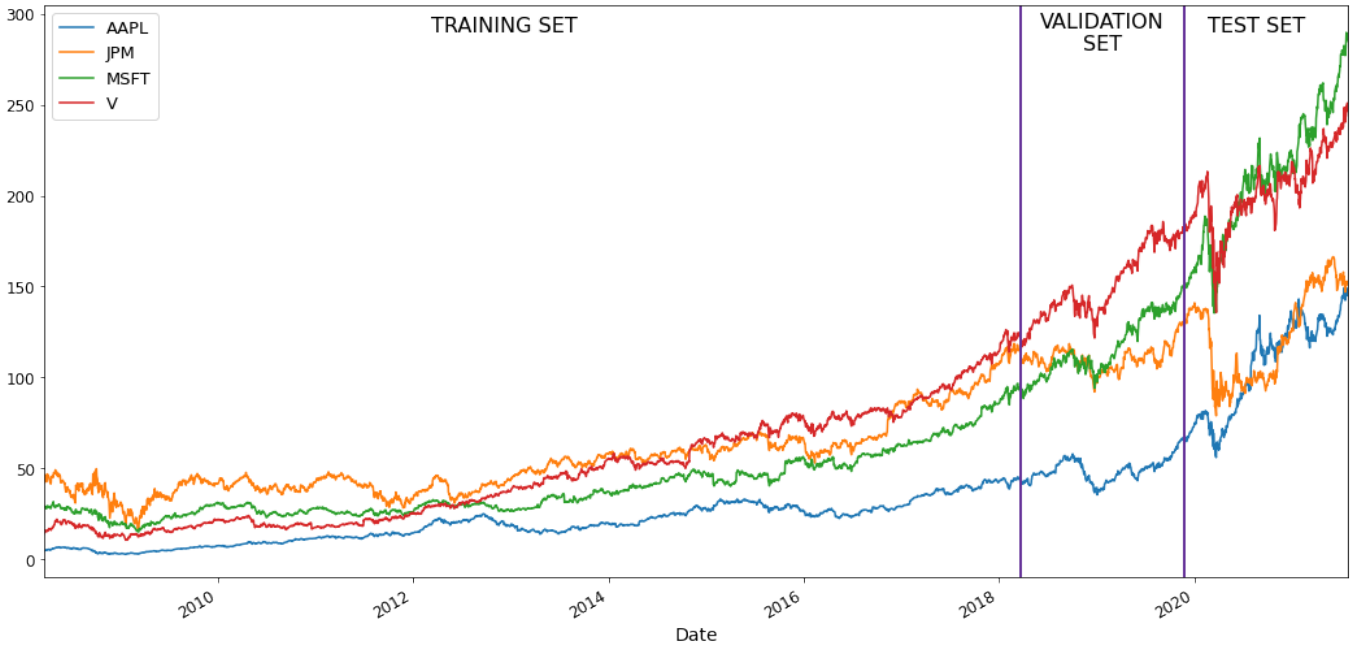

Figure 4.1: Segmentation of data into the three sets

4.3.1 Observation Space

The observation space refers to the coordinate space of the data input received by the agent from the environment. Both, the DQN and the DDPG, support continuous observations, and therefore have identically defined observation spaces. In the current setting, an observation at time t comprises of a three dimensional price tensor $\boldsymbol{X}_t$ with shape (f, m, n), where $f = 4$ is the number of features - namely the Open, High, Low, Close prices for each instrument, $m = 4$ is the number of non-cash assets, and $n = 50$ is the window length for the number of periods before step t which influence the trade decision. It should be noted that even though there may be trace trading signals encapsulated in the raw asset prices, it is more common to normalise the observations before passing them to the agent. Moreover, since only the price changes affect the portfolio performance, the raw prices in the price tensor can be divided by the latest period's (period t) closing price. As such $\boldsymbol{X}_t = \left[\boldsymbol{X}_t^o, \boldsymbol{X}_t^h, \boldsymbol{X}_t^l, \boldsymbol{X}_t^c\right]$, where $\boldsymbol{X}_t^o$, $\boldsymbol{X}_t^h$, $\boldsymbol{X}_t^l$ and $\boldsymbol{X}_t^c$ are the normalised open, high, low, close prices respectively.

$$
\begin{aligned}
\boldsymbol{X}_t^o &= \left[\boldsymbol{v}_{t-n+1}^o \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_{t-n+2}^o \oslash \boldsymbol{v}_t^c \mid ... \mid \boldsymbol{v}_{t-1}^o \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_t^o \oslash \boldsymbol{v}_t^c \right] \\
\boldsymbol{X}_t^h &= \left[\boldsymbol{v}_{t-n+1}^h \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_{t-n+2}^h \oslash \boldsymbol{v}_t^c \mid ... \mid \boldsymbol{v}_{t-1}^h \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_t^h \oslash \boldsymbol{v}_t^c \right] \\
\boldsymbol{X}_t^l &= \left[\boldsymbol{v}_{t-n+1}^l \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_{t-n+2}^l \oslash \boldsymbol{v}_t^c \mid ... \mid \boldsymbol{v}_{t-1}^l \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_t^l \oslash \boldsymbol{v}_t^c \right] \\
\boldsymbol{X}_t^c &= \left[\boldsymbol{v}_{t-n+1}^c \oslash \boldsymbol{v}_t^c \mid \boldsymbol{v}_{t-n+2}^c \oslash \boldsymbol{v}_t^c \mid ... \mid \boldsymbol{v}_{t-1}^c \oslash \boldsymbol{v}_t^c \mid \mathbf{1} \right]
\end{aligned}
\tag{4.1}
$$

In Equation 4.1, vector $\boldsymbol{v}_b^a$ represents the raw asset prices for feature a at time-step b, and the symbol $\oslash$ denotes element-wise division.

In addition to the price vector, the observation received by the RL agent also consists of portfolio weights at the beginning of the previous trade period. This allows the agent to attribute changes in the weights as a factor that determines the transaction costs, and thereby minimise needless rebalancing or trade execution. The weight vector at time t, $\boldsymbol{w}_t$, is therefore determined as a function of the state S_t, which is a tuple containing the price tensor $\boldsymbol{X}_t$ and the portfolio weights at step $t-1$, $\boldsymbol{w}_{t-1}$.

$$
S_t = (\boldsymbol{X}_t, \boldsymbol{w}_{t-1}) \tag{4.2}
$$

where $\boldsymbol{w}_{t-1} = [w_{t-1,0}, w_{t-1,1}, ..., w_{t-1,m}]$

4.3.2 Action Space

When it comes to the action space, the DQN and DDPG algorithms, as we have seen, differ in that the DQN can only output actions belonging to a predefined discrete set, while the DDPG supports continuous actions and may output any real value within a specified range. The actions in question, $\boldsymbol{w}_t$, are simply the weights of the portfolio assets, including the risk-free cash asset. Here on, by "taking an action" we imply that the portfolio is rebalanced such that the assets end up with weights given by the action returned by the agent. We first look at the discretised action space defined for the DQN before moving on the the continuous action space for the DDPG.

To specify the discrete action space in the RL environment, we first need to build the *action set*, which is the complete set of the unique actions an agent may take. To do this, we use the Action Discretisation Algorithm provided by Z. Gao et al. (2020). For reference, this is shown below with minor modifications as Algorithm 4.

The algorithm accepts the number of non-cash assets, M, and the maximum divisions possible, N, and outputs the *action set* in the form of a list. The total number of unique actions is given by the combination $\binom{M+N}{M}$, used in the outer `for` loop. Within this loop we initialise each action with a zero vector, and proceed to iteratively update the values to cover all possible weight combinations. Finally the action vector is normalised to sum up to 1 and added to the actions set. The number of divisions, N, dictates the intervals for the unique proportions of an asset that can be held. This is given by $1/N$ and the holdings of a single asset can only be multiples of this unit. In our case, we have $M = 4$ and selecting $N = 8$, we get a total of 495 actions. This implies that the output layer of the DQN neural network consists of 495 nodes. Since $1/8 = 0.125$, this becomes the smallest unit of the portfolio. The action to be taken by the agent is selected as the *argmax* of the values of the output nodes. This requires that we also maintain a mapping between the action number (corresponding to the output node) and the actual action in the action set. The action corresponding to the node with the maximum value in the output layer is the one that is executed in the environment.

Algorithm 4 Action Discretisation Algorithm (Z. Gao et al., 2020)

Input: Asset number M, number of divisions N

Output: Action set A

$A = [\,]$
$seq = (0, 1, 2, \ldots, M + N - 1)$
for $c \in combinations(seq, M)$ **do**
  $action = (0, 0, ..., 0)$
  for $i = 0 \ldots len(c) - 1$ **do**
    $action[i + 1] = c[i + 1] - c[i] - 1$
  end for
  $action[0] = c[0]$
  $action[M] = M + N - c[M - 1] - 1$
  for $j = 0 \ldots M$ **do**
    $action[j] = action[j]/N$
  end for
  $A+ = [action]$
end for

For the continuous action space in the DDPG, portfolio weights can take any value within a specified range. We set this range to be from 0 to 1, both inclusive, for each asset. Since the agent returns weights between this range for each of the asset but the portfolio weights must sum up to 1, the

weights returned by the agent are normalised to result in the actual portfolio weights.

$$\boldsymbol{w}_t \leftarrow \frac{\boldsymbol{w}_t}{\sum \boldsymbol{w}_t} \tag{4.3}$$

4.3.3 Steps and Episodes

The trading dynamics of the environment are time dependent. Since we use daily price data, time is divided into equal length periods of a single day. Only the days when trading occurs are considered, that is market holidays are completely ignored as no trading actions may be taken on such days. As we saw earlier, the agent receives an observation and selects an action to take. This occurs once for each trading day, and is considered to be one step in the execution of the algorithm.

The trading process for the application is inspired by the methodology popularised by Jiang et al. (2017) in their breakthrough work which we surveyed in Chapter 3. The initial portfolio weights are set to $\boldsymbol{w}_0 = (1, 0, ...0, 0)^\intercal$. On each trading day, the assets trade on the exchange until the market closes, with the prices moving up and down during the interim. In preparation for the next trading day, at each day's end the portfolio weights are reallocated between the assets based on the Open, High, Low and Close prices for the given day as well as on the record of past prices within the time window being considered. Thus, at the end of trading day $t-1$, the portfolio is rebalanced to have weights $\boldsymbol{w}_t$.

The *price relative vector*, $\boldsymbol{y}_t$, for the tth trading day encapsulates the price changes over the course of the trading day since the previous closing. It is given as the element-wise division of $\boldsymbol{X}_t^c$ by $\boldsymbol{X}_{t-1}^c$. The first element in the *price relative vector* represents the risk-free asset. Since the price of cash does not change, at least within the domain of our problem statement as all prices are quoted in cash, this is hardcoded to always be 1.

$$\boldsymbol{y}_t = \boldsymbol{X}_t^c \oslash \boldsymbol{X}_{t-1}^c = \left(1, \frac{x_{1,t}^c}{x_{1,t-1}^c}, \frac{x_{2,t}^c}{x_{2,t-1}^c}, ..., \frac{x_{m,t}^c}{x_{m,t-1}^c}\right)^\intercal \tag{4.4}$$

Over the course of the next trading day, t, the weights evolve due to the price movements in the market from $\boldsymbol{w}_t$ into $\boldsymbol{w}_t'$, and then the process is repeated.

$$\boldsymbol{w}_t' = \frac{\boldsymbol{y}_t \odot \boldsymbol{w}_t}{\boldsymbol{y}_t \cdot \boldsymbol{w}_t} \tag{4.5}$$

Here, $\odot$ denotes element-wise multiplication. Finally, at the end of a certain number of steps, we reach a terminal state which is defined as the last trading day in the dataset. This marks the end of the episode wherein the environment's reset method is called to revert the environment to the

start state and subsequently begin another training episode.

4.3.4 Transaction Costs

Based on the surveyed literature, a standard commission rate, c, of 0.25% is chosen for the purpose of this application. This commission is charged on the value of traded assets, to simulate the brokerage fee charged on executed trade orders. The total commission charged for the rebalancing of weights at the end of time period $t-1$, μ_t, is then given by reallocating the portfolio weights from $\boldsymbol{w}'_{t-1}$ into $\boldsymbol{w}_t$.

$$\mu_t = c \sum_{i=1}^{m} \left| w_{t,i} - w'_{t-1,i} \right| \tag{4.6}$$

This allows us to calculate the portfolio value at the close of trading day t as a function of the previous portfolio value,

$$p_t = p_{t-1} \left(1 - \mu_t\right) \boldsymbol{y}_t \cdot \boldsymbol{w}_t \tag{4.7}$$

4.3.5 Reward Signal

According to the dynamics of an MDP, at the end trading day t, along with a new observation for the next day, S_{t+1}, the agent is also returned a reward signal, R_{t+1}. This is the immediate reward, calculated from taking action A_t in state S_t. We set the reward signal to be the daily return obtained by the agent, denoted by ρ_t. As we shall see in the next section, using daily returns for the reward actually performs better than one incorporating the Sharpe ratio in terms of trading a risk-sensitive portfolio. Similar results were also observed by Jin and El-Saawy (2016) in their experiments with the DQN where the volatility penalised reward obtained a higher Sharpe ratio.

$$R_t = \rho_t = \mu_t \boldsymbol{y}_t \cdot \boldsymbol{w}_t - 1 \tag{4.8}$$

4.4 Model Specifications

The experimental setup for the trading agent makes use of the Stable-Baselines3 framework (Raffin et al., 2019) for building Reinforcement Learning agents in Python. This library makes available standard and reliable implementations of various state of the art RL algorithms which are built in PyTorch (Paszke et al., 2019), and enables fast prototyping alongside providing the ability for endless customisation. At the core of our experiments are the DQN and DDPG implementations provided by Stable-Baselines3. The default neural network architectures for these (Figure 4.2)

involve a feature extraction neural network pipeline which transforms the high-dimensional input observations into feature vectors. This feeds into a Multi-Layer Perceptron (MLP) that learns the Q-values in the case of the DQN, or two MLPs (actor and critic) for the DDPG which learn the policy and Q-values respectively. Of course, in addition to this each network has a separate target network which is updated periodically with the values from the primary network.

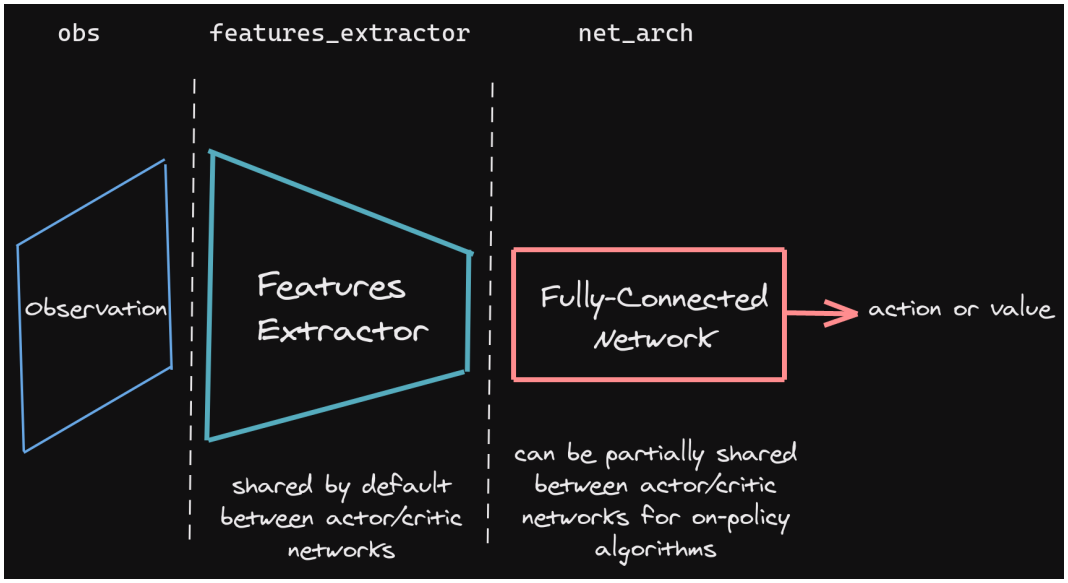

Figure 4.2: Default Architecture of Stable-Baseline3 Algorithms
https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html

By default, the feature extractors too make use of an MLP architecture. However, finding this insufficient for our purpose, we design our own feature extraction pipeline using CNNs. The neural network architecture we define for feature extraction is almost exactly the same for both, the DQN and DDPG algorithms, with minor changes in the input/output sizes of the layers. The details of these are provided in Table 4.2 in the (c, h, w) format where c denotes the number of channels, and h and w refer to the height and width of each channel in the convolutional layers. The first dimension of each of the kernels, which is the filter applied to the channels, is chosen to be 1, making each asset independent from the rest during the convolution operations. Valid padding is applied, which means that the features are not padded with zeros while the stride denotes the number of steps taken between each convolution operation. Rectified Linear Unit (ReLU) functions are used for activations between each layer, before finally flattening out the output features to feed into the fully connected networks.

Moreover, instead of passing the entire observation S_t to the input layer, we first perform feature

extraction on the price tensor $\boldsymbol{X}_t$, and thereafter concatenate the portfolio weights present in the observation ($\boldsymbol{w}_{t-1}$) to the output - as there is no need to run feature extraction on these. In our experiments, this approach shows an improvement over passing the entire observation to the feature extractor.

Layer/Activation	**DQN**		**DDPG**		**Kernel Shape**	**Stride**	**Padding**
	Input	**Output**	**Input**	**Output**			
Conv2D	(4, 4, 50)	(4, 4, 43)	(4, 4, 50)	(8, 4, 43)	(1, 8)	1	valid
ReLU							
Conv2D	(4, 4, 43)	(8, 4, 28)	(8, 4, 43)	(16, 4, 28)	(1, 16)	1	valid
ReLU							
Conv2D	(8, 4, 28)	(16, 4, 1)	(16, 4, 28)	(32, 4, 1)	(1, 28)	1	valid
ReLU							
Flatten	(16, 4, 1)	64	(32, 4, 1)	128			

Table 4.2: Custom Feature Extraction Network

The fully-connected MLP architectures for the agents are also completely customisable, allowing us to specify the number and size of each layer of the network. Yet again we use a similar architecture for the DQN and DDPG, with the DDPG making use of a slightly more complex network (Table 4.3). Both the actor and critic networks for the DDPG have the same architecture. Note that the inputs in the first layer of the fully connected networks now include the portfolio weights from the observation and therefore have a size greater by 5 than the output layer of the feature extractor.

4.4.1 Hyperparameter Tuning

The hyperparameter tuning phase of the project required building and validating more than 200 models over the course of 4 weeks, which led to hyperparameters that were able to perform consistently for the chosen problem. Our findings and notes on the chosen hyperparameters are summarised below. Unless otherwise stated, the points below are applicable to both algorithms.

Environment Parameters and Feature Selection

- Two options for the stock price features were tested. The first included the Open, High, Low and Close prices, while the second used only the Adjusted Close prices for the assets in the

Layer/Activation	DQN		DDPG	
	Input	**Output**	**Input**	**Output**
Linear	$64 + 5 = 69$	256	$128 + 5 = 133$	512
Relu				
Linear	256	128	512	256
Relu				
Linear	128	64	256	128
Relu				
Linear	64	495	128	5

Table 4.3: MLP Network Architectures

portfolio. Here, the 4 feature set was found to consistently perform better for both algorithms.

- Window sizes were varied between 1 and 50. In most cases, the larger window size performed better, and considering that a window of length 50 was also used in some of the literature reviewed, as well as the inevitable increase in computational demand and data requirements on going higher, we too settled for 50.

- When discretising the action space for the DQN, we tested 3 values for the number of divisions - 4, 6 and 8 - and chose the largest as the agent was still able to learn well. Going further to 10 divisions would have required 1001 unique actions compared to the 495 in case of 8 divisions, and we decided against it.

- A variety of reward signals - daily returns, daily returns with added Sharpe ratio, log returns, normalised log returns - were experimented with when training the agents. In general, agents rewarded with the daily returns showed the best performance on the validation set when measured on the cumulative return, and even the Sharpe ratio. This is surprising given that one of the reward functions specifically incorporated the Sharpe ratio. A possible reason for this could be the agent failing to make an association between the Sharpe ratio of the traded portfolio since the start of the episode and the observation - which only included data from the past 50 trading days. Moreover, the Sharpe ratio also depends on the past asset weights at each time step, however, as per our design, observations included weights only from the previous trading day which may have prevented this from being learnt by the agent. Additionally, the logarithmic returns may have underperformed on cumulative return because the logarithmic transformation penalises large wealth growths (Z. Zhang et al., 2020).

- The commission rate was not varied as it corresponds to the brokerage fee and therefore not in our control. We chose a standard rate of 0.25% based on the literature.

Network Parameters

- Various kinds of architectures were tested for feature extraction, ranging from simply flattening the values in the observations, to MLPs and CNNs with varying number of layers and neurons/channels. Based on numerous experiments, the best performing architectures - the ones presented above in Table 4.2 - were chosen.

- For the fully connected network, we iteratively tested performance by varying the size and number of layers starting from the default configurations in the Stable-Baseline3 implementations until the larger, more complex networks showed a decrease in performance from the previous one. The selected architectures are given in Table 4.3.

- L2 (Ridge) regularization to shrink model weights by penalising large values was also tested but performance without regularization was usually found to be better.

Training Parameters

- To see sufficient learning, the learning rates had to be set to values much lower than the defaults for the implemented algorithms. These defaults were those applied by the authors of the DQN and DDPG algorithms, and used as the default parameter values in the Stable-Baselines3 implementations. A considerable number of trials were conducted by only varying the learning rates to determine the best values for the two algorithms. Small changes in the values would often result in completely different performance. Due to the greater size and complexity of the DDPG networks, learning rates were usually 1 - 2 orders of magnitudes lower than the DQN for comparative performance between the two models. The best performance for the chosen network architectures was observed for values between 0.00002 - 0.00005 for the DQN and 0.000001 - 0.000003 for the DDPG.

- Batch sizes of 32, 64, 128 and 256 were tested. In most trials, a batch size of 64 was found to perform best for both algorithms.

- A particular pain point during the initial stages of hyperparameter tuning was finding the best gamma (discount factor) value for agents. It is natural to assume that that the default value of 0.99 would be best for maximising long term performance. The DQN, however, showed a significant performance bump when gamma was set to 0.5 as opposed to values closer to the extremes of 0 or 1, where learning would peak and flatline soon after ϵ was completely

decayed. The DDPG performed best with gamma closer to 1. This might hint at its greater flexibility - due to the continuous action space - to course correct over longer periods of time by making incremental changes in portfolio weights.

- After a certain point, increasing the size of the replay buffer showed no noticeable change in model performance. We used a sufficiently large buffer size of 10000.
- The Adam optimiser was used by default for training in each of the experiments conducted.
- Default values of the Polyak update coefficient, 1 and 0.005, were used for both algorithms as the performance was satisfactory and these are required to be close to 1 for the DQN and close to 0 for the DDPG.
- The train frequency, which determines how often weights are updated was drastically reduced for the DDPG from once per episode to once per 100 steps which greatly improved performance.
- The DQN target update, which decides how often the values are copied over to the target network was reduced from 10000 to 1000 and provided a performance boost.

Exploration Parameters

- For the DQN, we tested multiple epsilon decay schedules from high ($\epsilon = 1$) or medium-high ($\epsilon = 0.8$) exploration at the start of training to a final value of 0.01 or 0.001, decayed over 40 - 60% of the training time. By choosing such an annealing pattern, we were able to carefully control the exploration to be high enough to converge to the correct action value estimates and result in an agent that had learnt well.
- In case of the DDPG, we experimented with noise drawn from both, Gaussian distributions and the Ornstein-Uhlenbeck process. Using agents with no action noise as a benchmark, it was found that adding action noise did provide small benefits when it came to the agents learning ability by reducing the possibility of over-fitting on the data. However, the distinction between the two kinds of noise was often harder to make, with there being random fluctuations in performance on slightly varying hyperparameter values.

The final hyperparameters selected from the best performing models in the experiments are given in Tables 4.4 and 4.5.

Hyperparameter	**DQN**	**DDPG**
Learning Rate, α	0.00005	0.000002
Batch Size	64	64
Discount Factor, γ	0.5	1
Window Length	50	50
Divisions, N	8	-
Commission Rate, c	0.0025	0.0025
Replay Buffer Size	10000	10000
Train Frequency	1	100
Polyak Update Coefficient, τ	1	0.005
Target Update	1000	-

Table 4.4: Values of hyperparameters selected after tuning

Model	**Method**	**Parameter**	**Value**
DQN	Annealing ϵ-greedy Mechanism	Initial ϵ	0.8
		Final ϵ	0.01
		Decay	0.4
DDPG	Normal Action Noise	Mean	0
		Sigma	0.02

Table 4.5: Exploration Parameters

Alongside this dissertation, we make available our entire code base for training, testing and evaluating Reinforcement Learning agents for portfolio optimisation. The code is also available in the form of a GitHub repository, *alphaQ* (Mann, 2021). We hope this allows for easy customisation and contribution, as well as inspires future work that builds upon ours. We also urge those reading to experiment further with alternate values for the hyperparameters to see if greater improvements in performance can be achieved. Alternatively, different RL algorithms could be added and compared with the the ones trained by us. Our code is structured in a manner that makes doing this quite straightforward.

Chapter 5

Results and Evaluation

In this chapter we compare the performance of the trained DQN and DDPG models against the benchmarks and baseline strategies defined in Section 2.3.5. Each of these will be evaluated on the set of metrics provided in Section 2.3.4. To do so, we make use of the Universal Portfolios Python library (Marigold, 2014) which provides implementations for each of the algorithms in a standardised interface. We then perform additional checks to evaluate the agents on their generalisation and learning ability.

5.1 Model Performance and Comparison

To evaluate the models trained, we first run prediction loops for each of the agents over the test environment, instantiated using the test data as created in Section 4.2. This immediately allows a comparison to be made between the performance of our agents and the various benchmarks and baselines presented earlier. Figure 5.1 illustrates the cumulative return over the testing period using a time-series plot.

The DQN and DDPG agents trained by us are seen to overwhelmingly outperform the benchmarks and baseline strategies. The only strategy that beats our RL agents happens to be BCRP. It must be noted here again, however, that the BCRP is not a strategy that can actually be traded as it is calculated in hindsight from the best possible portfolio composition over the trading period. This makes our agents the most profitable amongst the strategies that can actually be applied in the markets.

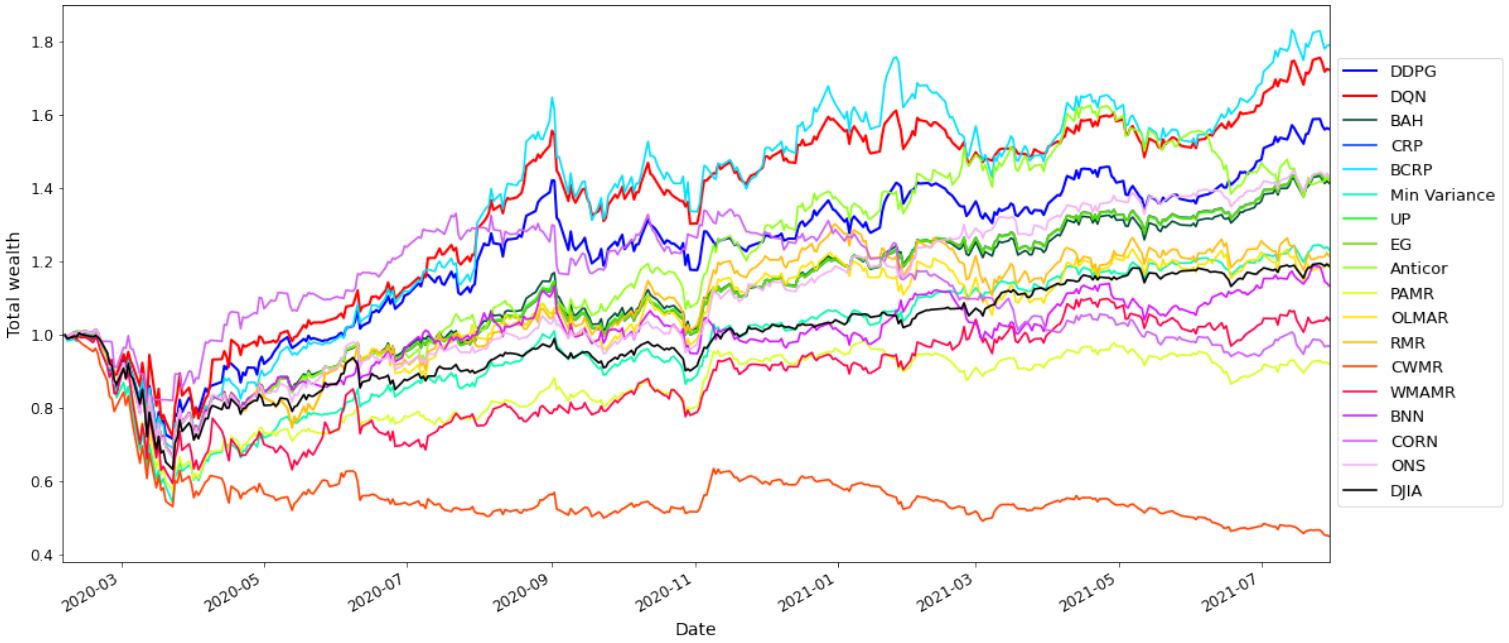

Figure 5.1: Portfolio Performance following different strategies

In Table 5.1, we further compare the traditional portfolio selection strategies with our agents on the Annualised Return, Sharpe Ratio, and Maximum Drawdown - in addition to the Cumulative Return. The table is divided into six sections based on the broad categorisation of the strategy type. The first section "Trained Agents" includes the two agents trained by us. The next two sections contain the benchmarks. Note that the naive benchmarks are so named due to the fact that they don't entirely provide a reasonable comparison. The Min Variance portfolio attempts to minimise risk at all costs and then maximises return for the minimum risk. Whereas, when comparing with the Dow Jones (DJIA), we must consider that the chosen assets themselves outperform the index with the simple Buy and Hold strategy. This makes it all the more likely that actively trading them would further improve performance. The final three sections of the table follow the same grouping as introduced in Section 2.3.5.

The Reinforcement Learning agents are seen to score within the top three positions on each of the chosen metrics. Further, between the DQN and DDPG, the DQN beats the DDPG with a comfortable margin on each evaluated metric and even scores as the top strategy in terms of having the lowest Maximum Drawdown. On the Sharpe ratio it narrowly misses out on the top position to BCRP, with the DDPG not too far behind. This shows that in spite of not using the Sharpe ratio in the reward function during training, the agents learn to trade risk-sensitive portfolios and even

Category	Algorithm	Cumulative Return (%)	Annualised Return (%)	Sharpe ratio	MDD (%)
Trained Agents	**DDPG**	**55.95**	**36.8**	**1.0**	**28.96**
	DQN	**72.15**	**44.37**	**1.13**	**27.92**
Benchmarks	**BAH**	41.2	29.51	0.84	34.23
	UCRP	41.98	29.83	0.85	34.33
	BCRP	**78.96**	**47.89**	**1.16**	31.43
Naive Benchmarks	**Min Variance**	23.03	20.44	0.57	46.47
	DJIA	18.61	16.39	0.53	37.09
Follow-the-Winner	**UP**	47.58	29.64	0.85	34.36
	EG	41.94	29.81	0.85	34.33
	ONS	42.95	30.72	0.85	34.63
Follow-the-Loser	**Anticor**	41.86	30.97	0.81	34.33
	PAMR	−8.19	−0.62	0.02	43.08
	OLMAR	18.31	19.46	0.48	34.33
	RMR	21.28	21.11	0.52	34.33
	CWMR	−55.16	−45.58	−1.1	55.05
	WMAMR	3.82	13.33	0.29	41.12
Pattern Matching	**BNN**	13.18	15.41	0.41	34.33
	CORN	−3.24	1.92	0.07	**29.93**

Table 5.1: Comparison of Models and Trading Strategies
Green, Blue and Orange (in that order) indicate the top three performers on each metric

out perform the traditional strategies. Moreover, it must be noted that the DQN comes close to the best held portfolio's (BCRP) performance even though it has to deal with added commission fees which is not charged in case of BCRP where the portfolio remains fixed. The DDPG, although not as impressive as the DQN and BCRP, performs far better than the other strategies. It must

further be noted here that although the DQN performs better than the DDPG, we cannot use this instance to draw a general conclusion about the algorithms as there may be various cases where, due to different assets and hyperparameters, the DDPG may do better.

We also see that, on average, all strategies have a high Maximum Drawdown value. This is owing to the 2020 crash in markets resulting from COVID lockdowns across the world. We selected the test set to particularly include the crash in order to measure the downside risk and speed of recovery of the agents. From the results we find both agents have the lowest downside risk amongst the strategies they are compared with and make a recovery faster than a majority of the traditional strategies. For an extended comparison of our agents with these strategies on the MDD period and an expanded set of technical financial metrics, please refer to Appendix A.

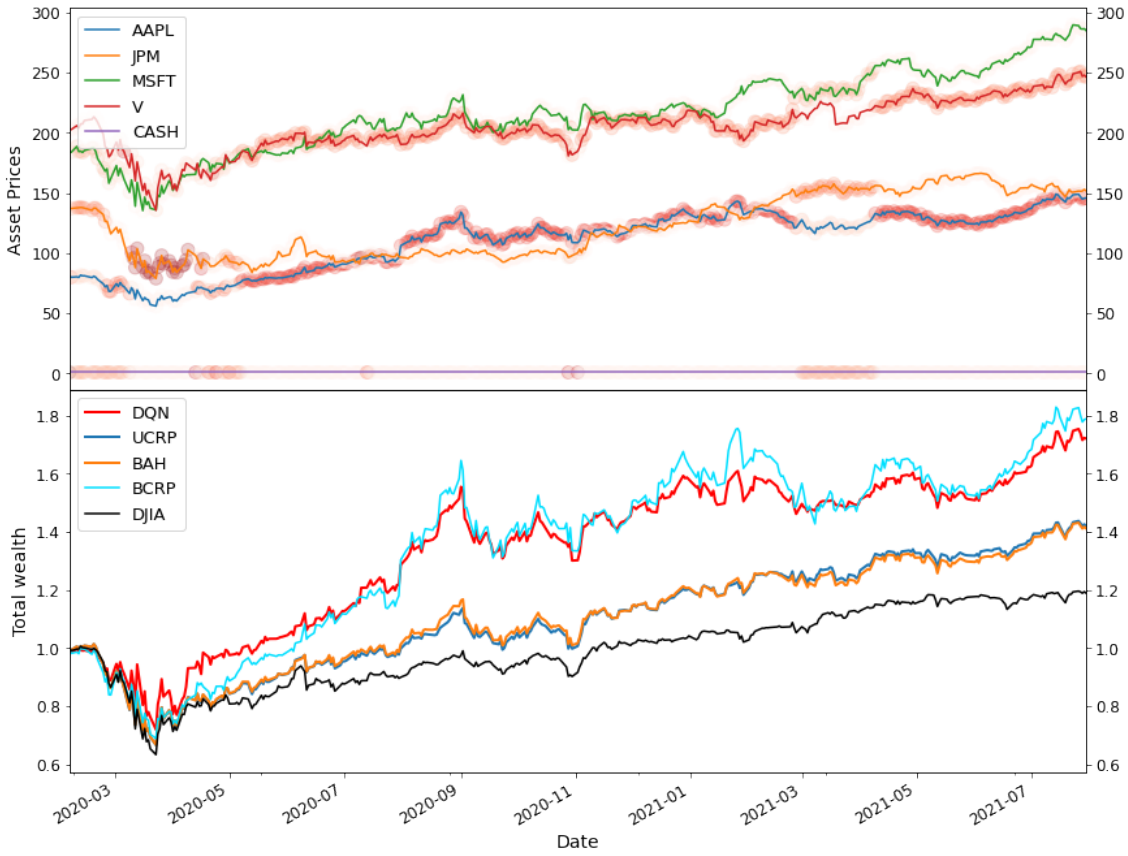

Figure 5.2: Asset weights and portfolio performance using DQN

Next, we illustrate the asset holdings over the investment period for each of the portfolios traded by our algorithms. This is presented in the form of a time-series heatmap in Figures 5.2 and

5.3 for the DQN and DDPG respectively. The heatmap corresponds to the proportion of wealth held in a particular asset. Higher this value, the darker the colour seen in the figure. To draw inferences about the effects of changes in asset weights on the portfolio performance we also include the cumulative return plot for the portfolio along with select benchmarks.

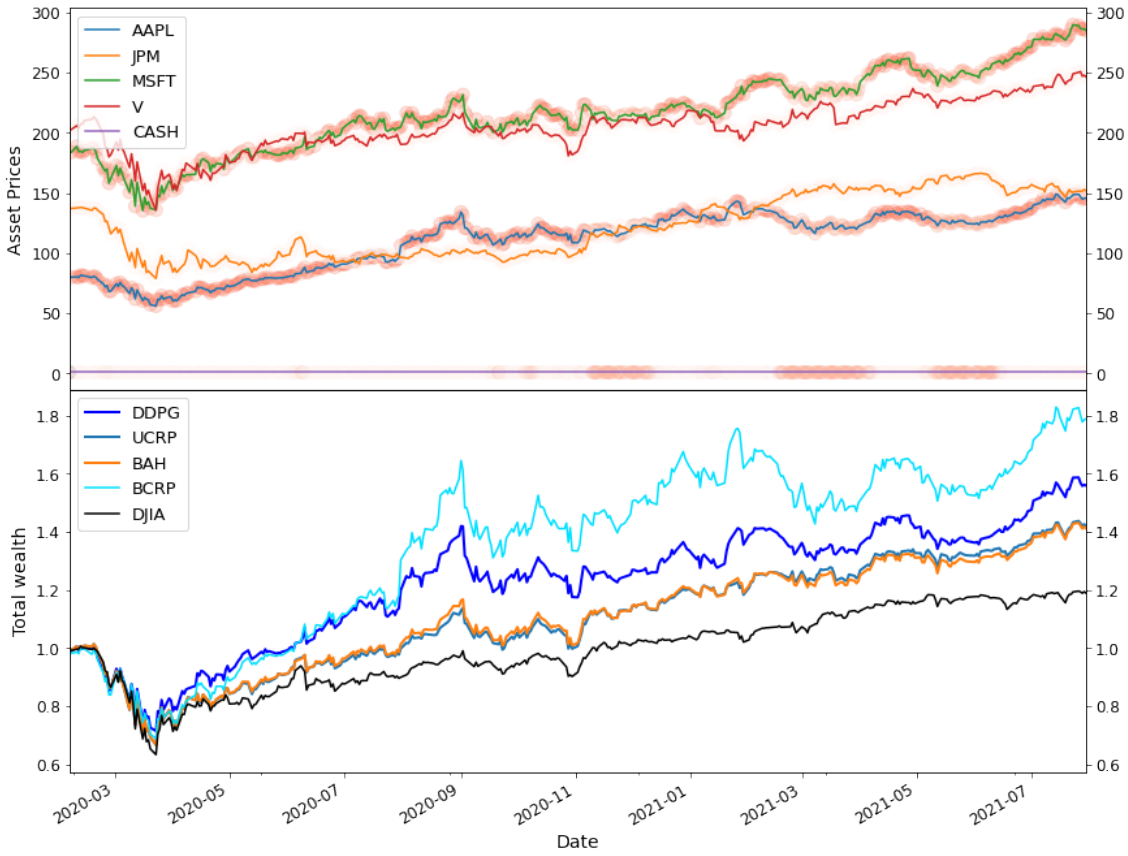

Figure 5.3: Asset weights and portfolio performance using DDPG

From these figures it is apparent that the agents have learnt to exploit certain strategies which enable them to beat the traditional strategies. These include holding on to historically well performing and less risky assets instead of making constant trades and incurring commission charges. In various instances when assets prices start to crash, we also see the agents quickly reduce the weights of these assets in the portfolio, and put back wealth into the asset only after the prices start recovering. Further, the agents shift wealth to the risk-free asset during periods of relatively higher volatility or during phases when the best assets are underperforming.

The models presented in this and the previous chapter are but two amongst the several that were

shortlisted. A few of the top performers are available alongside the code repository. These may be accessed and evaluated using the jupyter notebook also provided with the code.

5.2 Additional Experiments

In addition to the comparisons made above between trained agents and traditional portfolio selection strategies, we conduct experiments to evaluate the generalisation and learning ability of the agents.

In particular, we conduct two sets of tests. The first evaluates the existing agents on their performance when trading 10 randomly selected portfolios from the DJIA. This allows us to gauge how well the agent has learned to detect universal patterns in stock price data that may allow it to be profitable or make risk-sensitive trades. Second, we use the same learning algorithm, network architecture and hyperparameters as presented earlier to train new agents on 4-asset portfolios comprising of different assets, once more selected from the DJIA. This provides an idea of the learning capacity of agents with the same architecture and hyperparameters for a given algorithm. For a complete list of the Dow Jones Index constituents used here, please refer to Appendix B.

We use the same testing period for these experiments as the original test set and restrict ourselves to portfolios with four assets (other that the risk-free asset). This was not specifically required, and is only to make comparisons between models feel more standardised. The performance of the existing agents on different asset portfolios is summarised in Table 5.2 using the same metrics previously employed. Similarly, the retrained agents are evaluated in Table 5.3. Please note that although the performance of agents in these trials, based on the metrics alone, seems significantly worse than the results obtained in our primary set of results, such a comparison is baseless as the assets in the portfolios being compared are entirely different. For a detailed understanding of these results, please refer to Appendices C and D where we provide figures comparing agent performance for each portfolio alongside benchmarks for that particular portfolio.

In general, we find that both sets of experiments show positive results. The existing agents when tested on different assets, in most cases, perform as well as or better than the BAH and UCRP benchmarks *for that portfolio*. Performance of this level provides sufficient evidence to conclude that the agents are able to generalise to and trade novel portfolio by exploiting the strategies mentioned earlier. In the second set of experiments we find that agents may be trained on data from a variety of unique portfolios and still beat benchmarks. This illustrates the flexibility and learning potential of the chosen combinations of network architectures and hyperparameters.

Tickers	DQN				DDPG			
	Cumulative Return (%)	**Annualised Return (%)**	**Sharpe ratio**	**MDD (%)**	**Cumulative Return (%)**	**Annualised Return (%)**	**Sharpe ratio**	**MDD (%)**
WBA, JPM, MRK, WMT	15.41	13.41	0.49	24.34	7.7	9.97	0.32	29.61
UNH, AMGN, NKE, VZ	10.03	10.18	0.37	23.77	14.87	13.92	0.46	27.63
BA, INTC, V, UNH	-28.34	-13.14	-0.31	46.66	8.98	13.36	0.34	42.76
MSFT, GS, BA, AMGN	-40.05	-21.19	-0.41	58.65	18.2	16.11	0.52	30.68
VZ, MSFT, AXP, KO	28.72	21.98	0.7	24.66	39.69	27.37	0.88	30.26
WMT, MMM, MCD, JNJ	11.22	10.64	0.4	23.12	11.46	9.96	0.43	23.86
NKE, GS, CSCO, MSFT	25.09	21.89	0.6	41.52	25.63	20.45	0.64	28.78
V, TRV, MCD, CRM	20.33	20.09	0.51	33.41	22.59	18.88	0.59	30.73
WMT, MCD, JNJ, INTC	-30.08	-20.9	-0.82	35.21	−1.93	2.39	0.09	25.48
AXP, JNJ, TRV, PG	-3.38	4.17	0.12	37.65	22.95	17.63	0.65	28.2

Table 5.2: Performance metrics when trading different portfolios

Tickers	Model	Cumulative Return (%)	Annualised Return (%)	Sharpe ratio	MDD (%)
DQN					
AXP, CVX, DIS, KO	DQN_A	25.94	25.25	0.57	46.62
JNJ, MCD, MMM, WMT	DQN_B	35.31	23.05	1.01	19.95
CAT, CSCO, HD, IBM	DQN_C	51.93	33.72	1.03	30.57
DDPG					
JNJ, MCD, MMM, WMT	DDPG_A	17.59	13.51	0.6	21.8
AMGN, NKE, UNH, VZ	DDPG_B	30.94	22.71	0.76	28.8
GS, NKE, PG, UNH	DDPG_C	49.62	32.18	1.02	34.04

Table 5.3: Performance metrics for retrained agents

Chapter 6

Conclusion

To conclude this report, we first summarise all our contributions, before moving on the discuss how this work may be extended in the future.

Although most of the work undertaken in this dissertation is based on prior work on the topic of Reinforcement Learning for Portfolio Management, we have made several novel contributions. First, we provide one of the most comprehensive surveys of the literature relating to the application of the DQN and DDPG algorithms for portfolio allocation. Though there exist some surveys on the application of RL in finance which also look at portfolio management (Fischer, 2018; Kolm & Ritter, 2019; Mosavi et al., 2020; Sato, 2019), this is the first time there has been a focussed comparison of specifically the DQN and DDPG algorithms. Secondly, we provide an extremely user-friendly and completely customisable tool in the form of the *alphaQ* package which can enable absolute beginners to experiment with the Reinforcement Learning algorithms for portfolio management. Not only does it provide a complete framework for the training, validation, test pipeline, it also makes it extremely easy to plug in any one of the Deep RL algorithms implemented in Stable-Baselines3 and compare performance across numerous benchmarks and baseline portfolio selection strategies. Along with a detailed study of the Reinforcement Learning and Portfolio Management theory, which we provide in Chapter 2, and the specifics of our set up, experimentation and results (Chapters 4 and 5), this dissertation provides a thorough understanding and analysis of building RL agents for portfolio management.

As for the particulars of the agents trained, we find that after some amount of hyperparameter tuning we are able to build trading agents that can consistently perform better than numerous benchmarks and baselines. In addition to this, our experiments indicate that the same set of

hyperparameters may be used to train agents on completely different assets. In some cases the same agents with the same network weights themselves (without retraining) can beat benchmarks when evaluated on other assets. Both these points together highlight the considerable generalisation and learning ability of the models created using our framework.

6.1 Future Work

In spite of the comprehensive nature of this report, we have only scratched the surface in terms of all that can be done to take this work forward. There were many ideas which had to be discarded as there wasn't enough time to implement, evaluate, or test them sufficiently. The best of these are presented here for anyone who would like to work on it.

The models we built can, at best, be considered generic given that the framework we used, Stable-Baselines3, is specifically created to build models for a wide variety of different Reinforcement Learning tasks. It can be expected that using network architectures built and optimised specifically for the portfolio optimisation problem will lead to much better results, in line with some of the research covered earlier in the literature survey. Alternatively, it would be worthwhile to test the DQN and DDPG implementations of other existing Reinforcement Learning frameworks, and find out how they compare to our implementations.

As for the hyperparameter tuning, due to the sheer number of tunable parameters, we have, in all likelihood, only explored a very small portion of the solution space. With greater time and resources devoted to hyperparameter optimization the outcomes could be much improved.

There is also a possibility of making improvements to the feature set used to train the agents. Some of the research we came across utilised features in addition to the Open, High, Low and Close prices. Specifically, stock fundamental data, further augmented with features extracted from open or public sources such as sentiments from social media, or company outlook from corporate disclosure reports, could help the agent learn to trade better by filling in the blanks not evident in the stock prices alone.

Further changes could be made to the observations passed to the agent as well. Instead of passing portfolio weights from the previous trading day only, a vector comprising weights from the entire observation period could be used. This may then allow the use of a reward signal comprising of the Sharpe ratio, which we had to abandon, as the agent will be better equipped to associate volatility in the portfolio with changes in the asset weights - in addition to arising from price movements alone.

The reward itself could be calculated differently. In our case, instead of using the daily return as the reward for each trading day, it might make sense to calculate the reward over the risk free rate or the market's rate of return. This might be more intuitive as performing worse than the market, for example, should not be rewarded. As such, the agent may further benefit from being passed the market history as part of the observations when deciding on an action.

Certain ideas presented in the reviewed literature may also be applied to our implementation. Two noteworthy ones are the ability to take up short positions, which opens a whole new dimension to the feasible actions an agent may take, and volatility penalisation to disincentivise actions which lead to higher portfolio variance.

Finally, to achieve further gains in performance, it would be quite suitable to build ensemble models which execute trading actions based on cumulative decisions from multiple agents or models. These may or may not be solely Reinforcement Learning based and could incorporate existing - and better understood - online portfolio selection strategies similar to the ones we presented in Section 2.3.5.

Bibliography

Agarwal, A., Hazan, E., Kale, S., & Schapire, R. E. (2006). Algorithms for portfolio management based on the newton method. *Proceedings of the 23rd International Conference on Machine Learning*, 9–16. https://doi.org/10.1145/1143844.1143846

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., & Zaremba, W. (2017). Hindsight experience replay. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 5055–5065.

Aroussi, R. (2017). *Yfinance*. https://github.com/ranaroussi/yfinance

Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 449–458.

Bellman, R. (1957). *Dynamic Programming*. Dover Publications.

Borodin, A., El-Yaniv, R., & Gogan, V. (2004). Can we learn to beat the best stock. *Journal of Artificial Intelligence Research*, *21*, 579–594. https://doi.org/10.1613/jair.1336

Cover, T. M. (1991). Universal portfolios. *Mathematical Finance*, *1*(1), 1–29. https://doi.org/https://doi.org/10.1111/j.1467-9965.1991.tb00002.x

Du, X., Zhai, J., & Lv, K. (2009). Algorithm trading using q-learning and recurrent reinforcement learning. *Positions*, *1*.

Feldman, R. M., & Valdez-Flores, C. (2010). Markov processes. *Applied probability and stochastic processes* (pp. 181–199). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-05158-6_6

Fischer, T. G. (2018). *Reinforcement learning in financial markets - a survey* (FAU Discussion Papers in Economics No. 12/2018). Friedrich-Alexander University Erlangen-Nuremberg, Institute for Economics. https://ideas.repec.org/p/zbw/iwqwdp/122018.html

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international con-*

ference on machine learning (pp. 1587–1596). PMLR. http://proceedings.mlr.press/v80/fujimoto18a.html

Gao, L., & Zhang, W. (2013). Weighted moving average passive aggressive algorithm for online portfolio selection. *2013 5th International Conference on Intelligent Human-Machine Systems and Cybernetics*, *1*, 327–330. https://doi.org/10.1109/IHMSC.2013.84

Gao, Z., Gao, Y., Hu, Y., Jiang, Z., & Su, J. (2020). Application of deep q-network in portfolio management. *2020 5th IEEE International Conference on Big Data Analytics (ICBDA)*, 268–275.

Györfi, L., Udina, F., & Walk, H. (2008). Nonparametric nearest neighbor based empirical portfolio selection strategies. *26*(2), 145–157. https://doi.org/doi:10.1524/stnd.2008.0917

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th international conference on machine learning* (pp. 1861–1870). PMLR. http://proceedings.mlr.press/v80/haarnoja18b.html

Hakansson, N. H., & Ziemba, W. T. (1995). Chapter 3 capital growth theory. *Finance* (pp. 65–86). Elsevier. https://doi.org/https://doi.org/10.1016/S0927-0507(05)80047-7

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. https://doi.org/10.1109/CVPR.2016.90

Helmbold, D., Schapire, R., Singer, Y., & Warmuth, M. K. (1996). On-line portfolio selection using multiplicative updates. *ICML*, 243–251.

Huang, D., Zhou, J., Li, B., Hoi, S. C. H., & Zhou, S. (2016). *Robust median reversion strategy for online portfolio selection*. https://doi.org/10.1109/TKDE.2016.2563433

Huang, G., Zhou, X., & Song, Q. (2020). Deep reinforcement learning for portfolio management based on the empirical study of chinese stock market. *arXiv:2012.13773*.

Jiang, Z., & Liang, J. (2017). Cryptocurrency portfolio management with deep reinforcement learning. *2017 Intelligent Systems Conference (IntelliSys)*, 905–913.

Jiang, Z., Xu, D., & Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem. *arXiv:1706.10059*.

Jin, O., & El-Saawy, H. (2016). Portfolio management usig reinforcement learning.

Kelly, J. L. (1956). *A new interpretation of information rate*. https://doi.org/10.1002/j.1538-7305.1956.tb03809.x

Kolm, P., & Ritter, G. (2019). Modern perspectives on reinforcement learning in finance. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.3449401

Li, B., & Hoi, S. C. H. (2012). On-line portfolio selection with moving average reversion. *ICML*. http://icml.cc/2012/papers/168.pdf

Li, B., & Hoi, S. C. H. (2014). Online portfolio selection: A survey. *ACM Computing Surveys*, *46*(3). https://doi.org/10.1145/2512962

Li, B., Hoi, S. C., & Gopalkrishnan, V. (2011). Corn: Correlation-driven nonparametric learning approach for portfolio selection. *ACM Transactions on Intelligent Systems and Technology*, *2*(3), 1–21. https://doi.org/10.1145/1961189.1961193

Li, B., Hoi, S. C., Zhao, P., & Gopalkrishnan, V. (2011). Confidence weighted mean reversion strategy for on-line portfolio selection. In G. Gordon, D. Dunson, & M. Dudík (Eds.), *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 434–442). PMLR. http://proceedings.mlr.press/v15/li11b.html

Li, B., Zhao, P., Hoi, S., & Gopalkrishnan, V. (2012). Pamr: Passive aggressive mean reversion strategy for portfolio selection. *Machine Learning*, *87*. https://doi.org/10.1007/s10994-012-5281-z

Li, X., Li, Y., Zhan, Y., & Liu, X.-Y. (2019). Optimistic bull or pessimistic bear: Adaptive deep reinforcement learning for stock portfolio allocation. *arXiv:1907.01503*.

Liang, Z., Chen, H., Zhu, J., Jiang, K., & Li, Y. (2018). *Adversarial Deep Reinforcement Learning in Portfolio Management* (Papers No. 1808.09940). arXiv.org. https://ideas.repec.org/p/arx/papers/1808.09940.html

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Y. Bengio & Y. LeCun (Eds.), *Iclr*. http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15

Lin, L.-J. (1992). *Reinforcement learning for robots using neural networks* (Doctoral dissertation) [UMI Order No. GAX93-22750]. USA, Carnegie Mellon University.

Lucarelli, G., & Borrotti, M. (2020). A deep q-learning portfolio management framework for the cryptocurrency market. *Neural Computing and Applications*, *32*, 17229–17244. https://doi.org/10.1007/s00521-020-05359-8

Magdon-Ismail, M., Atiya, A. F., Pratap, A., & Abu-Mostafa, Y. S. (2004). On the maximum drawdown of a brownian motion. *Journal of Applied Probability*, *41*(1), 147–161. http://www.jstor.org/stable/3215821

Mann, A. (2021). *alphaQ*. https://github.com/mannmann2/alphaQ

Marigold. (2014). *Universal portfolios*. https://github.com/Marigold/universal-portfolios

Markowitz, H. (1952). Portfolio selection. *The Journal of Finance*, *7*(1), 77–91. http://www.jstor.org/stable/2975974

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In M. F. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33rd international conference on machine learning* (pp. 1928–1937). PMLR. http://proceedings.mlr.press/v48/mniha16.html

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *Nips deep learning workshop*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. http://dx.doi.org/10.1038/nature14236

Mosavi, A., Faghan, Y., Ghamisi, P., Duan, P., Ardabili, S. F., Salwana, E., & Band, S. S. (2020). Comprehensive review of deep reinforcement learning methods and applications in economics. *Mathematics*, *8*(10). https://doi.org/10.3390/math8101640

OpenAI. (2016). *OpenAI Gym*. https://github.com/openai/gym

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. (2019). *Stable Baselines3*. https://github.com/DLR-RM/stable-baselines3

Sato, Y. (2019). Model-free reinforcement learning for financial portfolios: A brief survey. *ArXiv*, *abs/1904.04973*.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay. *International Conference on Learning Representations (ICLR*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, *abs/1707.06347*. http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17

Sharpe, W. F. (1994). The sharpe ratio. *The Journal of Portfolio Management*, *21*(1), 49–58. https://doi.org/10.3905/jpm.1994.409501

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, I–387–I–395.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd). The MIT Press.

Tsitsiklis, J., & Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*(5), 674–690. https://doi.org/10.1109/9.580874

Uhlenbeck, G. E., & Ornstein, L. S. (1930). On the theory of the brownian motion. *Phys. Rev.*, *36*, 823–841. https://doi.org/10.1103/PhysRev.36.823

Watkins, C., & Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, *8*, 279–292. https://doi.org/10.1007/BF00992698

Xiong, Z., Liu, X.-Y., Zhong, S., Yang, H., & Walid, A. (2018). Practical deep reinforcement learning approach for stock trading. *arXiv:1811.07522*.

Yahoo. (1997). *Yahoo Finance*. https://finance.yahoo.com/

Zhang, H., Jiang, Z., & Su, J. (2021). A deep deterministic policy gradient-based strategy for stocks portfolio management. *2021 IEEE 6th International Conference on Big Data Analytics (ICBDA)*, 230–238. https://doi.org/10.1109/ICBDA51983.2021.9403049

Zhang, Z., Zohren, S., & Stephen, R. (2020). Deep reinforcement learning for trading. *The Journal of Financial Data Science*. https://doi.org/10.3905/jfds.2020.1.030

Appendix A

Additional Metrics

Algorithm	MDD Period (days)	Ulcer Index	Profit Factor	Winning days (%)	Beta	Alpha	Appraisal Ratio	Information Ratio	Annualised Volatility	Annual Turnover
DDPG	149.53	4.26	1.17	54.28	0.96	0.08	0.51	0.41	0.37	5.38
DQN	105.67	6.28	1.21	55.61	1.04	0.13	0.86	0.93	0.39	24.7
BAH	113.64	3.26	1.14	55.08	1.0	-0.01	-0.29	-0.35	0.35	0.0
UCRP	113.64	3.32	1.15	54.28	1.0	-0.01	-4.82	-3.8	0.35	2.44
BCRP	110.65	4.41	1.19	52.82	1.0	0.17	0.79	0.79	0.41	0.0
Min Variance	186.41	1.42	1.08	54.81	0.96	-0.09	-0.71	-0.82	0.36	25.66
UP	113.64	3.3	1.15	54.55	1.0	-0.01	-1.0	-1.37	0.35	1.94
EG	113.64	3.32	1.15	54.55	1.0	-0.01	-2.39	-3.07	0.35	2.31
Anticor	103.67	3.24	1.12	52.67	0.99	0.01	0.06	0.02	0.38	34.74
PAMR	367.84	0.03	0.97	54.81	0.95	-0.28	-2.19	-2.32	0.36	100.73
OLMAR	110.65	1.89	1.05	50.8	1.03	-0.12	-0.65	-0.62	0.4	56.79
RMR	144.54	2.01	1.06	51.87	1.02	-0.1	-0.53	-0.51	0.4	57.34
CWMR	371.83	-1.02	0.77	44.92	1.03	-0.77	-3.69	-3.67	0.41	354.09
WMAMR	258.18	0.73	1.01	51.34	1.16	-0.22	-0.96	-0.74	0.47	78.76
BNN	153.51	1.39	1.04	51.08	0.98	-0.14	-0.94	-0.99	0.38	34.97
CORN	175.45	0.13	0.98	39.72	0.49	-0.13	-0.56	-0.98	0.29	244.68
ONS	129.59	3.37	1.15	52.67	1.0	0.0	0.03	0.01	0.36	8.75
DJIA	191.4	1.61	1.08	55.61	1.0	-0.0	-0.0	-0.82	0.31	0.0

Table A.1: Expanded comparison of models on financial metrics

Appendix B

Dow Jones Industrial Average Constituents

Ticker Symbol	Company	Ticker Symbol	Company
AAPL	Apple Inc	JNJ	Johnson & Johnson
AMGN	Amgen Inc	JPM	JPMorgan Chase & Co
AXP	American Express Co	KO	Coca-Cola Co
BA	Boeing Co	MCD	McDonald's Corp
CAT	Caterpillar Inc	MMM	3M Co.
CRM	Salesforce.com Inc	MRK	Merck & Co Inc
CSCO	Cisco Systems Inc	MSFT	Microsoft Corp
CVX	Chevron Corp	NKE	Nike Inc
DIS	Walt Disney Co	PG	Proctor & Gamble Co
DOW	Dow Inc	TRV	Travelers Companies Inc
GS	Goldman Sachs Group Inc	UNH	UnitedHealth Group Inc
HD	Home Depot Inc	V	Visa Inc
HON	Honeywell International Inc	VZ	Verizon Communications Inc
IBM	IBM Corp	WBA	Walgreens Boots Alliance Inc
INTC	Intel Corp	WMT	Walmart Inc

Table B.1: Constituents of the DJIA Index (as on July 1, 2021)

Appendix C

Experiments on Agent Generalisation Ability

C.1 DQN

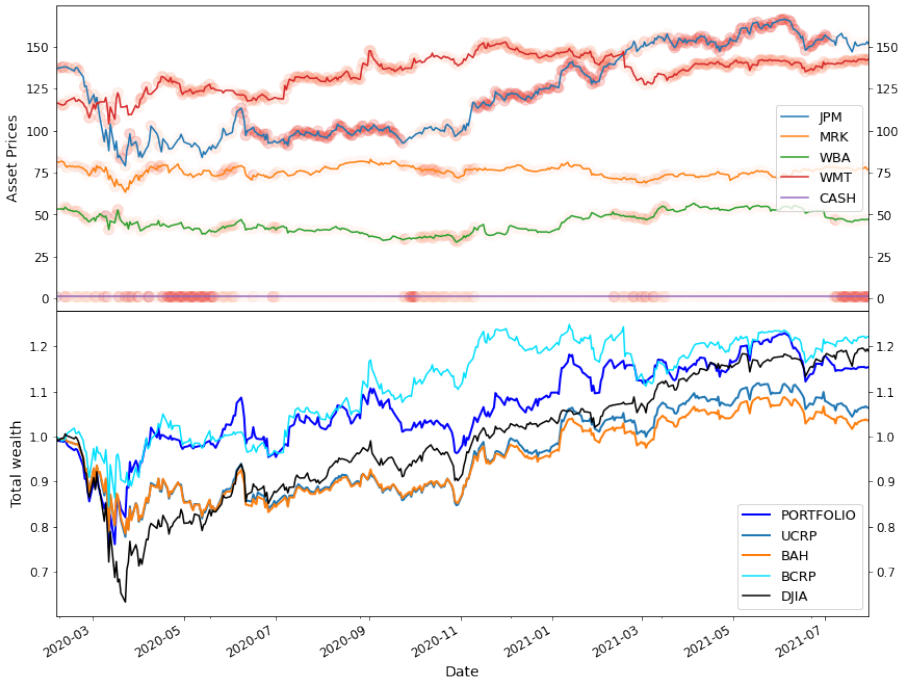

Figure C.1: Best DQN on JPM, MRK, WBA, WMT

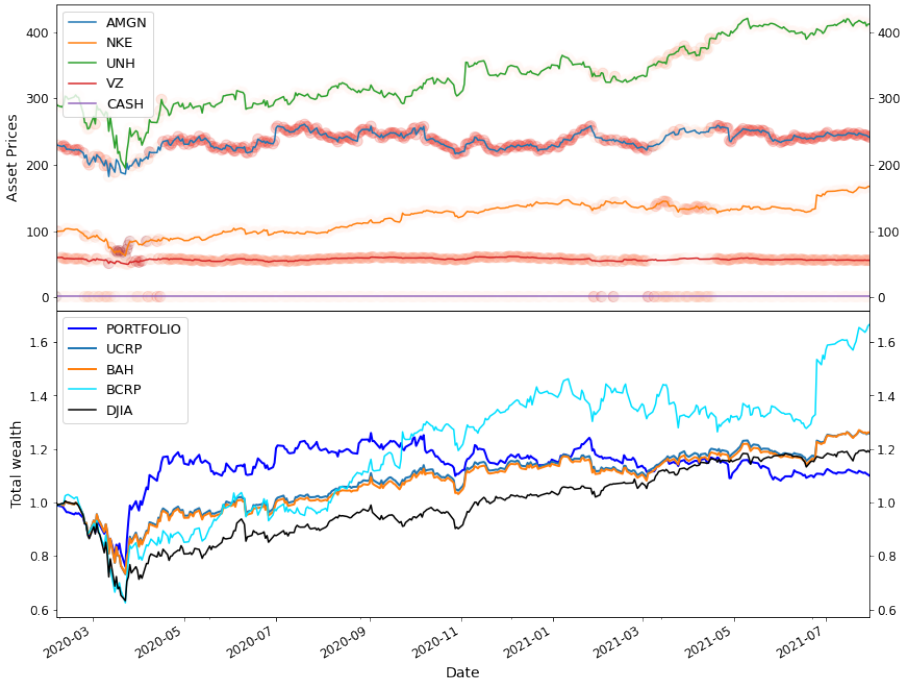

Figure C.2: Best DQN on AMGN, NKE, UNH, VZ

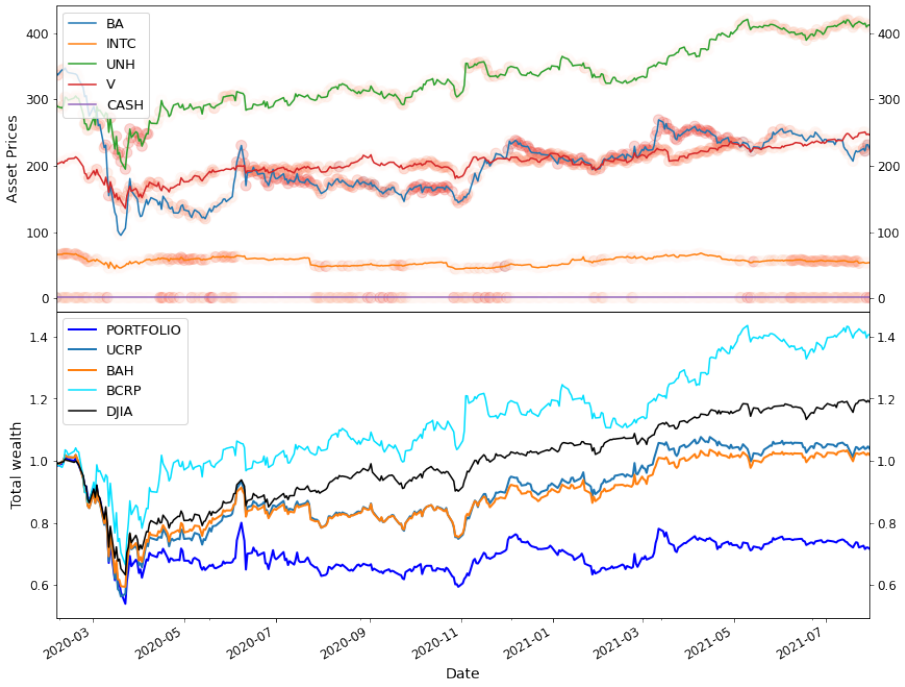

Figure C.3: Best DQN on BA, INTC, UNH, V

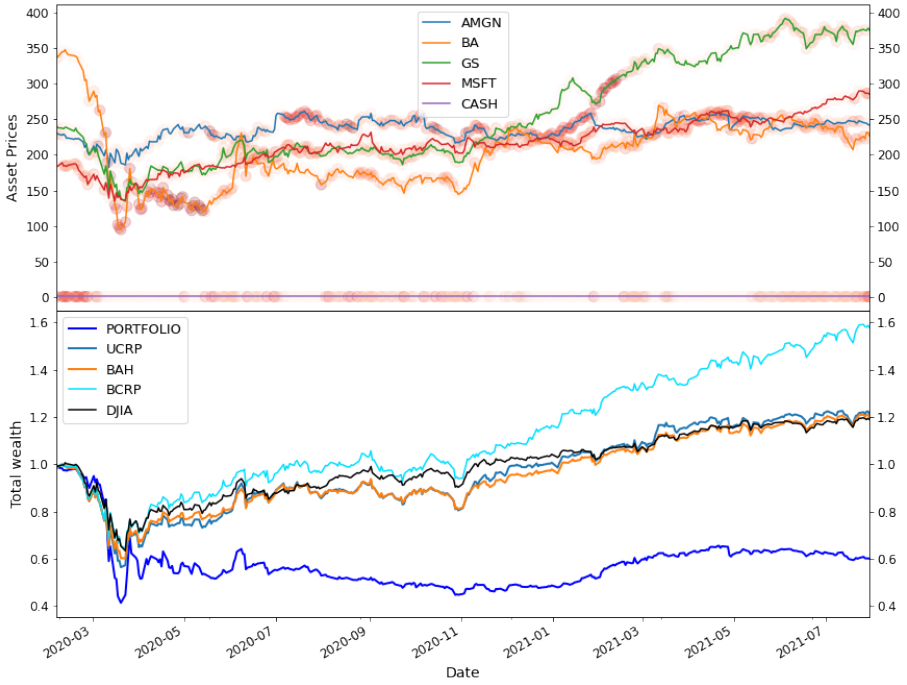

Figure C.4: Best DQN on AMGN, BA, GS, MSFT

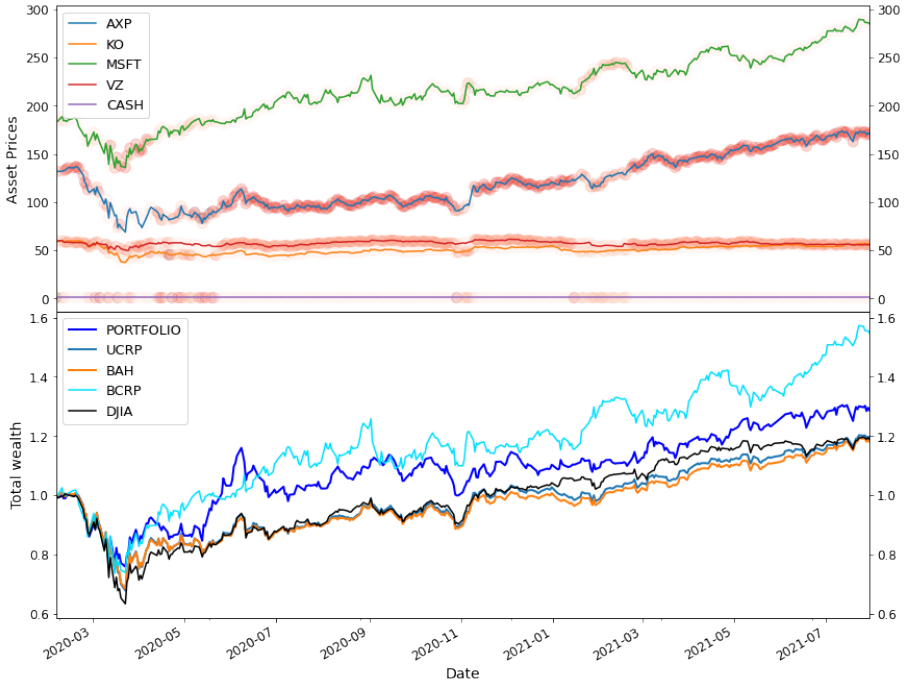

Figure C.5: Best DQN on AXP, KO, MSFT, VZ

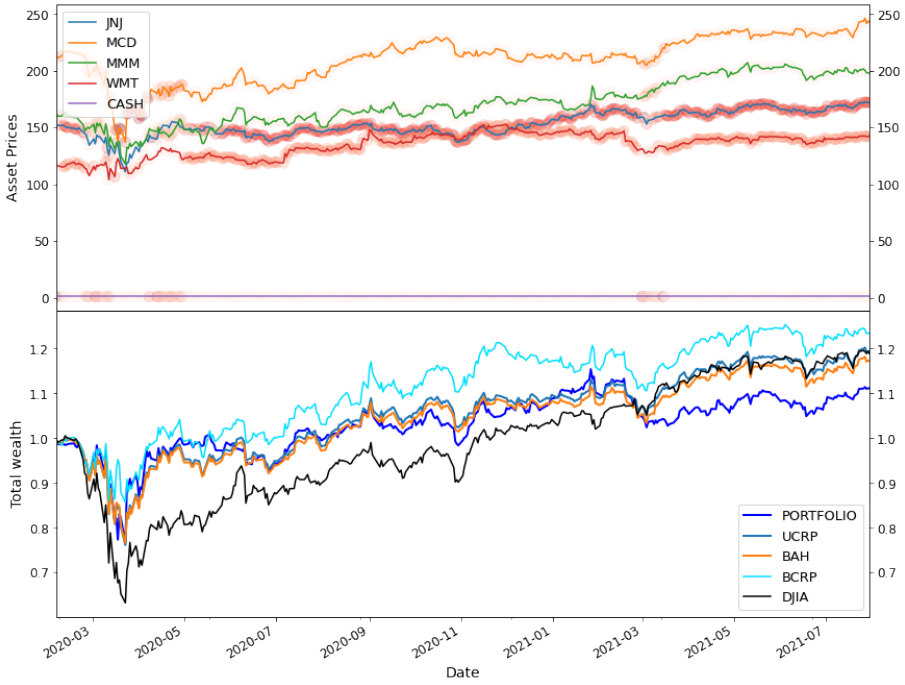

Figure C.6: Best DQN on JNJ, MCD, MMM, WMT

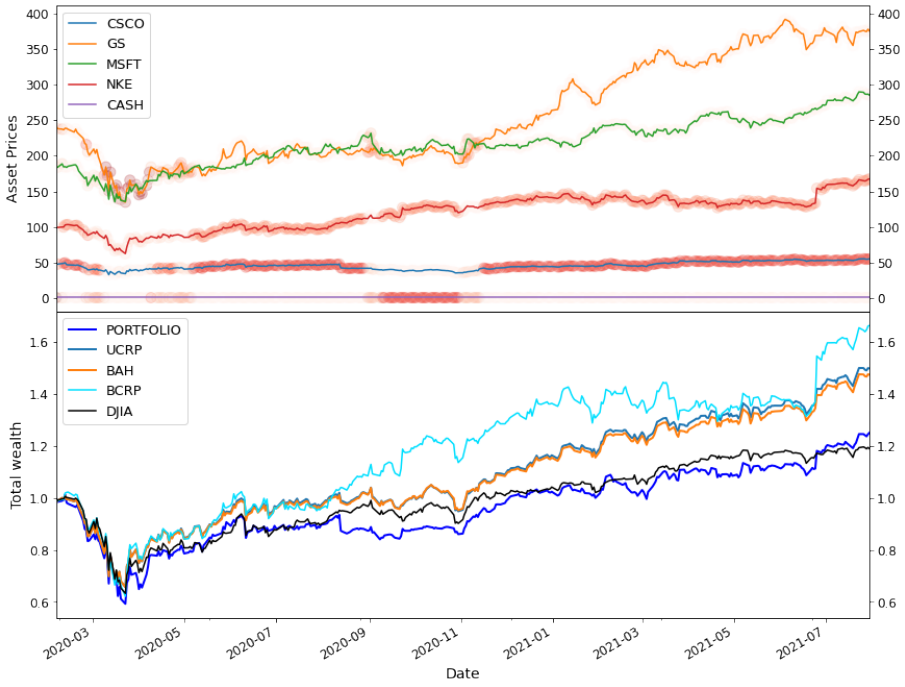

Figure C.7: Best DQN on CSCO, GS, MSFT, NKE

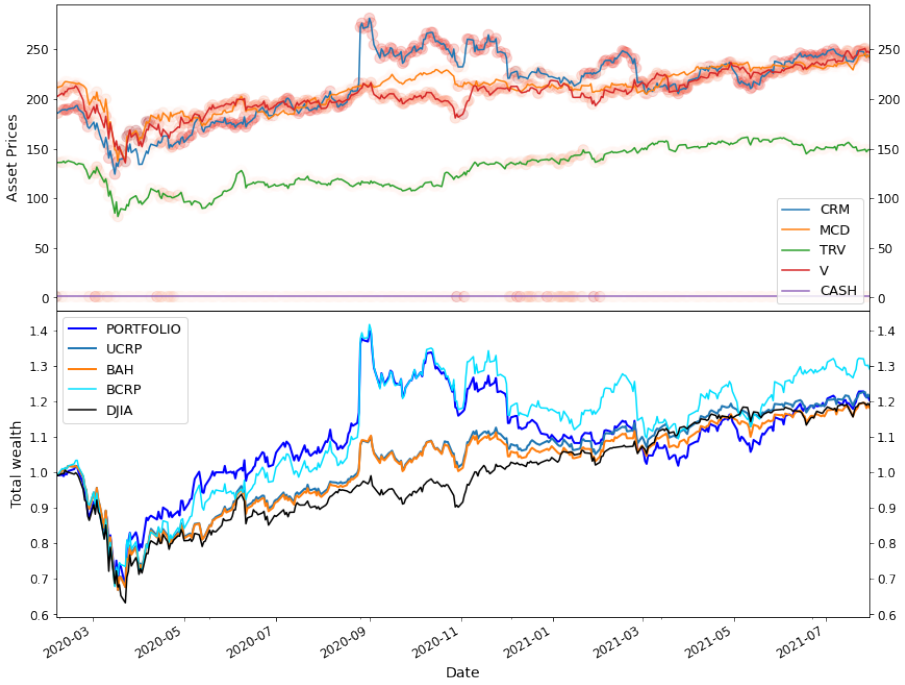

Figure C.8: Best DQN on CRM, MCD, TRV, V

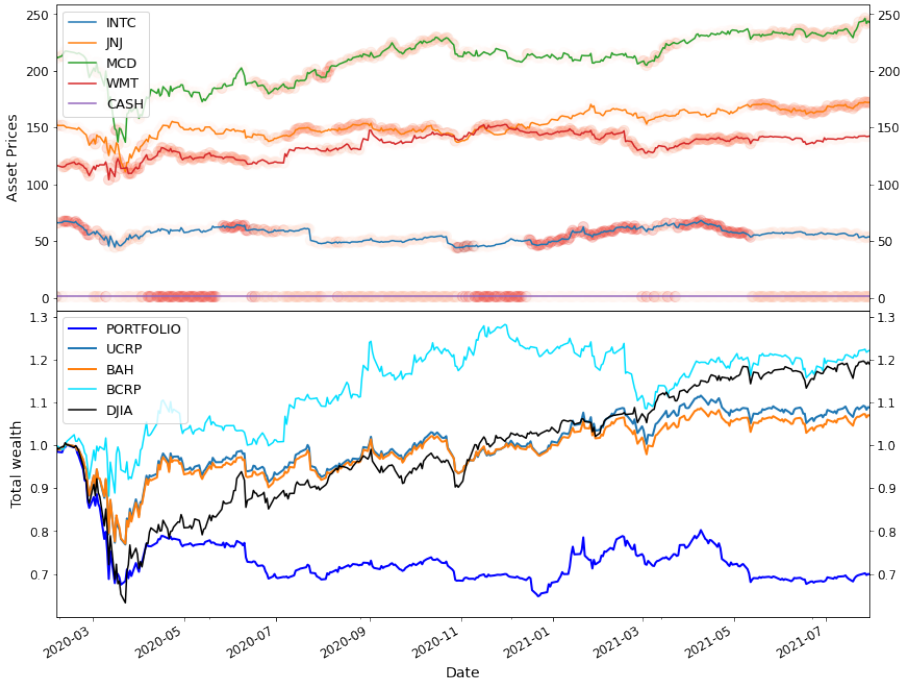

Figure C.9: Best DQN on INTC, JNJ, MCD, WMT

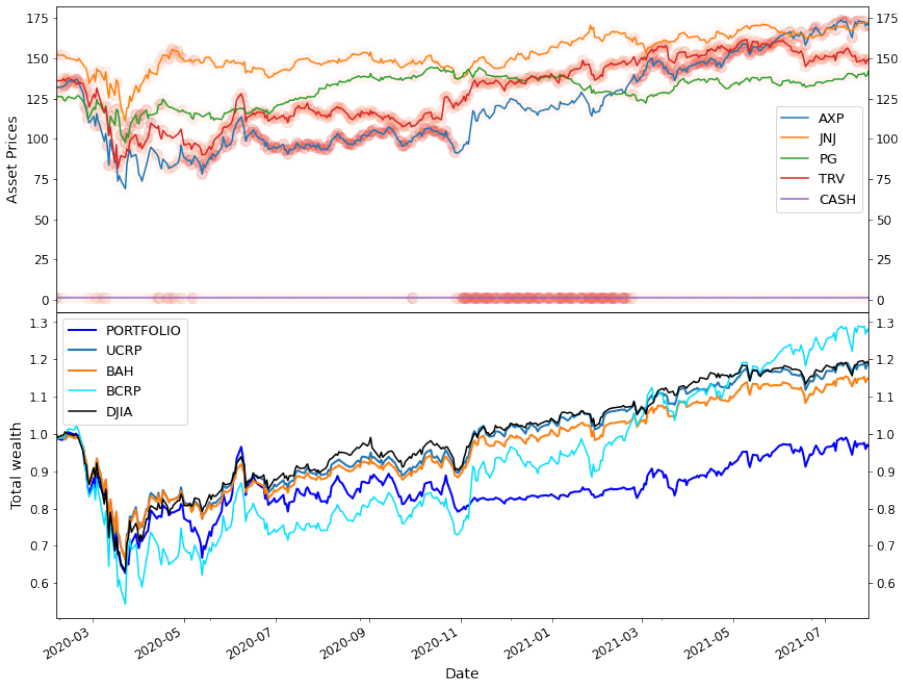

Figure C.10: Best DQN on AXP, JNJ, PG, TRV

C.2 DDPG

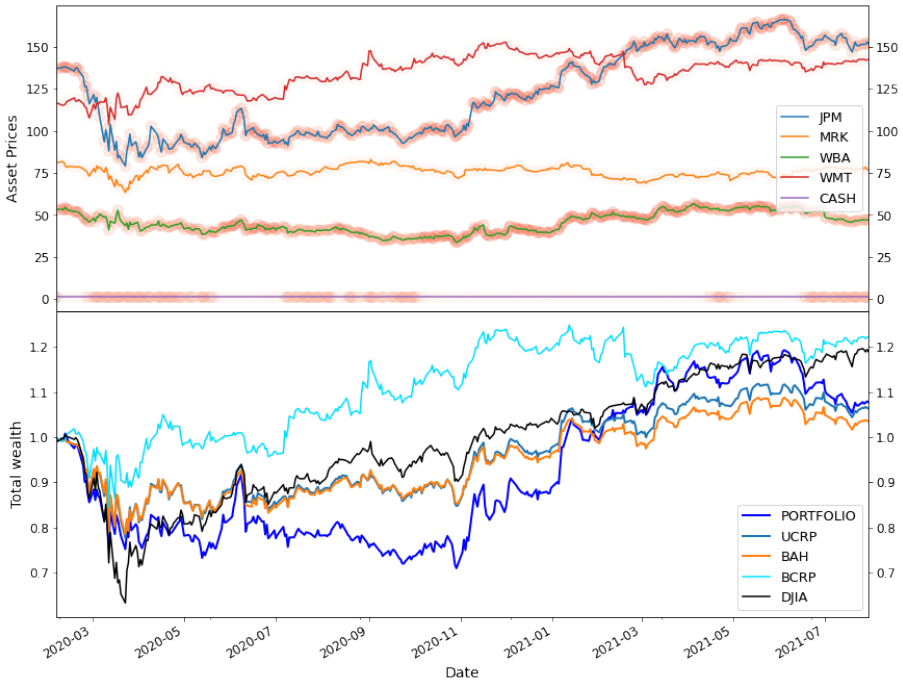

Figure C.11: Best DDPG on JPM, MRK, WBA, WMT

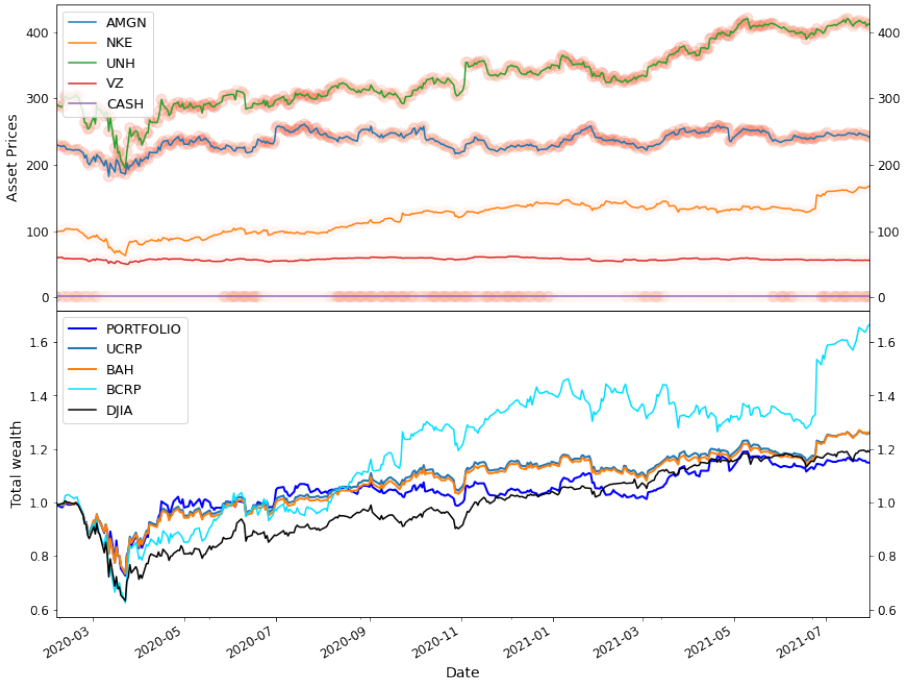

Figure C.12: Best DDPG on AMGN, NKE, UNH, VZ

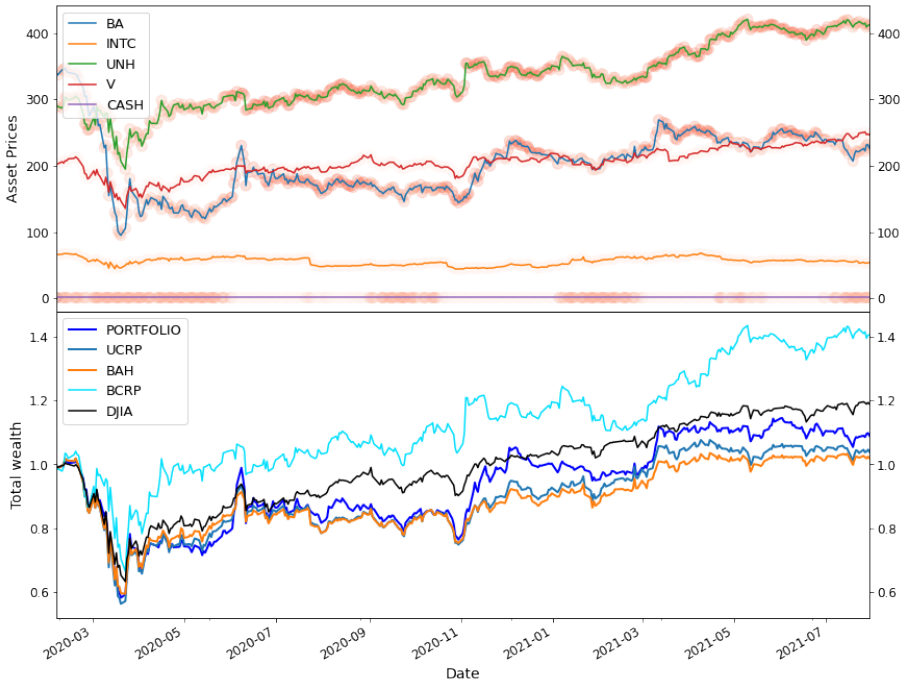

Figure C.13: Best DDPG on BA, INTC, UNH, V

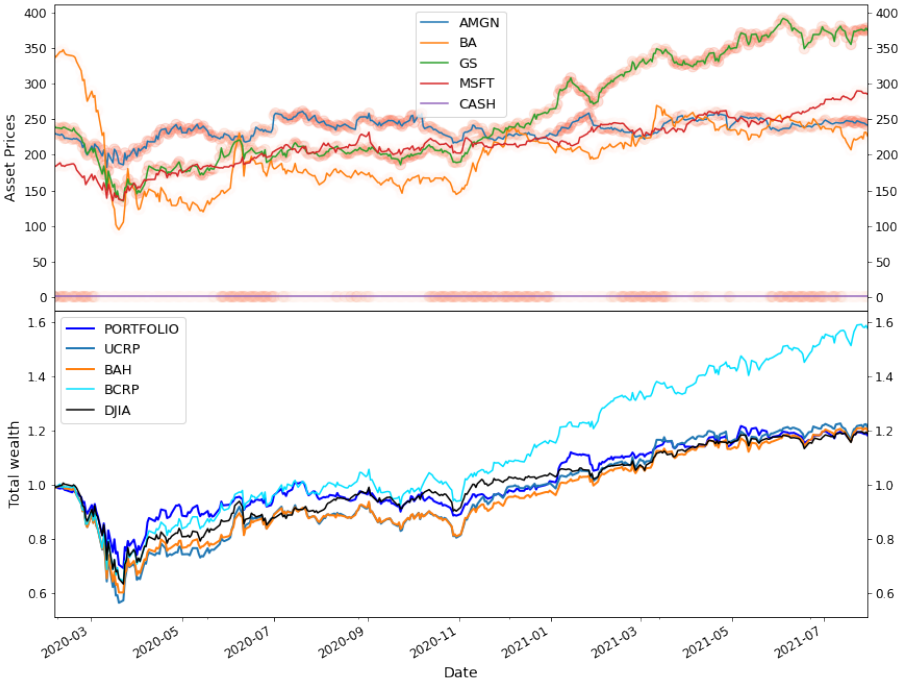

Figure C.14: Best DDPG on AMGN, BA, GS, MSFT

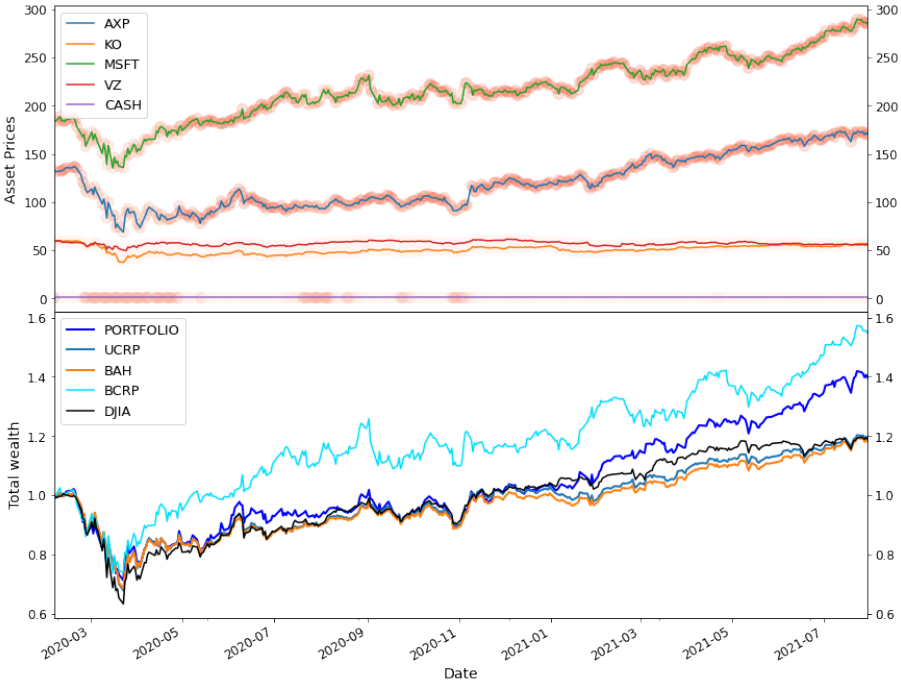

Figure C.15: Best DDPG on AXP, KO, MSFT, VZ

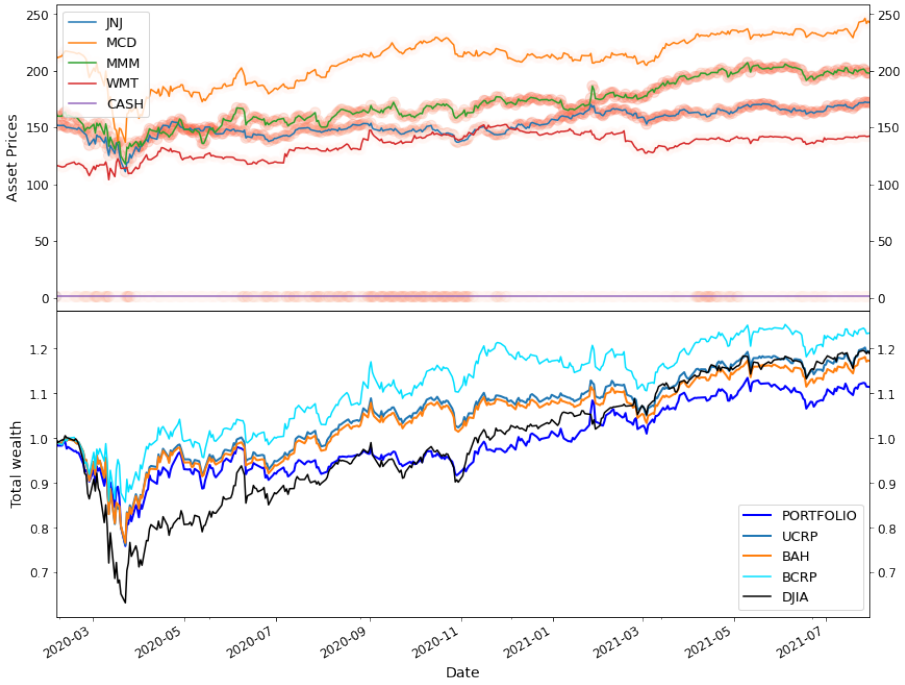

Figure C.16: Best DDPG on JNJ, MCD, MMM, WMT

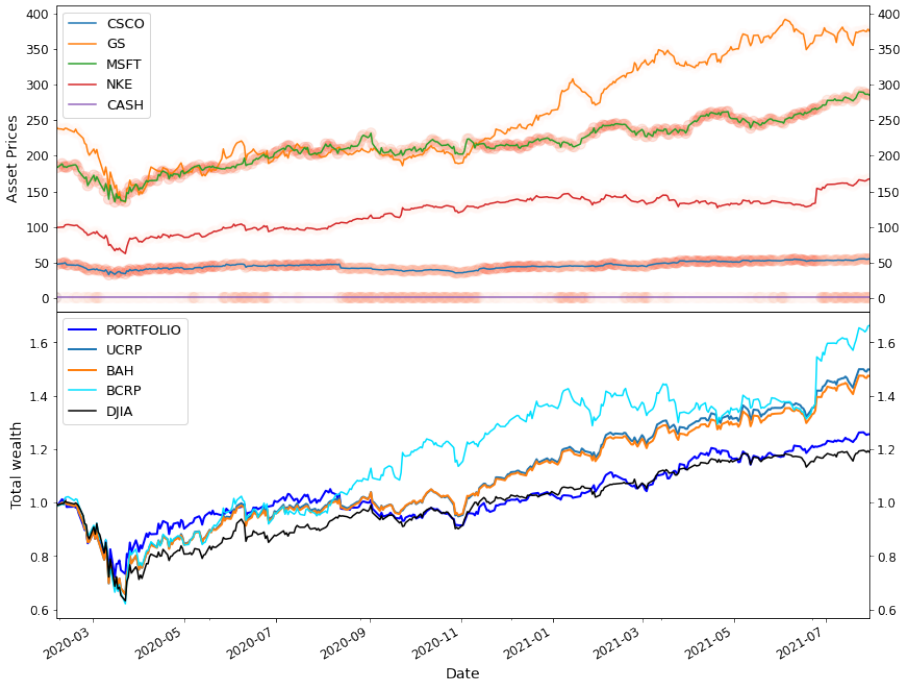

Figure C.17: Best DDPG on CSCO, GS, MSFT, NKE

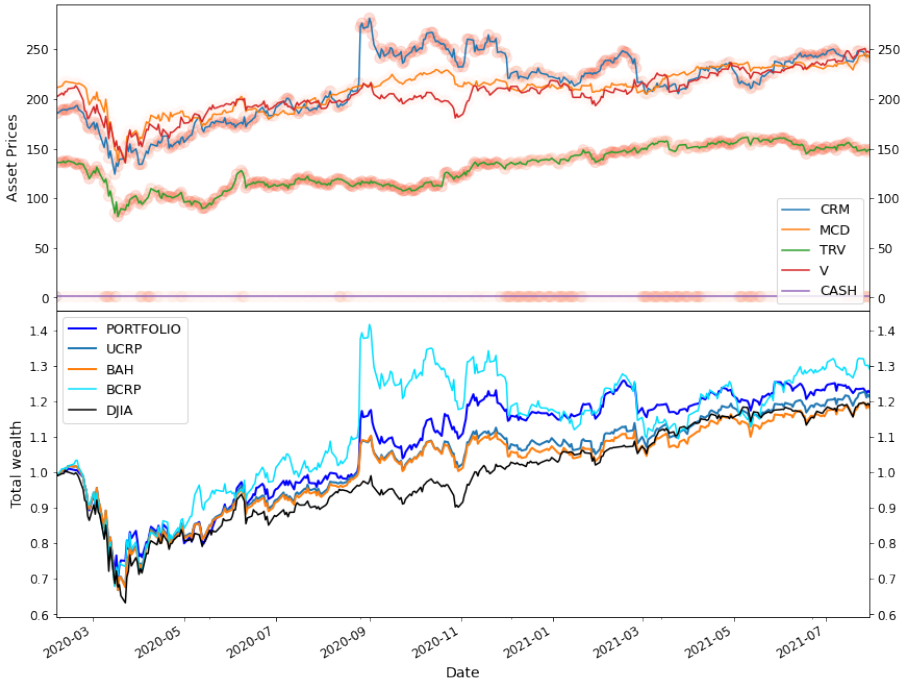

Figure C.18: Best DDPG on CRM, MCD, TRV, V

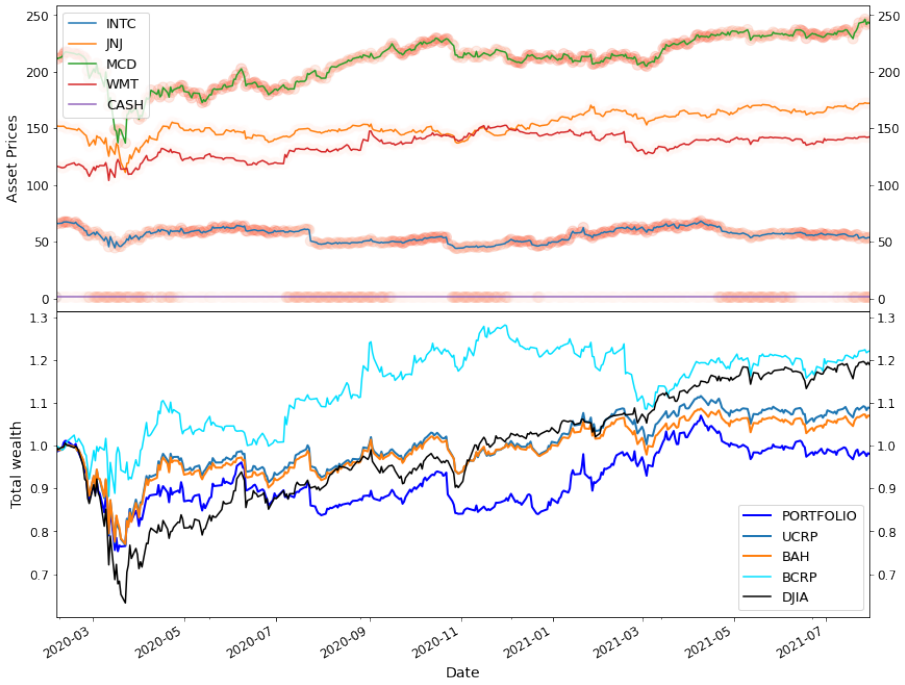

Figure C.19: Best DDPG on INTC, JNJ, MCD, WMT

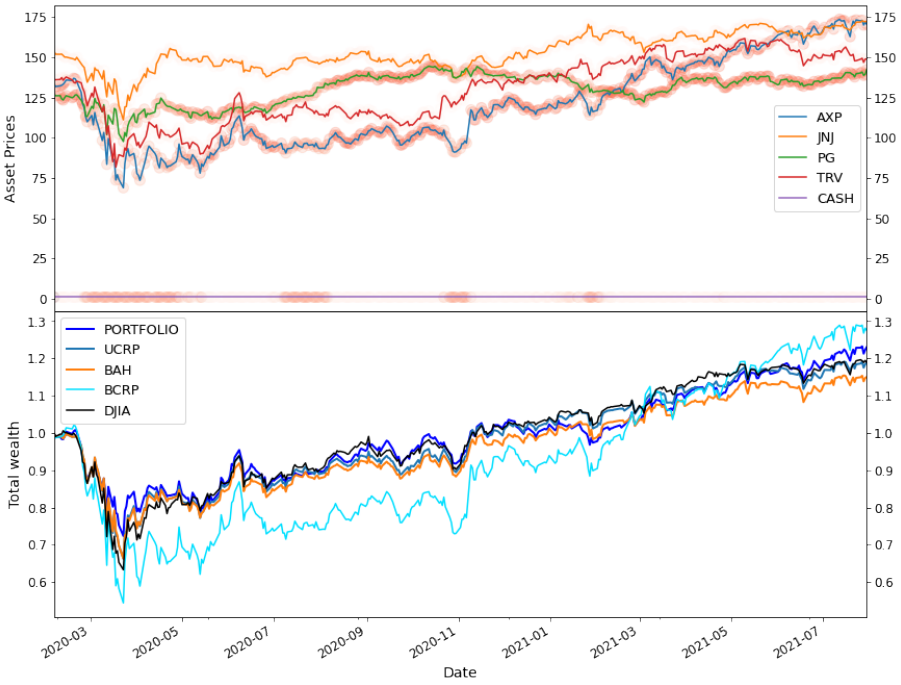

Figure C.20: Best DDPG on AXP, JNJ, PG, TRV

Appendix D

Experiments on Hyperparameter Learning Capacity

D.1 DQN

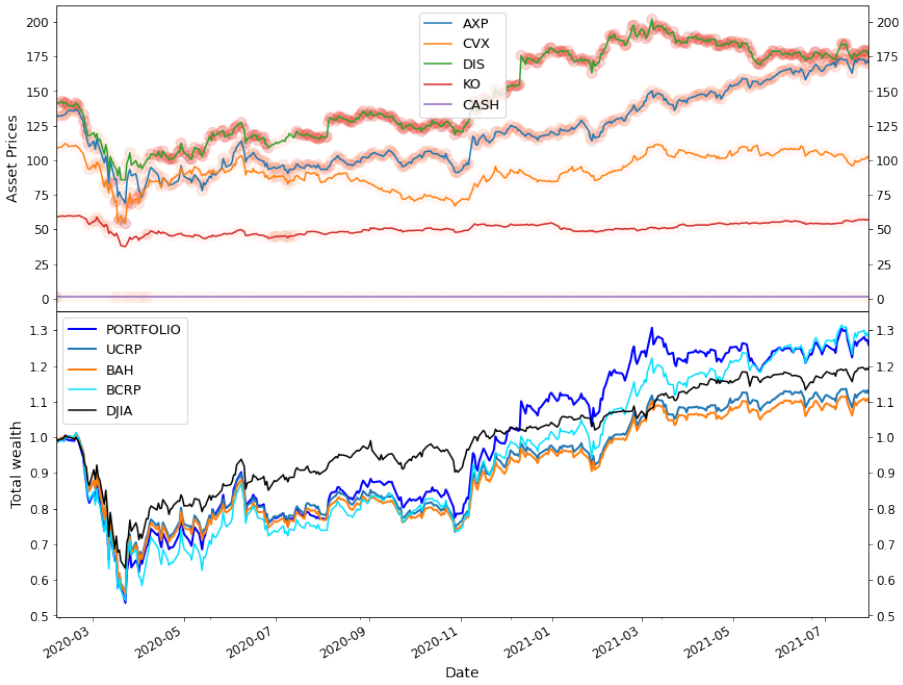

Figure D.1: DQN retrained on AXP, CVX, DIS, KO

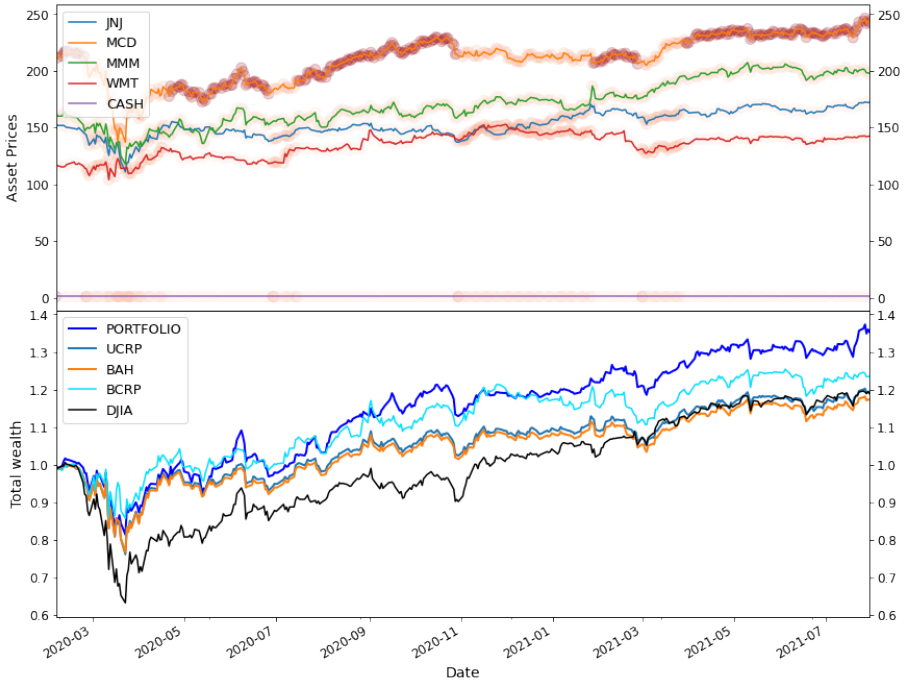

Figure D.2: DQN retrained on JNJ, MCD, MMM, WMT

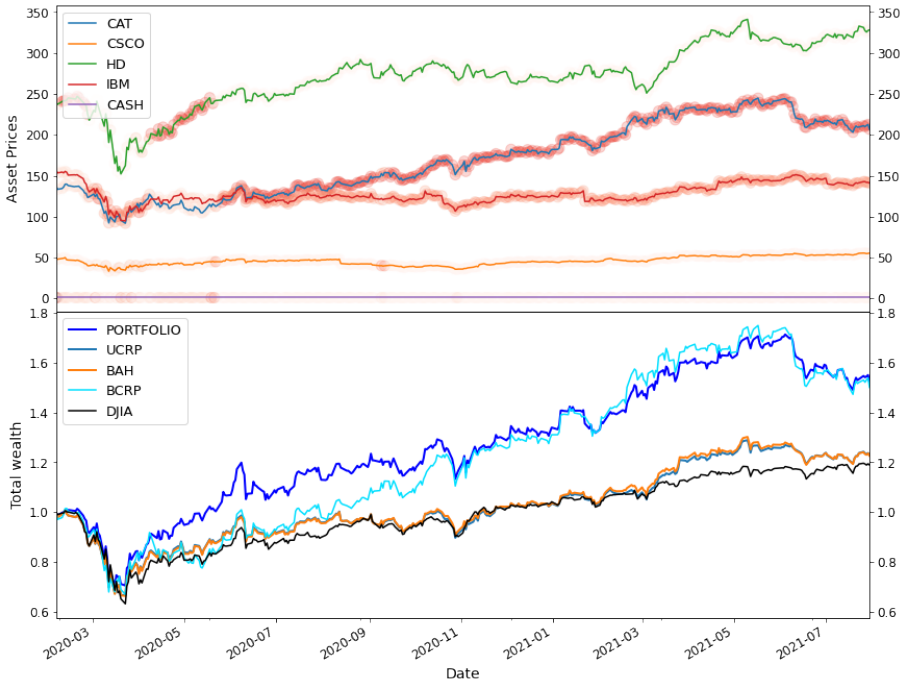

Figure D.3: DQN retrained on CAT, CSCO, HD, IBM

D.2 DDPG

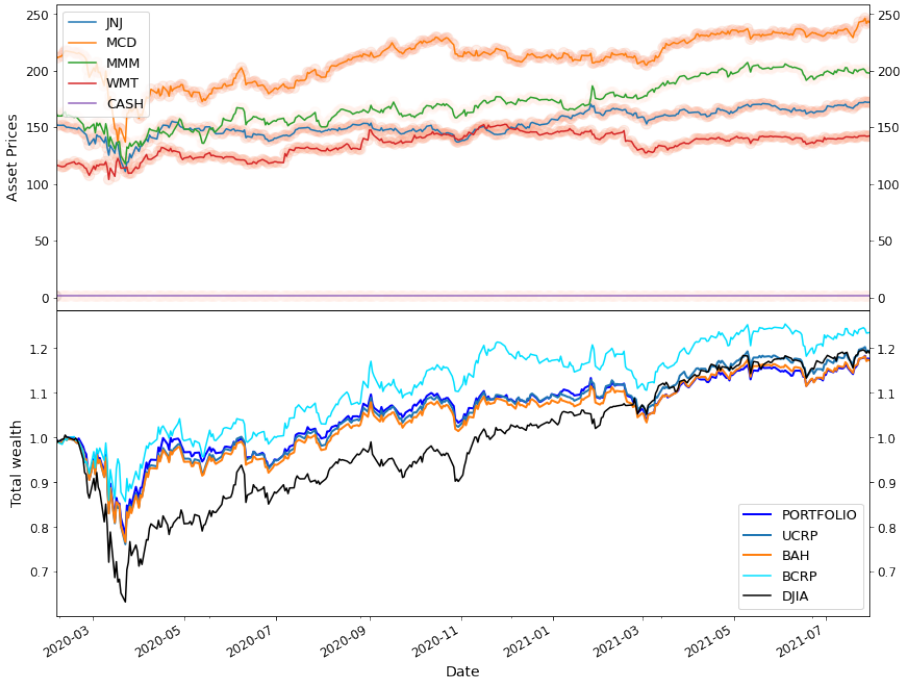

Figure D.4: DDPG retrained on JNJ, MCD, MMM, WMT

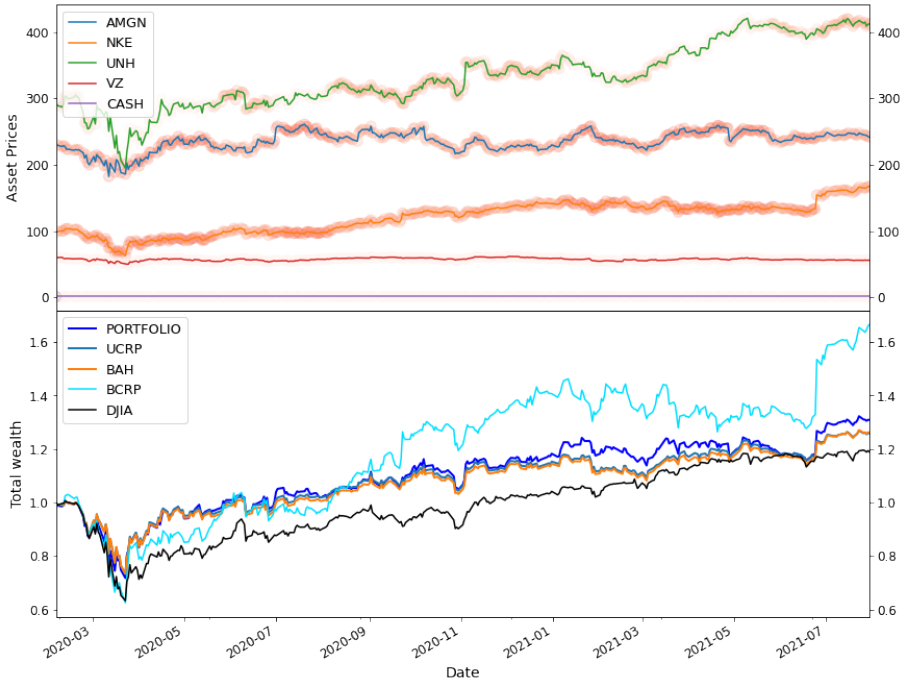

Figure D.5: DDPG retrained on AMGN, NKE, UNH, VZ

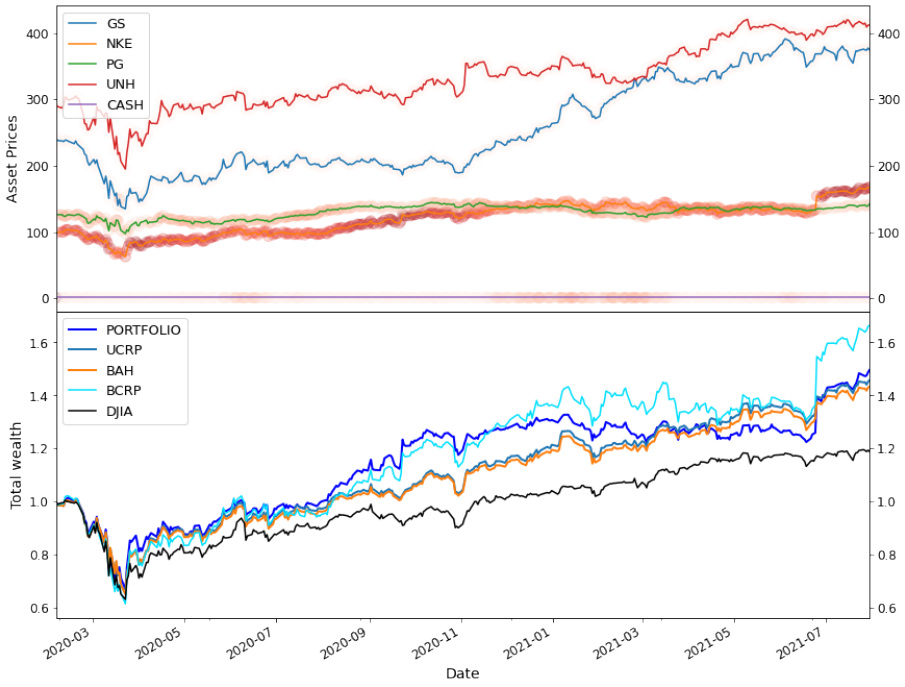

Figure D.6: DDPG retrained on GS, NKE, PG, UNH

Appendix E

Source Code

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

- Arman Mann (August 31, 2021)

Path	File	Description
`./`	`main.ipynb`	Jupyter notebook containing instructions and guide to training, testing and evaluating the trading agents.
`./`	`config.py`	Configurations for environment set up, model and training parameters, benchmarks and baseline strategies.
`alphaQ/`	`env.py`	Portfolio trading environment built using OpenAI Gym.
`alphaQ/`	`eval.py`	Functions to evaluate agents and facilitate comparisons between strategies.
`alphaQ/`	`utils.py`	Various utility functions used throughout the project to process data, plot results, etc.
`alphaQ/agent/`	`features.py`	Feature Extraction class for RL agent.
`alphaQ/agent/`	`utils.py`	Agent utilities to create agent interface, maintain agent performance record, display attributes, load models.
`alphaQ/agent/`	`callbacks.py`	Copy of the Evaluation Callback class from Stable-Baselines3 with a minor modification.

Table E.1: Table of contents and descriptions for source code files

Figure E.1: config.py

```python
"""Config file."""

from universal import algos
from stable_baselines3 import DQN, DDPG
from stable_baselines3.common.noise import OrnsteinUhlenbeckActionNoise, NormalActionNoise


MODELS = {
    'dqn': DQN,
    'ddpg': DDPG
}

MODEL_PATH = {
    'ddpg': 'models/DDPG_best.zip',
    'dqn': 'models/DQN_best.zip'
}

ACTION_SPACE = {
    'dqn': 'discrete',
    'ddpg': 'continuous',
}

ACTION_NOISE = {
    "normal": NormalActionNoise,
    "ornstein_uhlenbeck": OrnsteinUhlenbeckActionNoise,
}

# DJIA as of April 14, 2021
DOW_TICKERS = [
  'UNH',   # United Health
  'GS',    # Goldman Sachs
  'HD',    # Home Depot
  'MSFT',  # Microsoft
  'BA',    # Boeing
  'AMGN',  # Amgen
  'CAT',   # Caterpillar
  'MCD',   # McDonald's
  'HON',   # Honeywell
  'CRM',   # Salesforce
  'V',     # Visa (2013-09-20/2008-03-19)
  'MMM',   # 3M
  'DIS',   # Disney
  'JNJ',   # Johnson & Johnson
  'TRV',   # Travelers Companies
  'JPM',   # JPMorgan Chase
  'AXP',   # American Express
  'WMT',   # Walmart
  'PG',    # Proctor & Gamble
```

```
  'IBM',   # IBM
  'NKE',   # Nike
  'AAPL',  # Apple (2015-03-19)
  'CVX',   # Chevron Corporation
  'MRK',   # Merck & Co
  'DOW',   # Dow Inc
  'INTC',  # Intel
  'VZ',    # Verizon
  'WBA',   # Walgreens Boots Alliance
  'KO',    # Coca-Cola
  'CSCO',  # Cisco
]

##################################################
# Environment Configurations
##################################################

# TICKERS = ['AAPL', 'GE', 'JPM', 'MSFT', 'VOD', 'GS', 'MMM']
# TICKERS = ['AAPL', 'GE', 'JPM', 'MSFT']
TICKERS = ['AAPL', 'MSFT', 'V', 'JPM']
# TICKERS = ['AAPL', 'MSFT', 'V', 'JPM', 'JNJ', 'WMT']

# start date from which to begin downloading ticker data
START = '2008-03-19'
# START = '2013-09-20'

# end date for data used in experiments
END = '2021-07-31'

# number of units to divide discreted weights into
SLICES = 8

# number of days' prices used for a single observation
WINDOW_LENGTH = 50

# fee charged as commission by the broker
COMMISSION_RATE = 0.0025

##################################################
# Training Hyperparameters
##################################################

DDPG_KWARGS = dict(
    policy='MultiInputPolicy',
    learning_rate=0.000002,
    learning_starts=10000,
    buffer_size=10000,
    batch_size=64,
    tau=0.005,
    gamma=1,
```

```
    train_freq=100,
    action_noise='normal'  # 'normal' | 'ornstein_uhlenbeck' | None
)

DQN_KWARGS = dict(
    policy='MultiInputPolicy',
    learning_rate=0.00005,
    learning_starts=10000,
    buffer_size=10000,
    batch_size=64,
    tau=0.005,
    gamma=0.5,
    train_freq=1,
    target_update_interval=1000
)

TRAIN_VERBOSE_LEVEL = 2
RANDOM_SEED = 42

#####################################################
# Exploration configurations
#####################################################

DDPG_EX_PARAMS = dict(
    noise_sigma=0.2,
)

DQN_EX_PARAMS = dict(
    exploration_fraction=0.4,
    exploration_initial_eps=0.8,
    exploration_final_eps=0.01,
)

#####################################################
# Combined Model and NN Configurations
#####################################################

MODEL_PARAMS = {
    'ddpg': dict(
        net_arch=[512, 256, 128],
        multiplier=2,
        hyperparams=DDPG_KWARGS,
        exploration=DDPG_EX_PARAMS,
    ),
    'dqn': dict(
        net_arch=[256, 128, 64],
        multiplier=1,
        hyperparams=DQN_KWARGS,
        exploration=DQN_EX_PARAMS,
    )
```

```
}

##################################################
# Callback configurations
##################################################

CALLBACK_ENABLED = True
CALLBACK_START = 20000
SAVE_PATH = 'models/'
LOG_PATH = 'logs/'
CALLBACK_VERBOSE_LEVEL = 2

##################################################

# metrics to be calculated when evaluating strategies
ATTRIBUTES = [
    'total_wealth',
    'cumulative_return',
    'annualised_return',
    'sharpe',
    'max_drawdown',
    'max_drawdown_period',
    'ulcer_index',
    'profit_factor',
    'winning_pct',
    'beta',
    'alpha',
    'appraisal_ratio',
    'information_ratio',
    'annualised_volatility',
    'annual_turnover',
]

# online portfolio selection strategies for
# benchmarking/comparing agent performance
OLPS_STRATEGIES = [
    # benchmarks
    algos.BAH(),
    algos.CRP(),
    algos.BCRP(),
    # algos.DCRP(),
    algos.MPT(window=50, min_history=1, mu_estimator='historical',
    ↪  cov_estimator='empirical', q=0),  # min-variance

    # follow the winner
    algos.UP(),
    algos.EG(),

    # follow the loser
    algos.Anticor(window=WINDOW_LENGTH),
```

```
    algos.PAMR(eps=1),
    algos.OLMAR(window=WINDOW_LENGTH),
    algos.RMR(window=WINDOW_LENGTH),
    algos.CWMR(),
    algos.WMAMR(window=WINDOW_LENGTH),

    # pattern matching
    algos.BNN(k=WINDOW_LENGTH),
    algos.CORN(),

    # meta-learning
    algos.ONS(),
]
```

Figure E.2: alphaQ/env.py

```
"""Trading environment for portfolio optimisation task."""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import gym
from universal.algos import BAH, BCRP

import config
from alphaQ.agent.utils import AgentStrategy, Record
from alphaQ.utils import get_action_space, download_ticker_data


class PortfolioEnv(gym.Env):
    """Portfolio trading environment."""

    def __init__(self, **env_config):

        tickers = env_config['tickers']
        prices = env_config.get('prices')
        market_prices = env_config.get('market_prices')
        # define trading frequqncy
        trading_period = env_config.get('trading_period')
        # set window length for observations
        self.window_length = env_config.get('window_length', config.WINDOW_LENGTH)
        # set commission fee rate
        self.trading_cost = env_config.get('trading_cost', config.COMMISSION_RATE)
        # to use previous weights in observations or not
        self.observation_with_weights = env_config.get('observation_with_weights', True)
        # continuous or discrete action space
        self.action_space_type = env_config.get('action_space_type', 'discrete')
        # enable or disable rendering for environment
        self.render_enabled = env_config.get('render', True)
```

```
        self.render_mode = env_config.get('render_mode', 'train')

        # prices not provided
        if prices is None and tickers is not None:
            # fetch prices
            print('downloading data...')
            prices = download_ticker_data(tickers, **env_config)

        # resample prices based on trading period
        if not trading_period:
            self.prices = prices.copy()
        else:
            self.prices = prices.resample(trading_period).last()

        # select feature for close prices
        if 'Adj Close' in self.features:
            self.close = 'Adj Close'
        else:
            self.close = 'Close'

        # calculate relative (percentage) returns
        self.returns = self.prices[self.close].pct_change()  #
        ↪  self._prices/self._prices.shift(1) - 1

        # calculate price relative vector
        self.Y = self.returns + 1  # self._prices/self._prices.shift(1) (eq 1)
        # add cash column
        self.Y['CASH'] = 1
        self.Y['CASH'].iloc[0] = np.nan

        if self.action_space_type == 'discrete':
            # discretize and set action space
            self.action_set = get_action_space(self.n_instruments, env_config.get('slices',
            ↪  config.SLICES))
            self.action_space = gym.spaces.Discrete(len(self.action_set))
        elif self.action_space_type == 'continuous':
            self.action_space = gym.spaces.Box(0, 1, (self.n_instruments+1,),
            ↪  dtype=np.float32)

        # define observation space with or without weights
        history_space = gym.spaces.Box(-np.inf,  np.inf, (self.n_features,
        ↪  self.n_instruments, self.window_length), dtype=np.float32)
        if self.observation_with_weights:
            weight_space = gym.spaces.Box(0, 1, (self.n_instruments+1,), dtype=np.float32)
            self.observation_space = gym.spaces.Dict({'history': history_space, 'weights':
            ↪  weight_space})
        else:
            self.observation_space = history_space

        # set counters to track total steps and number of episodes
```

```
        self.step_count = 0
        self.episode_count = 0

        # create an object to register training stats for agent
        self.record = Record(columns=self.instruments,
        ↪  index=self.dates[self.window_length-1:])
        self.result = None

        # calculate market rate
        if market_prices is not None:
            self.market = BAH().run(market_prices[self.window_length-1:])
            self.market.fee = self.trading_cost

        # calculate best constant rebalanced portfolio
        self.bcrp = BCRP().run(self.prices[self.close][self.window_length-1:])
        self.bcrp.fee = self.trading_cost

    @property
    def instruments(self) -> list:
        """List of non cash asset instrument universe."""
        return self.prices.columns.unique(level=1).tolist()

    @property
    def n_instruments(self) -> int:
        """Count of portfolio assets."""
        return len(self.instruments)

    @property
    def features(self) -> list:
        """List of features used to train on."""
        return self.prices.columns.unique(level=0).tolist()

    @property
    def n_features(self) -> int:
        """Count of features."""
        return len(self.features)

    @property
    def dates(self) -> pd.DatetimeIndex:
        """Date indices for asset prices."""
        return self.prices.index

    @property
    def index(self) -> pd.Timestamp:
        """Return current index."""
        return self.dates[self.window_length + self.counter - 1]

    def _get_observation(self):
        """Build current observation to send to the agent."""
        window_data = self.prices[self.counter:self.counter + self.window_length]
```

```
129             # divide price vector by latest close price for each asset
130             obs = window_data.values.reshape(self.window_length, self.n_features,
                ↪  self.n_instruments)/window_data[self.close].iloc[-1].values
131             obs = obs.transpose(1, 2, 0)
132 
133             if self.observation_with_weights:
134                 if self.action_space_type == 'continuous':
135                     return {'history': obs, 'weights': self.action}
136                 else:
137                     return {'history': obs, 'weights': self.weights}
138             return obs
139 
140         def _get_done(self):
141             """Check if episode has ended."""
142             return self.index == self.dates[-1]
143 
144         def step(self, action):
145             """Step through environment one step at a time.
146 
147             Parameters
148             ----------
149             action: object
150                 Action provided by the agent.
151 
152             Returns
153             -------
154             observation: object
155                 Agent's observation of the current environment.
156             reward: float
157                 Amount of reward returned after previous action.
158             done: bool
159                 Whether the episode has ended, in which case further step() calls will return
    ↪  undefined results.
160             info: dict
161                 Contains auxiliary diagnostic information (helpful for debugging, and sometimes
    ↪  learning).
162             """
163             # increment counter
164             self.counter += 1
165             # increment timestep
166             self.step_count += 1
167 
168             # save action
169             self.action = action
170 
171             # action validity check
172             if not self.action_space.contains(action):
173                 raise ValueError('invalid `action` attempted: %s' % (action))
174 
175             # normalise for continuos action space
```

```
176             if self.action_space_type == 'continuous':
177                 # self.weights = np.exp(actions)/np.sum(np.exp(actions))  # softmax
                    ↪  normalisation
178                 self.weights = action/action.sum()
179             # get weights from action set for discrete actions space
180             else:
181                 self.weights = self.action_set[self.action]
182 
183             # save weights in training record
184             self.record.actions.loc[self.index] = self.weights
185 
186             # ============================================================================
187 
188             # relative return value
189             y = self.Y.loc[self.index]
190 
191             # calculate commision to change from dw to new weights - ignoring cash for
                ↪  transaction cost
192             mu = self.trading_cost * np.abs(self.dw - self.weights)[:-1].sum()  # (eq 16)
193 
194             # calculate returns (with or without transaction fee)
195             # rho = (self.returns.loc[self.index] * self.weights[:-1]).sum()  # (eq 3)
196             # rho = np.dot(y, self.weights) - 1  # (eq 3)
197             rho = (1 - mu) * np.dot(y, self.weights) - 1  # (eq 9)
198 
199             # calculate log returns (with or without transaction fee)
200             # log_returns = np.log(np.dot(y, self.weights))  # (eq 4)
201             log_returns = np.log((1 - mu) * np.dot(y, self.weights))  # (eq 10)
202 
203             # calculate portfolio value (with or without transaction fee)
204             # self.portfolio_value = self.portfolio_value * (rho + 1)  # (eq 2)
205             # self.portfolio_value = self.portfolio_value * np.dot(y, self.weights)  # (eq 2)
206             self.portfolio_value = self.portfolio_value * (1 - mu) * np.dot(y, self.weights)  #
                ↪  (eq 11)
207 
208             # calculating changed weights over the course of trading day
209             self.dw = (y * self.weights)/np.dot(y, self.weights)  # (eq 7)
210 
211             # ============================================================================
212 
213             # fetch observation values
214             observation = self._get_observation()
215 
216             # fetch termination flag
217             done = self._get_done()
218 
219             # select reward
220             reward = rho
221             # reward = log_returns
222             # reward = log_returns*1000/len(self.prices)
```

```
223
224         # generate info object for current step
225         info = {
226             "reward": reward,
227             "portfolio_value": self.portfolio_value,  # value_memory
228             "rate_of_return": rho,  # return_memory
229             "log_return": log_returns,
230             "return": y.values,
231             # "return": self._returns.loc[self.index].values,
232             # 'actions': self.weights,
233             'date': self.index,
234             'step': self.counter
235         }
236         self.infos.append(info)
237
238         # use for reward incorporating sharpe ratio
239         # reward = rho + sharpe(np.array([x['rate_of_return'] for x in self.infos])
240
241         if done:
242             self.df_info = pd.DataFrame(self.infos,
                ↪  index=self.dates[self.window_length-1:])
243
244             # if done, increment episode count and add stats to training record
245             self.episode_count += 1
246             self.record.episodes.append({'rewards': self.df_info['rate_of_return'].sum(),
                ↪  'total_wealth': self.portfolio_value})
247
248             if self.render_mode == 'train':
249                 print("EPISODE:", self.episode_count, 'Steps:', self.counter)
250
251             if self.render_enabled:
252                 self.render(self.render_mode)
253
254         return observation, reward, done, info
255
256     def reset(self) -> pd.DataFrame:
257         """Reset the environment to an initial state and returns an observation.
258
259         Returns
260         -------
261         observation: object
262             The initial observation.
263         """
264         if self.action_space_type == 'continuous':
265             self.action = np.append(np.zeros(self.n_instruments), 1)
266         else:
267             # action 0 corresponds to starting portfolio weights in discretised action
                ↪  space
268             self.action = 0
269
```

```
270         # setting initial and changed weights to 0 for non-cash assets and 1 for cash
271         self.weights = np.append(np.zeros(self.n_instruments), 1)
272         self.dw = np.append(np.zeros(self.n_instruments), 1)
273 
274         self.portfolio_value = 1.0
275         self.infos = [{'portfolio_value': 1.0, 'rate_of_return': 0.0, 'return':
            ↪  np.ones(self.n_instruments+1)}]
276 
277         # reset episode step counter
278         self.counter = 0
279         # get initial observation
280         ob = self._get_observation()
281         return ob
282 
283     def render(self, mode='train') -> None:
284         """Render the environment."""
285         # initialize figure and axes
286         fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(14, 12), sharex=True)
287         plt.subplots_adjust(wspace=0, hspace=0)
288         ax0, ax1 = axes[0], axes[1]
289 
290         # plot asset closing prices
291         close_prices = self.prices[self.close].iloc[self.window_length-1:]
292         close_prices['CASH'] = 1
293         close_prices.plot(ax=ax0)  # :self.index
294 
295         # plot asset weights as heatmap over prices
296         for asset in close_prices:
297             close_prices[asset].reset_index().plot.scatter(x='Date', y=asset,
                ↪  c=self.record.actions[asset].values, cmap=plt.cm.Reds,
298                                                            marker='o', s=100, vmin=0,
                                                               ↪  vmax=1, alpha=0.2, ax=ax0,
                                                               ↪  colorbar=False)
299 
300         # agent
301         self.result =
            ↪  AgentStrategy(self.record.actions).run(self.prices[self.close].iloc[self.window_length-1:])
            ↪  # :self.index
302         self.result.fee = self.trading_cost
303 
304         # portfolio and metrics
305         self.result.plot(assets=False, weights=False, ucrp=True, bah=True, ax=ax1,
            ↪  linewidth=2)
306         # bcrp
307         self.bcrp.plot(assets=False, weights=False, ucrp=False, bah=False,
            ↪  portfolio_label='BCRP', ax=ax1, color=(0.0, 0.8789398846597463, 1.0))
308         # best stock
309         asset_equity = self.result.asset_equity
310         best_stock = asset_equity[asset_equity.iloc[-1].idxmax()]
311         best_stock.rename('Best Stock').plot(ax=ax1).legend()
```

```
312
313         # market
314         self.market.plot(assets=False, weights=False, ucrp=False, bah=False,
            ↪  portfolio_label='DJIA', ax=ax1, color='black')
315
316         print("===========================================")
317         print("Total wealth:", round(self.result.total_wealth, 4))
318         if mode == 'test':
319             print(self.result.summary()[8:])
320             print("Final Holdings:", np.round(self.weights, 3))
321         print("===========================================")
322
323         # axes settings
324         ax0.set_ylabel('Asset Prices')
325         ax0_ = ax0.twinx()
326         ax0_.set_ylim(ax0.get_ylim())
327
328         ax1.set_xlim(self.dates[self.window_length-1], self.dates.max())
329         ax1.yaxis.tick_right()
330         # ax1.yaxis.set_label_position("right")
331         ax1_ = ax1.twinx()
332         ax1_.set_ylim(ax1.get_ylim())
333
334         plt.show()
```

Figure E.3: `alphaQ/eval.py`

```
1   """Functions for evaluation and comparison of agent performance."""
2
3   import pandas as pd
4   from universal.algos import BAH
5
6   from config import WINDOW_LENGTH, COMMISSION_RATE, ATTRIBUTES, OLPS_STRATEGIES
7
8
9   def evalu8(agent, env):
10      """Closure runner for each episode."""
11      # reset environment and fetch observation
12      obs = env.reset()
13      # completion flag
14      done = False
15
16      # until episode completes
17      while (not done):
18          # determine action
19          action, _states = agent.predict(obs, deterministic=True)
20          # environment: take action
21          new_obs, reward, done, info = env.step(action)
22          # update current observation
23          obs = new_obs
```

```python


def get_stats(result):
    """Calculate metrics from strategy result."""
    alpha, beta = result.alpha_beta()
    stats = [
        result.total_wealth,
        (result.total_wealth - 1) * 100,
        result.annualized_return * 100,
        result.sharpe,
        result.max_drawdown * 100,
        result.drawdown_period,
        result.ulcer,
        result.profit_factor,
        result.winning_pct * 100,
        beta,
        alpha,
        result.appraisal_ucrp,
        result.information,
        result.annualized_volatility,
        result.turnover,
    ]
    return stats


def evaluate_baselines(data, custom_strategies=[], market_data={}, attributes=None):
    """Evaluate and compare baseline strategies (with agent strategies)."""
    strategies = OLPS_STRATEGIES.copy()
    # determine strategy names for index
    strategy_names = [algo.__class__.__name__ for algo in strategies]

    # if not list, convert to list
    if not isinstance(custom_strategies, list):
        custom_strategies = [custom_strategies]

    # append custom strategies to benchmarks/baselines
    strategies = custom_strategies + strategies
    strategy_names = list(map(lambda x: x.name, custom_strategies)) + strategy_names

    # initialise objects for metrics and results
    df_metrics = pd.DataFrame(index=strategy_names, columns=ATTRIBUTES)
    results = pd.Series(index=strategy_names)

    # for each strategyy calculate metrics/results and add to objects
    for name, algo in zip(df_metrics.index, strategies):
        print(name, end='.')
        result = algo.run(data['Close'][WINDOW_LENGTH-1:])
        result.fee = COMMISSION_RATE

        df_metrics.loc[name] = get_stats(result)
```

```
        results.loc[name] = result

    # for each market benchmark calculate metrics and results
    for name, data in market_data.items():
        print(name, end='.')
        result = BAH().run(data[WINDOW_LENGTH-1:])
        result.fee = COMMISSION_RATE

        df_metrics.loc[name] = get_stats(result)
        results.loc[name] = result

    # filter selected attributes
    if attributes:
        df_metrics = df_metrics[attributes]
    return results, df_metrics
```

Figure E.4: alphaQ/utils.py

```
"""Utility functions used in various modules."""

from datetime import datetime
from itertools import combinations

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf


def download_ticker_data(tickers, start='2000-01-01', end=datetime.today().date(),
    columns=['Adj Close']):
    """Download price data for multiple tickers in bulk."""
    data = yf.download(tickers, start=start, end=end)
    return data[columns]


def train_test_split(data, n_train=None, train_years=10, validation_set=True):
    """Split data into train, test (and validation) sets."""
    if n_train is None:
        if train_years is None:
            raise "Either `n_train` or `train_years` is required."
        n_train = int(train_years * 252)

    if n_train > data.shape[0]:
        raise "Invalid train size selected."

    train = data[:n_train]
    test = data[n_train:]

    if validation_set:
```

```
        test_size = int((data.shape[0] - n_train)/2)
        val, test = test[:test_size], test[test_size:]

        return train, val, test

    return train, test


def get_action_space(M, N):
    """Discretize and return action space for DQN.

    Parameters
    ----------
    M: int
        Number of traded assets.
    N: int
        Number of divisions/slices allowed in
        discretized action space.

    Returns
    -------
    A: list
        list of all unique actions in the action space
        after discretization.
    """
    A = []
    NUM = M + 1 + N - 1
    seq = list(range(NUM))

    for c in combinations(seq, M):
        c = list(c)
        action = np.zeros(M+1)
        for i in range(len(c)-1):
            action[i+1] = c[i+1] - c[i] - 1

        action[0] = c[0]
        action[M] = NUM - c[M-1] - 1
        for j in range(M+1):
            action[j] = action[j]/N

        A.append(action)

    return A


def plot_episodes(env):
    """Plot episode summaries and performance on env."""
    # create dataframe with episode stats
    episodes = pd.DataFrame(env.record.episodes)

```

```
    print("mean:")
    print(episodes.mean())

    _, axes = plt.subplots(figsize=(20, 5))

    # plot cumulative reward per-episode
    episodes.rewards.plot.bar(color='orange', ax=axes)
    episodes.rewards.rolling(window=10).mean().plot(ax=axes)
    # df_episodes.total_wealth.plot(color='r', ax=axes)

    # axes settings
    plt.xticks(rotation=90)
    axes.set(title='Reward per Episode', ylabel='Reward', xlabel='Episode')
    plt.show()


def sharpe(returns, freq=252, risk_free_rate=0):
    """Calculate sharpe ratio for returns."""
    eps = np.finfo(float).eps
    return np.sqrt(freq) * np.mean(returns - risk_free_rate) / np.std(returns -
    ↪  risk_free_rate + eps)
```

Figure E.5: alphaQ/agent/features.py

```
"""Feature extractor for RL model."""

import gym
from stable_baselines3.common.type_aliases import TensorDict
from stable_baselines3.common.torch_layers import BaseFeaturesExtractor

import torch as th
from torch import nn


class FeatureExtractor(BaseFeaturesExtractor):
    """CNN based feature extractor for stock price data.

    Inspired by the CNN implementation by Jiang et al. (2017)
    "A deep reinforcement learning framework for the financial portfolio
    management problem"

    Parameters
    ----------
        observation_space: gym.spaces.Dict
            Assumes CxHxW shape (channels first).

        features_dim: int
            Number of units in the last layer.

        multiplier: int
```

```
            Multiplication factor for layer size.
    """

    def __init__(self, observation_space: gym.spaces.Dict, features_dim: int = 512,
    ↪  multiplier: int = 1):
        super(FeatureExtractor, self).__init__(observation_space, features_dim)

        # history_shape = observation_space['history'].shape
        n_input_channels = observation_space['history'].shape[0]
        # n_instruments = observation_space['history'].shape[1]

        # define convolutional layers
        self.cnn = nn.Sequential(
            nn.Conv2d(n_input_channels, 4*multiplier, kernel_size=(1, 8), stride=1,
            ↪  padding=0),
            nn.ReLU(),
            nn.Conv2d(4*multiplier, 8*multiplier, kernel_size=(1, 16), stride=1,
            ↪  padding=0),
            nn.ReLU(),
            nn.Conv2d(8*multiplier, 16*multiplier, kernel_size=(1, 28), stride=1,
            ↪  padding=0),
            nn.ReLU(),
            nn.Flatten(),
        )

    def forward(self, observations: TensorDict) -> th.Tensor:
        """Forward pass of the neural network."""
        # print(observations['history'].shape, observations['weights'].shape)
        # determine output from convolutions
        a = self.cnn(observations['history'])
        # concatenate weights to network output
        k = th.cat((a, observations['weights']), dim=1)

        return k
```

Figure E.6: alphaQ/agent/utils.py

```
"""Agent utilities."""

import numpy as np
import pandas as pd

from universal.algo import Algo
from stable_baselines3 import DQN, DDPG


class AgentStrategy(Algo):
    """Universal Portfolio interface for RL agent strategy comparison."""

    PRICE_TYPE = 'raw'
```

```
14
15      def __init__(self, actions, name='PORTFOLIO'):
16          super().__init__()
17          self.actions = actions
18          self.name = name
19
20      def weights(self, S):
21          """Return weights."""
22          return self.actions
23
24
25  class Record:
26      """Local data structure for actions and rewards records."""
27
28      def __init__(self, index, columns):
29          # add cash column to assets
30          columns = columns + ['CASH']
31          # records of actions
32          self.actions = pd.DataFrame(columns=columns, index=index, dtype=float)
33          self.actions.iloc[0] = np.zeros(len(columns))
34          self.actions.iloc[0]['CASH'] = 1.0
35          # record of episode summaries
36          self.episodes = []
37
38
39  def load_model(path):
40      """Load agent from path based on name."""
41      # get filename from path
42      name = path.strip('/').split('/')[-1]
43      # load model based on saved file name
44      if name.startswith('DQN'):
45          return 'dqn', DQN.load(path)
46      elif name.startswith('DDPG'):
47          return 'ddpg', DDPG.load(path)
48      else:
49          print(' - Could not load model. Check name...')
50          return None
51
52
53  def display_attributes(agent, obs_space=None):
54      """Display selected agent attributes."""
55      print('learning_rate:', agent.learning_rate)
56      print('gamma:', agent.gamma)
57      print('batch_size:', agent.batch_size)
58      print('buffer_size:', agent.buffer_size)
59      print('polyak_update:', agent.tau)
60
61      if agent.action_noise:
62          print('action_noise:', agent.action_noise)
63
```

```
    eps = agent.__dict__.get('exploration_initial_eps')
    if eps:
        print('epsilon_initial:', eps,
              '\tepislon_final:', agent.exploration_final_eps,
              '\tepislon_fraction:', agent.exploration_fraction)

    if 'features_extractor_class' in agent.policy_kwargs:
        x = agent.policy_kwargs['features_extractor_class'](obs_space)
        print('feature_extractor:')
        print(x.cnn)

    if 'net_arch' in agent.policy_kwargs:
        print('net_arch:', agent.policy_kwargs['net_arch'])

    print()
```

Figure E.7: alphaQ/agent/callbacks.py

```
"""Copy of the class stable_baselines3.common.callback.EvalCallback.
https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/callbacks.py

This class is copied here with a single modification to accept a user defined
step count from the config file only after which trained models will be saved.
This ensures models are saved only after sufficient training has occured, and
prevents models from early on in the training when exploration is high
and random actions selected with more frequency from setting a record best
on the validation set and preventing any further models being saved.
"""

import os
import warnings
from typing import Any, Dict, Optional, Union

import gym
import numpy as np

from stable_baselines3.common.vec_env import DummyVecEnv, VecEnv, sync_envs_normalization
from stable_baselines3.common.evaluation import evaluate_policy
from stable_baselines3.common.callbacks import BaseCallback, EventCallback

from config import CALLBACK_START


class EvalCallback(EventCallback):
    """
    Callback for evaluating an agent.
    .. warning::
      When using multiple environments, each call to  ``env.step()``
      will effectively correspond to ``n_envs`` steps.
      To account for that, you can use ``eval_freq = max(eval_freq // n_envs, 1)``
```

```
        :param eval_env: The environment used for initialization
        :param callback_on_new_best: Callback to trigger
            when there is a new best model according to the ``mean_reward``
        :param n_eval_episodes: The number of episodes to test the agent
        :param eval_freq: Evaluate the agent every ``eval_freq`` call of the callback.
        :param log_path: Path to a folder where the evaluations (``evaluations.npz``)
            will be saved. It will be updated at each evaluation.
        :param best_model_save_path: Path to a folder where the best model
            according to performance on the eval env will be saved.
        :param deterministic: Whether the evaluation should
            use a stochastic or deterministic actions.
        :param render: Whether to render or not the environment during evaluation
        :param verbose:
        :param warn: Passed to ``evaluate_policy`` (warns if ``eval_env`` has not been
            wrapped with a Monitor wrapper)
        """

        def __init__(
            self,
            eval_env: Union[gym.Env, VecEnv],
            callback_on_new_best: Optional[BaseCallback] = None,
            n_eval_episodes: int = 5,
            eval_freq: int = 10000,
            log_path: str = None,
            best_model_save_path: str = None,
            deterministic: bool = True,
            render: bool = False,
            verbose: int = 1,
            warn: bool = True,
        ):
            super(EvalCallback, self).__init__(callback_on_new_best, verbose=verbose)
            self.n_eval_episodes = n_eval_episodes
            self.eval_freq = eval_freq
            self.best_mean_reward = -np.inf
            self.last_mean_reward = -np.inf
            self.deterministic = deterministic
            self.render = render
            self.warn = warn

            # Convert to VecEnv for consistency
            if not isinstance(eval_env, VecEnv):
                eval_env = DummyVecEnv([lambda: eval_env])

            self.eval_env = eval_env
            self.best_model_save_path = best_model_save_path
            # Logs will be written in ``evaluations.npz``
            if log_path is not None:
                log_path = os.path.join(log_path, "evaluations")
            self.log_path = log_path
            self.evaluations_results = []
```

```
        self.evaluations_timesteps = []
        self.evaluations_length = []
        # For computing success rate
        self._is_success_buffer = []
        self.evaluations_successes = []

    def _init_callback(self) -> None:
        # Does not work in some corner cases, where the wrapper is not the same
        if not isinstance(self.training_env, type(self.eval_env)):
            warnings.warn("Training and eval env are not of the same type"
            ↪  f"{self.training_env} != {self.eval_env}")

        # Create folders if needed
        if self.best_model_save_path is not None:
            os.makedirs(self.best_model_save_path, exist_ok=True)
        if self.log_path is not None:
            os.makedirs(os.path.dirname(self.log_path), exist_ok=True)

    def _log_success_callback(self, locals_: Dict[str, Any], globals_: Dict[str, Any]) ->
    ↪  None:
        """
        Callback passed to the  ``evaluate_policy`` function
        in order to log the success rate (when applicable),
        for instance when using HER.
        :param locals_:
        :param globals_:
        """
        info = locals_["info"]

        if locals_["done"]:
            maybe_is_success = info.get("is_success")
            if maybe_is_success is not None:
                self._is_success_buffer.append(maybe_is_success)

    def _on_step(self) -> bool:

        if self.n_calls < CALLBACK_START:
            # in case the callback threshold has not been reached,
            # return without saving model
            return True

        if self.eval_freq > 0 and self.n_calls % self.eval_freq == 0:
            # Sync training and eval env if there is VecNormalize
            sync_envs_normalization(self.training_env, self.eval_env)

            # Reset success rate buffer
            self._is_success_buffer = []

            episode_rewards, episode_lengths = evaluate_policy(
                self.model,
```

```
                self.eval_env,
                n_eval_episodes=self.n_eval_episodes,
                render=self.render,
                deterministic=self.deterministic,
                return_episode_rewards=True,
                warn=self.warn,
                callback=self._log_success_callback,
            )

            if self.log_path is not None:
                self.evaluations_timesteps.append(self.num_timesteps)
                self.evaluations_results.append(episode_rewards)
                self.evaluations_length.append(episode_lengths)

                kwargs = {}
                # Save success log if present
                if len(self._is_success_buffer) > 0:
                    self.evaluations_successes.append(self._is_success_buffer)
                    kwargs = dict(successes=self.evaluations_successes)

                np.savez(
                    self.log_path,
                    timesteps=self.evaluations_timesteps,
                    results=self.evaluations_results,
                    ep_lengths=self.evaluations_length,
                    **kwargs,
                )

            mean_reward, std_reward = np.mean(episode_rewards), np.std(episode_rewards)
            mean_ep_length, std_ep_length = np.mean(episode_lengths),
            ↪  np.std(episode_lengths)
            self.last_mean_reward = mean_reward

            if self.verbose > 0:
                print(f"Eval num_timesteps={self.num_timesteps}, "
                ↪  f"episode_reward={mean_reward:.2f} +/- {std_reward:.2f}")
                print(f"Episode length: {mean_ep_length:.2f} +/- {std_ep_length:.2f}")
            # Add to current Logger
            self.logger.record("eval/mean_reward", float(mean_reward))
            self.logger.record("eval/mean_ep_length", mean_ep_length)

            if len(self._is_success_buffer) > 0:
                success_rate = np.mean(self._is_success_buffer)
                if self.verbose > 0:
                    print(f"Success rate: {100 * success_rate:.2f}%")
                self.logger.record("eval/success_rate", success_rate)

            # Dump log so the evaluation results are printed with the correct timestep
            self.logger.record("time/total timesteps", self.num_timesteps,
            ↪  exclude="tensorboard")
```

```
178             self.logger.dump(self.num_timesteps)
179
180             if mean_reward > self.best_mean_reward:
181                 if self.verbose > 0:
182                     print("New best mean reward!")
183                 if self.best_model_save_path is not None:
184                     self.model.save(os.path.join(self.best_model_save_path, "best_model"))
185                 self.best_mean_reward = mean_reward
186                 # Trigger callback if needed
187                 if self.callback is not None:
188                     return self._on_event()
189
190         return True
191
192     def update_child_locals(self, locals_: Dict[str, Any]) -> None:
193         """
194         Update the references to the local variables.
195         :param locals_: the local variables during rollout collection
196         """
197         if self.callback:
198             self.callback.update_locals(locals_)
```