POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

Exploration Techniques for a Deep Reinforcement Learning Trading Agent

Supervisors

Prof. LUCA CAGLIERO

Dr. JACOPO FIOR

Candidate

ANDREA ZAPPAVIGNA

Academic Year 2021 - 2022

Abstract

Nowadays nearly all financial orders across the globe are performed by automated trading systems. In particular, machine learning and deep learning have been utilised to try and predict the trend of different financial securities. Financial market analysis has always been critical and, in recent years, more and more opportunities are available to extract new information from all kinds of financial data. In the past, reinforcement learning has been used to solve various financial problems, including portfolio optimisation, securities trading and risk management. Among those, stock trading is considered the most complex application of machine learning in quantitative finance, primarily due to the inherent stochastic behaviour of financial markets.

In this thesis, we investigate the usage of deep reinforcement learning techniques to forecast financial time series and perform automated stock trading. The objective is to automatically generate a profitable and reasonably safe trading strategy in any financial market. Furthermore, we propose extensions of a deep reinforcement learning algorithm to automatically trade financial securities, with a particular focus on multiple stock trading. The resulting agent must consistently outperform market indexes and generate positive returns while exposing itself to minimal risk.

Our approach is compared to existing state-of-the-art algorithms, namely FinRL's ensemble agent. The key difference of our approach is to enhance a single agent to deal with financial markets in their entirety, instead of leveraging multiple algorithms to avoid possible pitfalls. We also discuss how a reinforcement learning agent can learn to trade in highly volatile markets, plus methods to build a robust and stable algorithm across different markets.

Table of Contents

List of Tables III

List of Figures IV

Acronyms V

1 Introduction 1

2 Background 3

2.1 Machine Learning . . . 3

2.1.1 Deep Learning . . . 4

2.2 Reinforcement Learning . . . 6

2.2.1 Problem Setting . . . 7

2.2.2 Fundamental Components . . . 9

2.2.3 Categorizing Reinforcement Learning Agents . . . 13

2.3 Deep Q-Learning . . . 14

2.3.1 Action Branching Architectures . . . 16

2.3.2 Distributional Reinforcement Learning . . . 17

2.3.3 Exploration Methods . . . 18

2.4 Related Works . . . 18

3 Methodology 21

3.1 Model for stock trading . . . 21

3.2 Network architecture . . . 23

4 Experimental Results 28
4.1 Data Summary 28
4.2 Experimental Setting 30
4.3 Stock Trading 31
4.4 Cryptocurrencies Trading 37
4.5 Ensemble Comparison 39

5 Conclusions and future works 44

Appendix 45

Bibliography 50

List of Tables

4.1 Test portfolios 29
4.2 Summary of parameters 30
4.3 Performance evaluation comparison 33
4.4 Comparison of yearly returns 33
4.5 Cryptocurrencies performance evaluation comparison 38
4.6 Performance evaluation vs ENS 40
4.7 Yearly returns vs ENS 40

List of Figures

2.1 The agent–environment interaction in a Markov decision process . . 7
2.2 Backup diagrams for v_π and q_π 10
2.3 Backup diagrams for v_* and q_* 12

3.1 Action branching network architecture 24
3.2 Graphical representation of noisy layer 26

4.1 Cumulative returns comparison 32
4.2 Volatile market comparison 34
4.3 Monthly returns vs DDPG 35
4.4 Distribution of returns vs DDPG 36
4.5 Cumulative returns comparison 40
4.6 Monthly returns vs ENS 41
4.7 Distribution of returns vs ENS 42

A.1 TDQN vs DDPG tearsheet 49

Acronyms

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
RL	Reinforcement Learning
MDP	Markov Decision Process
DQN	Deep Q-Network
BDQN	Branching Deep Q-Network
QR-DQN	Quantile Regression Deep Q-Network
TDQN	Trading Deep Q-Network

Chapter 1

Introduction

Nowadays nearly all financial orders across the globe are performed by automated trading systems. In particular, machine learning and deep learning have been utilised to try and predict the trend of different financial securities. Financial market analysis has always been critical and, in recent years, more and more opportunities are available to extract new information from all kinds of financial data. In the past, reinforcement learning has been used to solve various financial problems, including portfolio optimisation, securities trading and risk management. Among those, stock trading is considered the most complex application of machine learning in quantitative finance, primarily due to the inherent stochastic behaviour of financial markets. Additionally, one of the primary challenges is how to deal with the intrinsic noisy component of financial time series data. Such data is also sequential (time-dependent), so we need to develop an algorithm that can dynamically predict asset behaviour and optimise itself with each additional timestep.

In this thesis, we argue in favour of deep reinforcement learning as a practical approach to time series forecasting and automated stock trading. The objective is to automatically generate a profitable and reasonably safe trading strategy in any financial market. Furthermore, we propose extensions of a deep reinforcement learning algorithm to automatically trade financial securities, which can consistently outperform market indexes and generate positive returns while exposing itself to minimal risk. We also discuss how a reinforcement learning agent can learn to trade in highly volatile markets, plus methods to build a robust and stable algorithm across different markets.

We define an artificial environment to model the stock trading problem as in real-world exchanges. The agent and the environment continually interact in a time-dependent system. Also, the agent should learn how to trade directly by trial-and-error and its experiences of the environment. Then we develop an advanced DQN-based agent that is able to outperform other reinforcement learning algorithms in the stock trading task. Most importantly, our agent must consistently

and reliably exceed the risk-adjusted returns of competing algorithms.

To achieve a realistic multi-asset trading strategy, this algorithm must be able to overcome a few challenging aspects. First, we address how changes to the network architecture allow us to solve the dimensionality problem of the agent's action space. Precisely, how independent action branches can efficiently compute values for each stock in the reference portfolio. Second, we shift the focus of our algorithm to learning the full value distribution through quantile regression. This distributional approach allows our agent to better grasp the intrinsic randomness of the stock market and, in some way, introduces a risk-aware component to our algorithm's behaviour. Practically speaking, this makes learning more stable and reliable even in unstable markets. Third, we discuss how to efficiently explore the initially unknown environment, and thus, gain more reward. To achieve this, we introduce noise perturbations to the network's weights. Also, the parameters of the perturbations are tuned by the network itself, which means that the agent autonomously finds the optimal degree of exploration for each input dimension. The resulting agent can consistently find the top-performing assets and, consequently, achieve the best returns over the long run.

To validate our approach in the stock trading task, we benchmark our agent against traditional trading strategies and state-of-the-art reinforcement learning algorithms. In order to verify the performance of our deep reinforcement learning agent we conduct backtests with a portfolio composed of 30 US-based stocks. The proposed TDQN algorithm can outperform competing algorithms and traditional financial strategies in the stock market. Importantly, these results are obtained in an extended test window, where the underlying market exhibits both upward and downward trends.

Concretely speaking our agent's strategy exceeds the reference index by 18% and 30% when comparing cumulative and risk-adjusted returns respectively. Additionally, we can consistently achieve 12% and 14% improvement over the top reinforcement learning algorithms using the same metrics. Lastly, we also experiment with cryptocurrency trading. In this highly-volatile market and with somewhat limited data, our algorithm obtains a 43% improvement in risk-adjusted returns over the equal-weighted buy-and-hold strategy when trading with a portfolio of 8 crypto tokens.

Chapter 2

Background

2.1 Machine Learning and Artificial Intelligence

Machine Learning (ML) is a discipline of computer science that focuses on developing algorithms that can learn from data and improve automatically through experience.[1] Traditionally speaking, programming an algorithm means explicitly defining how to solve a problem step-by-step. However, it tends to be challenging for humans to build the needed algorithms for more advanced tasks manually.

In practice, it can end up being more productive to help the machine create its own model of the problem, rather than having programmers specify each required step.[2] Nowadays, ML algorithms are employed in an assortment of fields and applications, such as speech recognition and computer vision, where it is troublesome or unfeasible to develop regular algorithms to solve the needed tasks.

As we now know, machine learning methods can autonomously detect patterns in available data and use them to solve complex tasks. Given the vast array of different applications, multiple kinds of ML techniques can be considered. In computer science literature, machine learning algorithms are traditionally divided into two broad categories:

- *Supervised learning* is the task of learning a function that maps an input to an output based on already labeled training data.

- *Unsupervised learning* is the task of drawing inferences from datasets consisting of input data without labeled responses.

Supervised learning algorithms learn a function that can be utilised to predict the output related to new information: the mathematical model infers a function from a set of training examples, allowing the algorithm to accurately decide the results for new inputs that were not part of the training data.[3]

The algorithms in question must learn to generalise from the training data to new unseen instances to solve this problem. The most well-known applications of supervised learning are *classification* and *regression* algorithms. Classification algorithms are utilised when the results are confined to a limited set of values, and regression algorithms are used when the results might have any numerical value. Since, ultimately, the focus is making predictions using data, these algorithms are closely related to statistics.

In contrast to previous examples where an external supervisor tags the input data, unsupervised learning algorithms can learn patterns from data that has not been labelled, classified or categorised. Instead, the models can identify common features in the input data and respond based on the presence or absence of such features in each new piece of information.

Unsupervised learning is often used as a preprocessing step as in feature learning, but it can also be used by itself. *Clustering* algorithms are the most common among the unsupervised learning ones, where clustering means splitting a set of observations into subsets so that data points within a cluster (grouping of objects) are similar according to one or more criteria. Data mining and association rule learning are connected fields of study, focusing on data analysis to discover hidden patterns and relationships in the data.

For both supervised and unsupervised ML tasks the idea of function approximators is at the core of the solution. After all, performing machine learning involves creating a mathematical model to make predictions. Most of the function approximators that are employed nowadays were initially developed and perfected in statistics. The most common types of models are decision trees,[4, 5] support-vector machines (SVMs),[6] linear and logistic regression.[7, 8]

Despite the transformative results obtained in certain fields, ML programs can still fail to deliver expected results. One major limitation of traditional statistical algorithms is difficulty in learning from high-dimensional data such as time series, images and videos. In recent years, mainly because of substantial advancements in deep learning, machine learning has gone through sensational improvements with this kind of information.

2.1.1 Deep Learning

Machine learning as we know it today grew out of the quest for artificial intelligence (AI). At the very beginning, when academic researchers were first interested in having machines learn from data, they approached the problem by developing computing systems to mimic the biological structures at the core of the human brain.

An *Artificial Neural Network* is an interconnected group of nodes, which loosely model the neurons in a human brain. Each connection can propagate a signal to other neurons, like the synapses in a biological brain.

The receiving neuron can process it and then transmit it to its downstream neurons. Nodes are linked to each other in various patterns. Nodes may also have a weight that changes during learning, which can increment (or decrement) the strength of the signal that is propagated through the network. Thus, the neural network forms a directed, weighted graph.

Network layers can be fully connected, with every neuron in one layer connecting to every neuron in the next layer. Alternatively, they can be pooling layers, where a group of neurons in one layer connect to a single neuron in the next layer. Neurons with only such connections are known as feedforward networks. On the other hand, networks that allow connections between neurons in the same or previous layers are known as recurrent neural networks.

Neural networks learn by processing sample observations, each containing a known input and result, forming associations between the two. This mapping is then stored within the weight of the network itself. Neural network training is usually conducted by minimising the error, which given an input example measures the difference between the network output and the target result.

While neural networks training is quite a complex topic, most of it can be explained as a mathematical optimisation process, where the goal is to minimise the error value or the cost function (sometimes also referred to as loss function).Subsequent weights adjustments will produce increasingly similar results to the target output. In order to adjust the network weights, we use backpropagation to fine-tune for each error found during learning. Practically, this means that the error amount is effectively divided among the upstream connections.

Deep learning (DL) is a subfield of machine learning methods based on artificial neural networks. Typically, neurons nodes are aggregated into multiple layers, hence the name deep neural networks. Signals travel from the input to the output layer, possibly after traversing the layers multiple times. Deep learning algorithms use such layers to extract higher-level features from the raw input progressively.

For example, in a computer vision application, the raw input is usually just a matrix of pixels; the first representational layers may abstract the pixels and encode edges; the intermediate layers may compose shapes; while higher layers may encode digits or letters. Notably, a deep learning process can learn such features on its own. Furthermore, different deep networks may perform different transformations on their inputs. For instance, varying layers and layer sizes can provide different degrees of abstraction.

Deep learning architectures such as recurrent neural networks and convolutional neural networks have been applied to several fields, including computer vision, speech recognition, natural language processing and many more.

Most importantly, DL methods eliminate the need for feature engineering by translating the data into a more compact intermediate representation and possibly deriving internal structures that remove redundancy in representation. This feature is crucial for both supervised and unsupervised learning tasks, where deep neural networks are often employed as components of larger network architectures.

2.2 Reinforcement Learning

Alan Turing is widely considered the father of computer science and artificial intelligence. Part of his legacy includes the Turing test, originally called the *imitation game*, which tests a machine's ability to exhibit intelligent behaviour indistinguishable from that of a human.

Nowadays, AI researchers agree that artificial intelligence is not, by definition, a simulation of human intelligence.[9] As a matter of fact, the intelligent agent paradigm defines intelligent behaviour in general, without reference to human beings:

> An *intelligent agent* is anything that perceives its environment and autonomously takes actions that maximize its chances of success.[10]

Modern textbooks define artificial intelligence as the study and design of intelligent agents, a definition that considers goal-directed behaviour to be the essence of intelligence. Most importantly, researchers can directly compare different approaches by asking which agent best maximises a given *goal function* using this definition.

Inspired by behavioural psychology, reinforcement learning (RL) studies how intelligent agents ought to take actions in an unknown environment in order to maximise reward. When we think about how we learn, the first idea that comes to mind is that humans learn by doing. Reinforcement learning is nothing but a computational approach to learning via interaction.[11]

Formally speaking, Reinforcement learning is learning what to do—how to map situations to actions—to maximise a numerical reward signal. The rational agent is not told which actions to take but instead must discover which actions yield the most reward by trying them. In most cases, actions affect both the immediate situation and the next one, so each action has an impact on all future rewards. These characteristics, trial-and-error search, sequential decision-making and delayed feedback, are the distinguishing features of reinforcement learning.

RL differs from supervised learning, as there is no external supervisor, only a reward signal. A rational agent must be able to learn from its own experience, even when it is dealing with an unknown environment.

RL is also different from unsupervised learning: while uncovering a hidden structure in an agent's experience can undoubtedly be helpful, by itself, it does not address the reinforcement learning problem of maximising a reward signal.

Moreover, reinforcement learning is different from other machine learning paradigms because the system is *dynamic*: the RL agent does not require complete knowledge of the environment, it only needs to interact with it and collect information. In practice, this means that the agent's actions directly affect the subsequent data it receives.

To summarise, reinforcement learning is a computational approach to understand and automate goal-directed learning and decision making. It is set apart from other ML approaches by its emphasis on learning from direct interaction with an unknown environment. For this reason, modern textbooks consider reinforcement learning to be a third machine learning paradigm, alongside supervised and unsupervised learning.

2.2.1 Problem Setting

We formalise the reinforcement learning problem as the optimal control of stochastic Markov decision processes (MDPs). As put it by Sutton and Barto, the basic idea is simply to capture the key components of the following problem: a learning agent interacts with its environment over time to achieve a goal. Therefore, the learner must be able to perceive the state of its environment to some extent and must be able to take actions that affect the state.[11]

The agent and the environment continually interact in a time-dependent system: the agent selects actions, while the environment responds to these actions and presents new situations. The environment also gives back numerical values that the agent seeks to maximise over time through its choice of actions, these values are called rewards. More specifically, the reward signal defines the goal of a reinforcement learning problem.

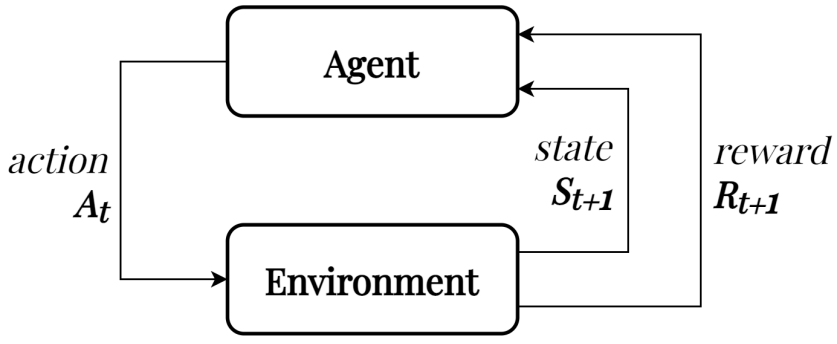


Figure 2.1: The agent–environment interaction in a Markov decision process

Concretely, at each time step t, the agent receives some representation of the environment's state S_t, and on that basis, selects an action A_t. Then, one timestep later, partly due to its action, the agent receives a numerical reward R_{t+1}, and finds itself in a new state S_{t+1} as shown in Figure 2.1.

Informally, we can think of the state as a representation of *"how the environment is"* at a particular time. Therefore, let us define the agent's internal representation of the environment as state. It is the information used by the agent to pick the following action, and for this reason, it is also called *information state*.

Reinforcement learning uses the formal framework of Markov decision processes to simply represent the interaction between a goal-directed learning agent and its environment in terms of states, actions, and rewards.

In a finite MDP, the random variables R_t and S_t have well defined probability distributions dependent only on the preceding state S_{t-1} and action A_{t-1}. That is, for particular values of these random variables there exists a probability of such values occurring at time t, given particular values of the prior state and action:

$$p\left(s', r \mid s, a\right) \;=\; \mathbb{P}\left(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right) \tag{2.1}$$

In a MDP, the probabilities given by p completely specify the environment's dynamics. Particularly, the probability of each possible value for R_t and S_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}. If the information state includes all information about past agent–environment interactions, then the state is said to have the Markov property.

MDPs can seem a bit abstract, but for the moment, it is only essential to understand that the function p can completely explain the dynamics of a RL process. Furthermore, as mentioned above, the state captures all relevant information from the history, which also means that the state is a sufficient statistic of the future.

Up to now, we informally said that the agent's goal is to maximise the cumulative reward it receives in the long run. In general, we seek to maximise the expected return, where the return G_t, is defined as a function of the reward sequence. In the simplest case, the return is the sum of the discounted rewards from timestep t:

$$G_t \;=\; \sum_k \gamma^k \cdot R_{t+k+1} \tag{2.2}$$

The discount rate γ, a parameter between 0 and 1, determines the present value of future rewards: a reward received t time steps in the future is worth only γ^{t-1} times what it would be worth if it were received immediately. In simpler terms, this expression values immediate reward above delayed return. By the above definition, we notice that returns at successive time steps are related to each other. As a consequence, we can rewrite the equation using a recursive approach as:

$$G_t \;=\; R_{t+1} + \gamma \cdot G_{t+1} \tag{2.3}$$

2.2.2 Fundamental Components

Beyond the agent and the environment, one can identify three main components of a reinforcement learning system:

- A *policy* defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. The policy is the core of a reinforcement learning agent because it is sufficient to determine behaviour, it essentially represents the agent's behaviour function.
- A *value function* is a prediction of future rewards given the current state. The value of a state is the total amount of reward an agent can expect to accumulate over the future iterations, starting from that state. In a way, the value function measures the long-term desirability of states. For this reason, the agent uses it to evaluate the goodness or badness of each state.
- A *model* mimics the behaviour of the environment, or more generally, allows us to predict how the environment will behave in the future. Models are utilized for planning a course of action by considering possible future states before they actually happen.

Methods for solving RL problems that use models and planning are called model-based methods, as opposed to more straightforward model-free methods that are explicitly trial-and-error learners. The concepts of policy and value function are critical to the reinforcement learning methods that we consider in this thesis, however we will consider only model-free methods.

Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent follows policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. In most reinforcement learning algorithms the policy is updated after every transition.

The value function of a state s under a policy π is the expected return when starting in s and following π after that. For MDPs, we call the function v_π the *state-value function* for policy π and can define it as:

$$v_\pi(s) \;=\; \mathbb{E}_\pi\left(G_t \mid S_t = s\right) \tag{2.4}$$

Intuitively speaking, the value function is supposed to represent the quality of the current state. Since the agent's goal is to maximise total reward, it will naturally prefer to be in a state with higher perceived value.

Similarly, we can define the value of taking action a in state s under a policy π as the expected return starting from s, taking the action a, and after that following policy π:

$$q_\pi(s,a) \;=\; \mathbb{E}_\pi\left(G_t \mid S_t = s, A_t = a\ \right) \tag{2.5}$$

We call q_π the *action-value function* for policy π. The value functions v_π and q_π are important because they can be estimated from experience. For example, suppose an agent follows policy π and maintains an average, for each state encountered, of the returns that have followed that state. In that case, the average will eventually converge to the state's value $v_\pi(s)$. Similarly, if different averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s, a)$.

Even though this trivial estimation works perfectly well for small MDPs, in real-world applications there are millions of states (and even more state-action pairs). Consequently, it is not practical to keep separate averages for each state individually. Instead, a better solution would be to maintain v_π and q_π as approximated functions and adjust the parameters to better match the observed returns at every step.

A fundamental property of functions used throughout reinforcement learning is that they satisfy recursive relationships similar to those introduced above in equation 2.3 for the return G_t. Thus, we can rewrite v_π as:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left(G_t \mid S_t = s\right) = \\ &= \mathbb{E}_\pi\left(R_{t+1} + \gamma \cdot G_{t+1} \mid S_t = s\right) = \\ &= \mathbb{E}_\pi\left(R_{t+1} + \gamma \cdot v_\pi(S_{t+1}) \mid S_t = s\right) \end{aligned} \tag{2.6}$$

As such, we notice that the value function v_π can be decomposed into two components, the immediate reward R_{t+1} and the discounted value of the successor state $\gamma \cdot v_\pi(S_{t+1})$. This recursive formula is known as the *Bellman expectation equation*, and it is the single most important formula of reinforcement learning. It expresses the relationship between the value of a state and the value of its successor states.

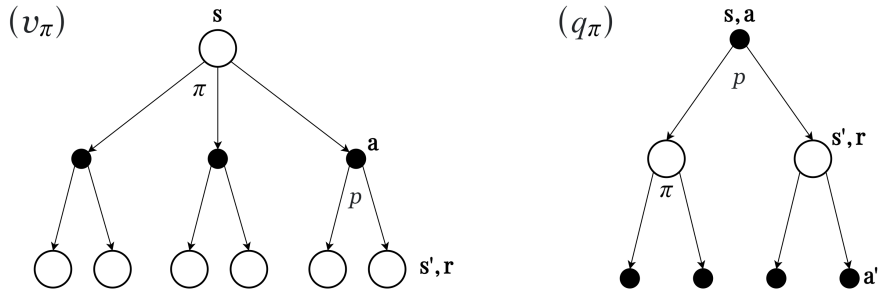


Figure 2.2: Backup diagrams for v_π and q_π

Think of looking ahead from a state to its possible successor states. Starting from state s, the root node at the top, the agent could take any action based on its policy π. From each of these actions, the environment transitions to one of several states and responds with a reward r, depending on its dynamics function p.

Finally, backup operations transfer value information back to the initial state (or a state-action pair) from its successor states. This one-step lookahead is represented in the backup diagram in Figure 2.2.

So, in a way, the Bellman equation (2.6) averages over all the possibilities of a, s' and r, weighting each by its probability of occurring. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \cdot v_\pi(s')\Big] \tag{2.7}$$

Solving a reinforcement learning task means finding a policy that accumulates a lot of reward over time. For finite MDPs, we can precisely define an optimal policy since value functions define a partial ordering over policies. This means that there is always at least one policy superior to all other options, the optimal policy π_*. Given an optimal policy, for all states s and actions a, we can write:

$$v_*(s) = \max_\pi \; v_\pi(s) \tag{2.8}$$

$$q_*(s, a) = \max_\pi \; q_\pi(s, a) \tag{2.9}$$

Combining 2.7 and 2.8, we can write the optimal value function v_* in a particular form, without reference to any specific policy. This is the *Bellman optimality equation* for v_*:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left(R_{t+1} + \gamma \cdot v_*(S_{t+1}) | S_t = s\right) = \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \cdot v_*(s')\Big] \end{aligned} \tag{2.10}$$

Intuitively, the Bellman optimality equation expresses that the value of a state under an optimal policy must equal the expected return for the best action from that state. Similarly, the Bellman optimality equation for q_* is:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left(R_{t+1} + \gamma \cdot \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right) = \\ &= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma \cdot \max_{a'} q_*(s', a')\Big] \end{aligned} \tag{2.11}$$

Once one has v_* or q_*, it is relatively easy to determine an optimal policy. For example, if we have the optimal value function, then the actions that appear best after a one-step search will be optimal actions. In other words, any greedy policy with respect to the optimal evaluation function is an optimal policy. The term greedy is used in computer science to describe any algorithm that makes choices based only on immediate benefit.

The backup diagrams in the figure below show graphically the spans of future states and actions considered in the Bellman optimality equations for v_* and q_*. These are similar to Figure 2.2 except that the maximum over the agent's choice points is taken rather than following some given policy.

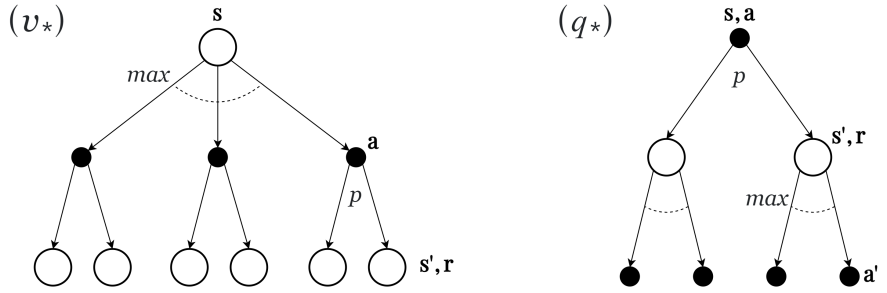


Figure 2.3: Backup diagrams for v_* and q_*

While greedy algorithms are a computationally efficient solution to many classic dynamic programming problems, they do not always work: a locally optimal decision does not always converge to the best global solution. However, for our problem, if one uses the Bellman optimality equation to evaluate the short-term consequences of actions (in our example, the one-step consequences), then a greedy policy is actually optimal in the long run. Moreover, since the formula for v_* (2.10) already considers the reward consequences of all possible future behaviour, a greedy choice is also optimal globally.

With the q_* version (2.11), the agent does not even have to do a one-step-ahead search: for any state s, it can simply find any action that maximises $q_*(s, a)$. The action-value function effectively encapsulates the results of all one-step-ahead searches. Moreover, it provides the optimal expected long-term return as an immediately available value for each state-action pair. Hence, the optimal action-value function allows our agent to select actions without knowing anything about possible successor states and their values, that is, without knowing anything about the environment's dynamics.

This latest form of the Bellman equation forms the basis to learn and make decisions for many reinforcement learning methods. However, as discussed above, even if we have a complete and accurate model of the environment's dynamics (which may not be possible in the case of limited data), it is usually not viable to compute an optimal policy by solving the Bellman optimality equation.

This is because the agent would typically be unable to fully use such a complex model because of its memory limitations. In conclusion, there are far more states than we can reasonably compute and store in most real-world applications, so approximations must be made.

2.2.3 Categorizing Reinforcement Learning Agents

As mentioned previously, computing value functions involves evaluating expectations over the whole state-space which is realistic only for the smallest (finite) MDPs. In real reinforcement learning applications, expectations are approximated by averaging over samples and also using function approximation techniques to deal with large state-action spaces.

Nowadays, many methods and algorithms are available to solve different RL tasks. Among the model-free methods, we can distinguish two main categories: *on-policy* methods, also called policy-based, and *off-policy* methods, also called value-based. Simply put, on-policy methods attempt to evaluate or improve the policy used to make decisions. In contrast, off-policy methods evaluate or improve a policy different from that which is used to select among actions.

More in detail, in policy-based algorithms, the agent learns the optimal policy and uses the same policy π_* to act. The policy used for updating and the policy used for acting is the same. In practice, in on-policy learning, the $q(s, a)$ function is updated according to action a, which we selected under our current policy $\pi(a|s)$.

The simplest implementation of on-policy learning is the *SARSA* algorithm[12], where the name represents the elements needed $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ for the action-value function update:

$$q(s, a) \leftarrow q(s, a) + \alpha \left[r + \gamma q(s', a') - q(s, a) \right] \tag{2.12}$$

On the other hand, in off-policy learning, the policy used to generate behaviour is kept unrelated to the policy that is evaluated and improved, called the target policy. In fact, the policy is implicit for many value-based algorithms as it can be derived directly from the action-value function. In this case, function $q(s, a)$ is updated according to greedy action a, the best available among all actions probed in the successor state s'.

One of the earliest examples of off-policy learning is an algorithm known as *Q-learning*. In this case, the learned action-value function directly approximates q_* independently of the policy being followed:

$$q(s, a) \leftarrow q(s, a) + \alpha \left[r + \gamma \max_{a'} q(s', a') - q(s, a) \right] \tag{2.13}$$

Learning is straightforward in off-policy-based methods because all samples are obtained according to the same logic. Therefore, we can reuse samples from previous observations without any issue. On the contrary, on-policy based methods must not use previous observations since they are not independent from each other.

For this reason, value-based methods are naturally more sample efficient as they can make use of any transition; in contrast, policy-based methods would introduce a bias when using off-policy experience.

On the flip side, value-based algorithms are not well-suited to deal with large or continuous action spaces. Both of these are non-issues with policy-based methods.

In recent years, *actor-critic* methods have been introduced to bridge the gap between the two previously mentioned methods. The core idea in the actor-critic approach is to update the *actor network* which approximates the policy probability distribution and the *critic network* that estimates the value function simultaneously. Over time, the actor learns to take better actions, and the critic evaluates those actions more precisely. The actor-critic approach can learn and adapt to large, continuous environments at the cost of increased complexity and execution time.

2.3 Deep Q-Learning

We now understand that computing value functions involve evaluating expectations over the whole state-space, which is unfeasible in actual MDPs. Therefore, in real reinforcement learning applications, function approximation techniques must be used to represent value functions over large state-action spaces.

One of the simplest and most popular options to obtain excellent results is the value-based algorithm known as *Q-learning*. The basic version of Q-learning keeps a lookup table of values $Q(s, a)$ with one entry for every state-action pair. In order to learn the optimal Q-value function, the algorithm uses the Bellman equation (2.11), whose unique solution is $Q_*(s, a)$.

Algorithm 1 Q-learning

input step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ arbitrarily
for each episode **do**
  Initialize S
  for each timestep t **do**
    Choose $A_t = a_*$ from $S_t = s$ using policy derived by Q
    $a_* \leftarrow \arg\max_a Q(s, a)$
    Take action A_t, observe $R_{t+1} = r, S_{t+1} = s'$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} q(s', a') - q(s, a)\right]$
    $s \leftarrow s'$
  end for
end for

This elementary tabular setting is often inapplicable due to the high-dimensional (and possibly continuous) state-action space. In that context, a parameterised value function $Q(s, a, \theta)$ is needed, where θ is a parameter that helps us compute an approximation of the Q-values. This primary variant is known as *fitted Q-learning*.

The input state is provided to the Q-network in neural fitted Q-learning, plus a different output is given for each possible action. This setting provides an efficient structure that has the advantage of computing the action-value update in a single forward pass in the neural network for a given s'.

The Q-values are often parameterised with a neural network $Q(s, a, \theta_k)$, where the parameters θ_k are updated by stochastic gradient descent (or a variant) by minimising the square loss. Thus, the Q-learning update amounts in updating the parameters as:

$$Y_k^Q = r + \gamma \cdot \max_{a'} q(s', a', \theta_k) \tag{2.14}$$

$$\theta_{k+1} = \theta_k + \alpha \Big[Y_k^Q - Q(s, a, \theta_k) \Big] \cdot \nabla_{\theta_k} Q(s, a, \theta_k) \tag{2.15}$$

where α is a tuning parameter called learning rate, a scalar step size that represents the speed at which the agent learns.

Despite the generalisation and extrapolation abilities of neural networks, this approach can still result in errors at different places in the state-action space. Consequently, convergence may be slow or even unstable. Another related side-effect of using function approximators is that Q-values tend to be overestimated due to the max operator. Due to the instabilities and the risk of overestimation, specific care has to be taken to ensure proper learning.

The *deep Q-network* (DQN) algorithm uses two ideas to limit such instabilities, the target network and the experience replay memory:

- The target Q-network parameters are updated only once every C iterations. This delayed update prevents the instabilities from propagating too quickly, reducing the risk of divergence as the target values Y_k^Q are kept fixed for C iterations.

- The replay memory keeps all information of the last N time steps, where the experience is collected by following an ϵ-greedy policy. The updates are then made on a set of tuples (s, a, r, s') called mini-batch, selected randomly within the replay memory.

These techniques allow for updates that cover a wide range of the action space. Consequently, these two ideas allow us to make a more extensive update of the parameters while efficiently parallelising the algorithm.

DQN can use other techniques in addition to the target Q-network and the replay memory. For example, to keep the target values on a reasonable scale and ensure proper learning in practice, rewards are often clipped between -1 and +1. Clipping the rewards limits the scale of possible errors and makes it easier to learn.

One of the major shortcomings of the Q-learning algorithm is that the max operation uses the same values both to select and evaluate an action.

This trivial operation tends to overestimate values in case of inaccuracies or noise, resulting in overoptimistic value estimates. Therefore, the DQN algorithm might induce an intrinsic bias.

Several techniques can be implemented to counter this issue, but the most popular solutions are *Double DQN* and *Dueling DQN*. Both methods keep track of multiple estimates for each variable, allowing us to find its values from unrelated information. Thus, regardless of whether errors in the estimated Q-values are due to function approximation or any other source, these methods manage to cancel out the positive bias in estimating the action values.

There are many more tips and tricks that researchers have discovered to make DQN training more stable and efficient; some of them will be highlighted in the following sections. It is important to note that even the basic DQN algorithm has achieved superhuman level control when tested on a suite of ATARI games. So while the ideas behind deep Q-networks are elementary, the performance is still class-leading among model-free methods.

2.3.1 Action Branching Architectures

As mentioned before, value-based reinforcement learning algorithms have been historically limited to domains with relatively small discrete action spaces. This limitation is due to the number of actions growing exponentially with increasing action space dimensionality. This combinatorial increase can rapidly decrease the performance of discrete-action agents, especially considering that large action spaces are already challenging to explore efficiently.

Despite this fact, DQN and its variants have been central to numerous successes of deep reinforcement learning. DQNs excellent track record can be partially explained by their intrinsic off-policy nature, which can, in principle, obtain better sample efficiency (which is of fundamental importance with smaller financial datasets) than corresponding policy gradient methods by reusing transitions from a replay memory of past experience.

Given the potential of off-policy algorithms, Tavakoli et al.[13] introduced a novel neural architecture featuring a shared decision module followed by several network branches, one for each action dimension. The shared module computes a latent representation of the input state, which is then used to evaluate the current state value. This state value is then forwarded to the action branches that independently compute the state-dependent action advantages before combining them to output the Q values for each action dimension.

This architecture achieves a linear increase in network outputs and allows traditional value-based algorithms such as Deep Q-Learning to operate effectively in domains with high-dimensional continuous action spaces.

2.3.2 Distributional Reinforcement Learning

One of the core principles of reinforcement learning is that the agent goal is to maximise cumulative reward. In off-policy agents, this is achieved by selecting the optimal action based upon the action-value function Q, which is described by Bellman's equation in terms of expected reward R and expected outcome of the current sample transition:

$$Q\left(s,a\right) = \ \mathbb{E}\ R\left(s,a\right) + \gamma\ \mathbb{E}\ Q(S',A') \tag{2.16}$$

Therefore, in a way, traditional reinforcement learning algorithms average over the aforementioned state transitions to estimate the value function. In contrast to the traditional solution, Bellamare et al.[14] argue in favour of the value distribution, a novel approach in which the distribution over returns is modeled explicitly inside the agent state instead of only estimating the mean. In practice, this new distributional approach on reinforcement learning studies the random return Z, whose expectation is the value Q:

$$Q\left(s,a\right) = \mathbb{E}\ Z(s,a) \tag{2.17}$$

Like many of the functions presented earlier this random return is also described by a recursive equation, but one of a distributional nature, where the action-value distribution Z is computed as:

$$Z(s,a) = R(s,a) + \gamma\ Z(S',A') \tag{2.18}$$

The formula above is called *distributional Bellman equation*. By combining this distributional approach with a neural network as the function approximator, we can implement risk-aware behaviour inside the RL agent itself. In fact, Dabney et al.[15] state that "the value distribution describes the intrinsic randomness of the agent's interactions with its environment, rather than some measure of uncertainty about the environment itself".

Approximating the full distribution through the distributional Bellman operator also preserves multimodality and randomness instead of only estimating the mean. The authors believe these qualities of the value distribution lead to more stable learning and increase the agent's robustness by improving its ability to propagate rarely occurring events.

The authors implement the aforementioned distributional perspective by approximating the value distribution Z, and then using quantile regression to train the network's parameters. This new algorithm can be built on top of the classic DQN architecture and is called *quantile regression DQN* (QR-DQN).

The only additional parameter of QR-DQN not shared by DQN is the number of quantiles N, which controls with what resolution we approximate the value distribution. As we increase N, the algorithm becomes increasingly able to distinguish and propagate low probability events within the environment, significantly improving robustness and performance over the baseline algorithm.

2.3.3 Exploration Methods

At its core, for model-free agents, reinforcement learning is like trial-and-error learning. We already know that the agent goal is to discover a suitable policy by learning from its experiences of the environment, so the question we now face is how to do it without losing too much reward along the way since the environment is initially unknown. This issue is known as the exploration versus exploitation tradeoff, and it is one of the most critical reinforcement learning problems.

The problem is as follows: how can an agent get as much reward as possible while learning about the environment as quickly as possible. This is a tricky problem because, in real applications, the best long-term strategy may involve short-term sacrifices, so the agent must initially gather enough information to be able to make the best overall decisions

The naïve solution to this problem is simply to add random noise to a greedy policy: the agent selects the action with the highest perceived value at each transition, but there is a non-zero chance to select a random action at each iteration. This method is known as ϵ-*greedy*, where ϵ is the probability of random exploration and usually follows a linearly decaying schedule. Unfortunately, naïve exploration methods are still used in many state-of-the-art publications.

Fortunato et al.[16] argue that such local (and state-independent) perturbations are "unlikely to lead to the large-scale behavioural patterns needed for efficient exploration in larger environments". Consequently, they propose a simple alternative approach called *NoisyNet*, where perturbations of the network weights are used to drive exploration.

The key insight is that changes of the weight vector can induce a consistent and potentially complex, state-dependent policy change over multiple time steps. The perturbations are sampled from a noise distribution, whose parameters are learned using gradients from the loss function alongside the other parameters of the agent.

2.4 Related Works

Dynamic programming applications in finance are far from a novel idea, but in recent years, mainly due to the deep learning revolution, the popularity of deep reinforcement learning (DRL) methods in economics and finance is booming.

This increase in popularity is because RL algorithms offer problem-solving techniques perfect for the dynamic control problem posed by financial markets. In fact, reinforcement learning methods offer a wide variety of capabilities for handling quite complex and realistic market environments. For example we can naturally incorporate many real-world limitations such as transaction costs and market volatility which is difficult to implement in traditional ML algorithms.

Furthermore, deep learning can be employed to help with noisy and nonlinear patterns of economic timeseries data. So by combining the modelling capabilities and computational efficiency of RL with the excellent feature extraction properties of DL we obtain the ideal, scalable solution to real-world financial problems.

There are many problems for reinforcement learning in economics and finance, but most applications can be grouped under the portfolio allocation [17, 18, 19] and stock trading [20, 21] categories. For this project we focus on automated stock trading, or more in detail, multi-asset trading for any financial securities.

Among the most recent applications of deep reinforcement learning for stock trading the vast majority operate with a critic-only approach. In particular, Deep Q-learning (DQN) and its variants are the most commonly used algorithms, given that it is relatively simple to work with and can still obtain great results despite being an older method.

Many implementations train agents on a single asset as in Jeong et al.[22], Dang[23]. This idea is, of course, unrealistic for real-world application since it would require many independent models for each security in the trader portfolio. Another limitation of DQN is that it only works with small, discrete action spaces, which is not ideal since asset prices are obviously continuous.

To solve this issue we can use actor-only or actor-critc methods instead, like in Yang et al.[24], Zhang et al.[25], Chen et al.[26]. This approach is better because by approximating the policy, which is a probability distribution, we can enhance the behaviour of our algorithm when dealing with stochastic financial markets.

Unfortunately, these methods require much more information to work optimally since they can not store and reuse previous experiences in a replay buffer. In theory this makes algorithms like DQN more sample efficient, which is absolutely essential with some of the more recent financial markets for which quality historical data is difficult to obtain for free.

Next, considering the problem setting and financial data representation, the vast majority of publications only rely on elementary data such as prices and volumes. However, some projects [27] include additional information in the form of technical indicators, which can always be computed starting from the primary data mentioned above. Additionally, some previous studies [28] also merge additional market-specific information like, for instance, the volatility index (VIX) for the stock market or even market sentiment information, extracted from external data like financial news.

For our project implementation we rely on a common framework called FinRL for deep reinforcement learning applications in quantitative finance. FinRL[29] is an extensive library containing everything from trading environments to historical market data and live-trading APIs. In addition, it provides comprehensive support for many essential operations such as backtesting and plotting through several open-source packages.

The framework allows us to focus on the algorithmic implementation while setting some common trading constraints: this standard setting will allow future projects to precisely compare our results. It also means that we can objectively compare our agent performance to previous state-of-the-art projects compatible with this framework, such as the ensemble agent[24]. The idea behind the ensemble agent is simple yet powerful: use three different actor-critic algorithms to develop a robust trading strategy. The ensemble strategy inherits and combines the best features of each algorithm. As a result, the overall trading agent is better than the sum of its parts as it can adjust to different market situations making it more reliable. The ensemble approach leverages multiple algorithms to avoid a single point of failure, as such the resulting strategy is naturally more resilient to trend reserval and market volatility in general. On the other hand, our solution is to enhance a single agent's ability to deal with financial markets in their entirety. This fundamental notion is the driving principle behind our agent, whose key components and proposed techniques can be reused and built upon in the future.

We believe that the ecosystem developed by Liu et al.[30] is a great starting point for developing reinforcement learning applications, as such we contributed to the project by writing an open leaderboard to compare different algorithms and parameter settings for each environment. Furthermore, we intend to publish our project results on the leaderboard so that community members can learn and improve their submissions even further.

Chapter 3

Methodology

As part of this thesis project, we define an environment to model the stock trading problem as in real trading exchanges. Furthermore, we propose Trading Deep Q-Network (TDQN), a DQN-based algorithm, to automatically trade financial securities within this environment. The algorithm can consistently outperform market indexes and generate positive returns while exposing itself to minimal risk.

In this chapter we explain in detail the architecture of the project, analyzing the stock market environment, the problem constraints, and the trading agent algorithm. Specifically, we discuss how our agent can learn to trade in highly volatile markets, and how to build a stable algorithm across different markets.

3.1 Model for stock trading

This project aims to develop an algorithm able to trade in a market environment with a portfolio of multiple stocks. For our experiments, an artificial `OpenAI Gym`[31] environment is employed to mimic a real-world trading exchange. The environment is provided as part of the `FinRL` framework by the AI4Finance Foundation.

We previously defined the reinforcement learning problem as the optimal control of stochastic Markov decision processes. Therefore, to model the interactive nature of the dynamic stock market environment, we develop a Markov Decision Process (MDP) as follows:

- The state space is composed of all the stock information available to our agent, including the stock prices and shares, as well as the remaining account balance.
- The action space is a vector of actions over the stocks in the portfolio. At each timestep, the allowed actions on each stock include selling, buying, or holding.

- The reward is proportional to the difference in total account value. The net worth of the account is the sum of the remaining cash balance plus the current value of the stock portfolio.

Before training a deep reinforcement trading agent, we must carefully consider what information is valuable and how can our trading agent obtain it. In practical trading, various information needs to be considered, such as the historical stock prices, current holding shares, market liquidity, and technical indicators. So, let us breakdown the components of the environment more in detail.

Excluding the remaining account balance, the state space is composed of two key parts the position state and the market features. The position state includes the current stock price and the number of shares in our possession, while the market features is a set of 8 different technical indicators for each ticket.

In financial technical analysis, an indicator is a mathematical calculation based on historical price and volume that aims to forecast financial market direction. Technical indicators are a fundamental part of technical analysis and are often employed to try to predict the market trend.

Our state-space representation makes it easy to port to other markets and captures intermarket relations like momentum, trend-reversal, and so forth. Furthermore, we only make use of public information without relying on high-quality data: financial data can be highly profitable, which is why it is not as easy to obtain beyond the US stock market.

Some assumptions still need to be made with respect to real trading exchanges, in particular, the environment assumes enough liquidity for the agent's transactions to be executed instantly at close price. Moreover, we also assume that our agent's actions will not influence the stock market behaviour. Given that the agent's initial capital is set to 1 million US$ by default, we uphold that the marked liquidity assumptions are fair.

Additionally, a transaction fee is charged for each trade to mimic the long list of transaction costs (exchange fees, execution fees, SEC fees) related to real-world stock trading. Different exchanges and brokers have different commission fees, but for our project, we set transaction cost as 0.1% of the value of each trade.

The action space is defined as $\{-k, \cdots, -1,0,1, \cdots, k\}$ for a single stock, where k is a predefined parameter that sets the maximum number of shares for each buying action. The action space is then normalized to $[-1,1]$, since for our agent it is easier to work with a normalized and symmetric distribution. In this setting, an action in absolute value closer to 1 represents a confident *Buy* or *Sell* signal, where the agent will trade a number of shares proportional to such confidence measure. On the other hand, an absolute value close to 0 represents a doubtful signal, which will be interpreted by the agent as a tentative *Buy* or *Sell*, or outright null *Hold* signal.

If the action generated is that of a *Hold* signal, then the positions owned in the previous timestamp are carried over, and no change is done to the position space. Further, if the action corresponds to a *Buy* signal, then a long contract is added to the position space, provided the number of contracts in the position space is less than the maximum number of contracts we can buy. This course of action is known as long selling. In long selling we buy assets because we believe that the value is going to increase in the future.

The *Sell* signals work in the same manner except that our agent must operate with a long-only strategy, which does not allow short selling. This portfolio setting means that all assets are bought using cash, and the value gained from selling assets is held in cash. That is, the agent cannot buy any asset without the necessary funds and can not sell any asset without holding it.

The agent can derive a stock trading strategy with lower turnover because the available trading actions for individual assets are limited to a range. The upper bound of our action space limits the issue of massive changes in portfolio weights and guarantees us lower losses from transaction costs with respect to unlimited trading strategies.

Finally, we define our reward function as the portfolio value change between consecutive timesteps. The theory is quite simple: we need to maximize the positive change of the portfolio value by buying and holding the stocks whose price will increase at the next step, and minimize the negative change of the portfolio value by selling the stocks whose price will decrease at the next time step.

Sometimes, sudden events may cause a stock market crash, such as the global pandemic in the first quarter of 2020. We employ the volatility index (VIX), which measures extreme asset price movements, to control the risk in a worst-case scenario. This index is incorporated with the reward function to address our risk-aversion for market crashes. When it is higher than a predefined threshold, which indicates extreme market conditions, we halt buying and the trading agent sells all shares, because all stock prices will fall. We resume trading once the volatility index returns under the threshold.

3.2 Network architecture

This section discusses how our agent is built from the ground up and every step needed to build a robust trading strategy. We start from a basic DQN algorithm to implement our trading agent and progressively enhance its ability to deal with the aforementioned financial environment.

As mentioned in Section 2.2.3, off-policy algorithms are almost always used in tasks with a somewhat restricted state-action space. This historical notion has already been questioned in the past and, especially with the introduction of deep

learning methods (neural networks as function approximators) within RL, proved old fashioned. Modern DQN algorithms are clearly able to deal with complex state spaces, as proven by Mnih et al.[32] when their agent was able to achieve superhuman level results when playing ATARI games starting from just pixels.

The main complication for this project is the complexity of the action space: DQN algorithms are not equipped by default with the tools needed to successfully solve a task with action spaces as widespread as ours. In general, off-policy algorithms struggle with high-dimensional action spaces because the count of actions that need to be evaluated at each timestep grows exponentially with the number of action dimensions.

Large actions spaces are handled better by learning policy directly with on-policy or actor-critic methods. In this case, the agent optimizes a parametrized policy function. This method is way more efficient from a computational standpoint, because by approximating the policy function we can avoid evaluating all actions at each timestep since the optimal action is chosen directly using the policy.

Due to this computational limit most state-of-the-art implementations of DQN operate with a minimal action space, often with only 2 or 3 input dimensions. For example, the action space size of the ATARI environment in OpenAI Gym is only 18. For our problem, the agent needs to be able to operate with an action space size in the hundreds. Specifically, given our problem setting, the DQN agent needs to be able to operate over 30 inputs, each with size $(2k + 1)$.

To deal with our task's continuous, high-dimensional action space, we employ the action branching architecture introduced in Section 2.3.1. The core idea is that it is possible to learn for each action dimension with a degree of independence. This fundamental notion is reflected in the network architecture, which includes several independent action branches as shown in Figure 3.1 below.

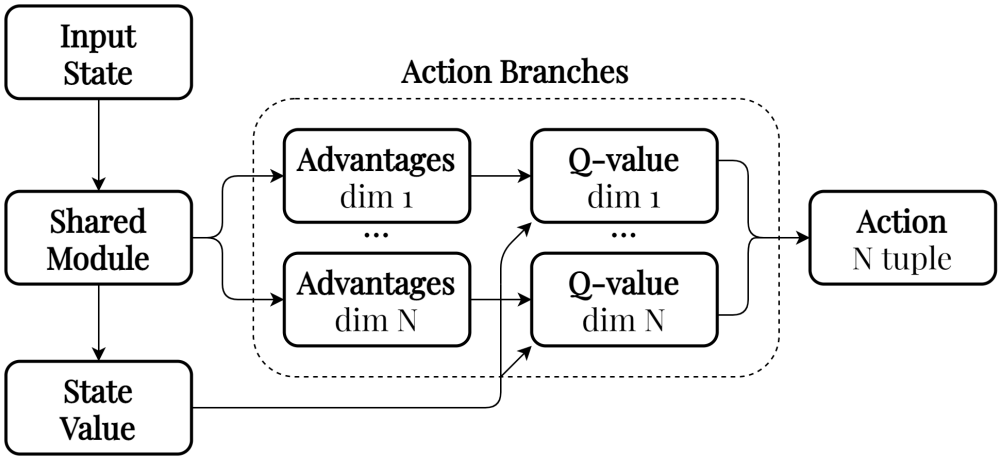


Figure 3.1: Action branching network architecture

In practice, the shared representation module computes an intermediate representation of the state input. This intermediate value is then used to evaluate the shared state value and the state-dependent action advantages for each subsequent action branch. Finally, both the state value and the action advantages are combined to output the Q values for each action dimension.

Note that even though the action branches are independent from each other, the shared representation module acts as a common denominator in evaluating the action advantages. As such, all components of the network share information with each other and even different action branches are interconnected to a degree. Intuitively one can imagine that this network architecture allows us to distribute the approximation of the value function over several action branches while maintaining a shared network module for coordination.

At this point, our branching DQN (BDQN) agent can operate within our financial environment with moderate success, but our purpose is to create a highly robust trading strategy. To achieve our ultimate goal, we need to enhance our agent's ability to learn under any circumstance. This means that we have to improve the stability of our agent.

During testing, we quickly realized that the stability metric was the top priority: our agent's performance is directly related to its stability. Even minor losses in stability can drastically alter our results in the long run. This behaviour can be partially explained by the intrinsic randomness of the stock market, which makes learning challenging under some circumstances even for our BDQN agent. Moreover, our experiments also underline that the stability metric (and thus, our agent performance) suffers from high variance across the pseudorandom seed initialization. That is, stability results vary wildly by only changing the agent or environment seed.

So our objective is to strengthen our agent's ability to learn across different market conditions, including high-turbulence periods, and independently of its initialization. To model the parametric uncertainty of our agent we turn to distributional reinforcement learning, as introduced in Section 2.3.2.

Changes with respect to the current BDQN algorithm are substantial but restricted to the algorithm's core, which allows us to combine quantile regression to our existing branching architecture. Referencing Figure 3.1 again, we can think that changes are restricted to the shared module and the state value, so action branches can still work independently as before. The resulting QR-BDQN agent is indeed more resilient to the randomness of the environment.

The reason behind this choice is that our trading agent should be responsive to different type of market trends: we cannot afford to underperform under different market behaviour. Therefore, our agent must be a 'jack-of-all-trades' and remain profitable whether the market exhibits a *bullish* (upward) trend or a *bearish* (downward) trend.

At this point, the algorithm can trade effectively even in an unstable market, trend reversal is easily spotted, and weight adjustments are quickly propagated through the network weights. One last shortcoming we wish to address is the agent's exploration within the financial environment. Analogously as before, during practical testing, we noticed a correlation between the exploration schedule of our agent and its performance. Namely, more in-depth exploration resulted in overall better performance across all metrics. It is not easy to pinpoint the exact cause of this relation. One hypothesis is that our algorithm tends to overfit the training data, resulting in underperformance if significant price changes appear in one of the stocks in the state space.

To address the exploration issue, we swap the fully connected linear layers of our QR-BDQN agent to noisy linear layers. This solution, introduced in Section 2.3.3, allows us to replace the conventional exploration heuristics (in our case, ϵ-greedy) with a totally different exploration method. Simply put, we inject the agent's network weights with gaussian noise. Having weights with greater uncertainty introduces more variability into the decisions made by the agent, which ultimately is what we need for our exploration. The stochasticity induced by noisy networks into the agent's behaviour can be used to augment its ability to explore.

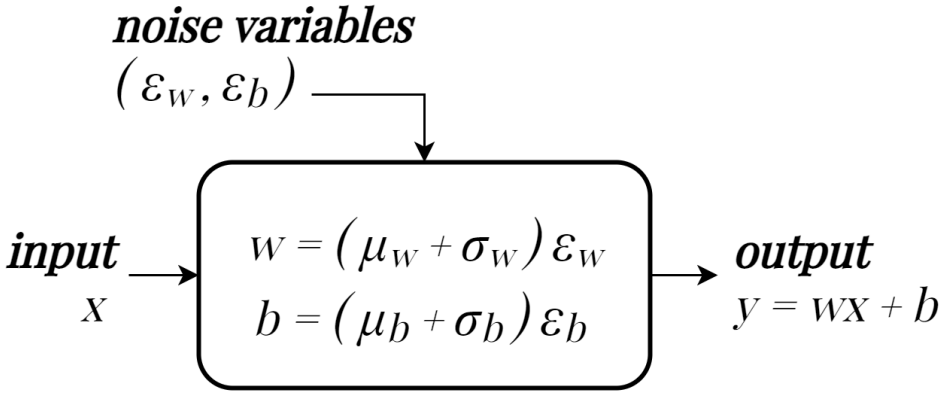


Figure 3.2: Graphical representation of noisy layer

As pictured in Figure 3.2, the parameters $(\mu_w, \sigma_w, \mu_b, \sigma_b)$ are the learnables of the network whereas (ϵ_w, ϵ_b) are noise variables. The noisy layer functions similarly to the standard fully connected linear layer, the main difference is that in the noisy layer both the weights vector and the bias is perturbed by some parametric zero-mean noise. The parameters of the noise are learned along with the remaining network weights, which means that the noise is automatically tuned by the algorithm itself.

For our purposes the optimal noise distribution for our network layers is factorized gaussian noise, which uses an independent noise source for each input and each output. The main reason to use this noise distribution is to reduce the computational overhead of random number generation in our algorithm, as recommended by the authors in the original NoisyNet implementation.[16]

Another interesting feature of the NoisyNet approach is that the degree of exploration is contextual and varies based upon the current state. This behaviour is a side-effect of the noise perturbations inside the weight vector, which induce a persistent change over multiple time steps. Such a complex, state-dependent change is the key to drive efficient exploration in larger environments.

Since at this point the network model is already modified quite extensively, we decided to run a series of tests to figure out the best configuration for our problem setting. At the end we ended up switching only the linear layers inside the shared module while keeping the layers of the action branches as default. Thus, given that the core network is composed of two layers, there are just two noisy linear layers in our model.

For the following chapters, our quantile regression branching deep Q-network agent with noisy linear layers will be referred to as trading deep Q-network (TDQN). This name is only introduced for the sake of clarity and is chosen due to the purpose of the algorithm, which is to be an automated trading agent that can operate with top results in any market and under any circumstances.

Chapter 4

Experimental Results

In this section, we prove that the trading strategy derived by our proposed TDQN agent can outperform other deep reinforcement learning algorithms. Additionally, we also compare our approach to other financial benchmarks and traditional trading strategies. For this comparison we perform backtesting in two different markets and verity results based on several financial performance measures.

4.1 Data Summary

For this project, the main portfolio employed in our experiments is composed of 30 US-based stocks. Specifically, it includes the stocks tracked by the Dow Jones 30. The reference DJIA index (*Dow Jones Industrial Average*) is a price-weighted measurement of 30 prominent companies listed on US stock exchanges.

We choose the DJIA to represent the US stock market since it is the most commonly followed equity index, but broader market indices such as the *Nasdaq 100* or *Standard & Poor's 500* can also be used as well. Another reason for choosing this stock portfolio is that we can objectively compare the performance of our agents against the reference index.

Our backtests use daily data from 2009/01/01 to 2021/12/31 obtained on *Yahoo Finance* for the performance evaluation. The dataset is, of course, split into a training set with records until 2019/01/01 for a total of ten years of daily data, while the remaining three years are used for testing. Note, however, that the agent can also learn while under evaluation since we are still working with a dynamic reinforcement learning environment. Furthermore, there are no data leakage issues due to time series record's sequential nature.

As our data model is not specific to the stock market, as explained in Section 3.1, we can also test our TDQN agent with cryptocurrencies. We selected a custom portfolio with 8 crypto trading pairs in the new cryptocurrency market environment.

The individual coins are chosen according to the market cap and the project's longevity. The pairing is Tether (USDT), a stablecoin created to maintain value equal to the US dollar.

For the performance evaluation, our backtest use data from the Binance Exchange obtained through *CryptoDataDownload*. We use hourly data from 2020/08/01 to 2021/12/31 in contrast to the previous setting. After the initial training, we start the backtesting trial from 2021/09/01 onwards.

A complete breakdown of the assets and the data used in both experiments can be found in Table 4.1 below.

US Stocks	**Crypto**
AAPL, AXP, BA,	BTC/USDT,
CAT, CSCO, CVX,	ETH/USDT,
DD, DIS, GS, HD,	ADA/USDT,
IBM, INTC, JNJ,	BNB/USDT,
JPM, KO, MCD, MMM,	XRP/USDT
MRK, MSFT, NKE,	LTC/USDT,
PFE, PG, RTX,	XLM/USDT,
TRV, UNH, V, VZ,	TRX/USDT
WBA, WMT, XOM	

Table 4.1: Test portfolios

Both datasets are composed of asset entries that contain the raw price data and volume for each timestamp. Remember that while the close price is indeed the essential value, our algorithm is also given a set of technical indicators computed starting from the initial raw price data.

The technical indicators available to our agent are provided by the `stockstats` package. The complete list is: Moving Average Convergence Divergence (MACD), Bollinger Band upper and lower bound, Relative Strength Index (RSI), Commodity Channel Index (CCI), Average Directional Index (ADX) and Simple Moving Average (SMA) over 30 and 60 days.

The market features portion of the state space also contains the VIX and turbulence values for the current timestamp along with the technical indicators. Our algorithm uses the VIX indicator as a measure of market volatility. Concretely, when the value of VIX is above a threshold, the agent halts all trading actions and sells the remaining shares in its possession.

4.2 Experimental Setting

In the experiment, we also need to set the parameters of the trading environment. Of course, many of the parameter settings can be considered default values in the financial reinforcement learning field, but let us list them nonetheless.

For the stock trading environment, we set initial capital to 1 million USD$, while it is only 100 thousand for cryptocurrency trading. In addition, for both environments, fees are set to 0.1% of the total order value whether the agent is buying or selling. This value is meant to represent an average of different real-world online exchanges, but it is only an approximation.

After thorough testing, the optimal parameters of TDQN are set to the values reported in Table 4.2. The only notable changes with respect to the respective authors recommendation are a doubling of the batch size from 32 to 64 and a decrease of the initial sigma zero for the noise distribution from 0.5 to 0.3. Additionally, we used the *Adam* optimizer for the policy and *ReLu* activation function inside the deep neural network module.

Parameter	**Value**	**Parameter**	**Value**
batch size	64	DNN input	64
replay mem size	1e6	DNN dim 1	64
learning rate (α)	1e−4	DNN dim 2	32
learning starts	1e4	quantiles	200
discount factor (γ)	0.99	sigma zero	0.3

Table 4.2: Summary of parameters

To objectively measure the performance of our trading agent against the reference index and other deep reinforcement learning benchmarks we rely on several financial metrics as provided by package `empyrical` courtesy of Quantopian. Let us provide a short overview for each of the most relevant metrics:

- *Cumulative return* and *Annualized return* measure the amount gained by the trading strategy over time. The former is the total compounded value, while the latter is a yearly average.
- *Annualized volatility* measures the variance in the returns obtained by the agent, aggregated over a year. A trading strategy with high volatility is correlated with an increased risk.
- *Max drawdown* is the maximum loss over the trading period. It is usually expressed as a percentage.

- *Sharpe ratio* and *Calmar ratio* both measure the risk-adjusted returns. Simply put, Sharpe is a function of returns versus volatility. On the other hand Calmar ratio is a function of returns over max drawdown. Sharpe ratio is the one traditionally used as measure of risk-adjusted returns.

- *Stability* measures the link between the variability of the underlying data and the strategy's correctness. Oversimplifying, we can think of stability as representing the quality of our agent trading strategy. It is important to note that stability is not related to financial theory like the other metrics, more often is associated with statistics.

Of course, cumulative and risk-adjusted returns are the most critical metrics in general. Still, for this project, we must also focus on volatility, max drawdown and stability as together they can give us an appropriate measure of our model robustness.

4.3 Stock Trading

We previously stated that we use the DJIA index as a reference benchmark regarding traditional financial strategies. On top of that, we also selected three distinct actor-critic based algorithms to compare with our agent, namely *Advantage Actor-Critic* (A2C)[33, 34], *Proximal Policy Optimization* (PPO)[35] and *Deep Deterministic Policy Gradient* (DDPG)[36]. The PyTorch implementation of these algorithms is provided by package `Stable-Baselines3`[37, 38], while the optimal parameter settings are thanks to the FinRL framework.

After extensive testing, these algorithms have been selected to represent how various agents respond to different financial market trends. Intuitively we can expect each agent to have unique strengths and weaknesses depending on the current state of the market: one model may be more efficient in a bullish market while the other may perform better in a bearish market. These same deep reinforcement learning algorithms have also been used in other notable stock trading projects, such as the ensemble agent proposed by Yang et al.[24].

Figure 4.1 demonstrates that the trading strategy devised by our TDQN agent significantly outperforms the DJIA index and the top-performing deep reinforcement learning agent DDPG. As shown in Table 4.3, our algorithm strategy obtains higher returns while maintaining lower annualized volatility and maximum drawdown with respect to the other trading strategies. Furthermore, these results should mean that the robustness of our algorithm is superior to the competing benchmarks, which is confirmed by the best stability score in the lot.

Figure 4.1: Cumulative returns comparison

Diving into more detail, we notice that our TDQN strategy obtains a Sharpe ratio of 1.01 and a Calmar ratio of 0.65 over three full years of trading, which is noticeably better than the 0.75 and 0.43 achieved by DJIA in the same period. Overall this sums up to a difference of 30% when balancing risk and return, or an 18% improvement when only returns are taken into account.

Aside from the statistics in Table 4.3 a more in-depth analysis reveals that the strategy derived by our TDQN agent has higher *win ratio* and *gain/pain ratio*. Simply put, these metrics measure the difference between winning and losing trades in number and amount respectively.

This means that our agent makes fewer losing trades, and when it does lose, the loss is minor with respect to the DJIA index. On top of that, the average drawdown is smaller and the *recovery factor* is much higher when compared to the traditional financial strategy. So when considering all of this together we can confidently say that our TDQN strategy is superior to the DJIA benchmark according to every significant metric.

Trading from Jan 1, 2019 to Dec 31, 2021

	TDQN	**DDPG**	**PPO**	**A2C**	**DJIA**
Cumulative Returns	74.1%	62.5%	57.2%	50.8%	55.9%
Annual Returns	20.3%	17.6%	16.3%	14.7%	16.0%
Annual Volatility	20.4%	21.0%	19.5%	22.3%	23.6%
Sharpe Ratio	1.01	0.88	0.87	0.73	0.75
Calmar Ratio	0.65	0.54	0.61	0.48	0.43
Stability	74.5%	59.0%	60.0%	54.8%	67.8%
Max drawdown	-31.2%	-32.5%	-26.7%	-32.0%	-37.1%

Table 4.3: Performance evaluation comparison

Among the other deep reinforcement learning agents, DDPG has the lead in returns while PPO has the lowest max drawdown and volatility. Leaving statistical results aside for a second, we can quickly see how the three benchmark agents behave differently in Figure 4.2. In this graph we show how all strategies performed during the stock market crash of February 2020, in particular, we can notice how high the volatility index VIX spiked.

For instance, even if A2C is the worst performing agent in cumulative returns, it quickly picked up on the trend reversal and behaved well in a bearish market. On the other hand, while it looks like PPO is the most adaptive to risk, from Table 4.4 we can clearly see how bad it was in a volatile market and that most of its gains were in 2021 instead. So it means that PPO is only preferred in a stable, growing market trend.

	TDQN	**DDPG**	**PPO**	**A2C**	**DJIA**
2019	23.0%	23.6%	10.6%	18.9%	23.7%
2020	12.2%	7.11%	4.11%	9.96%	5.80%
2021	38.9%	31.8%	42.5%	21.9%	26.4%

Table 4.4: Comparison of yearly returns

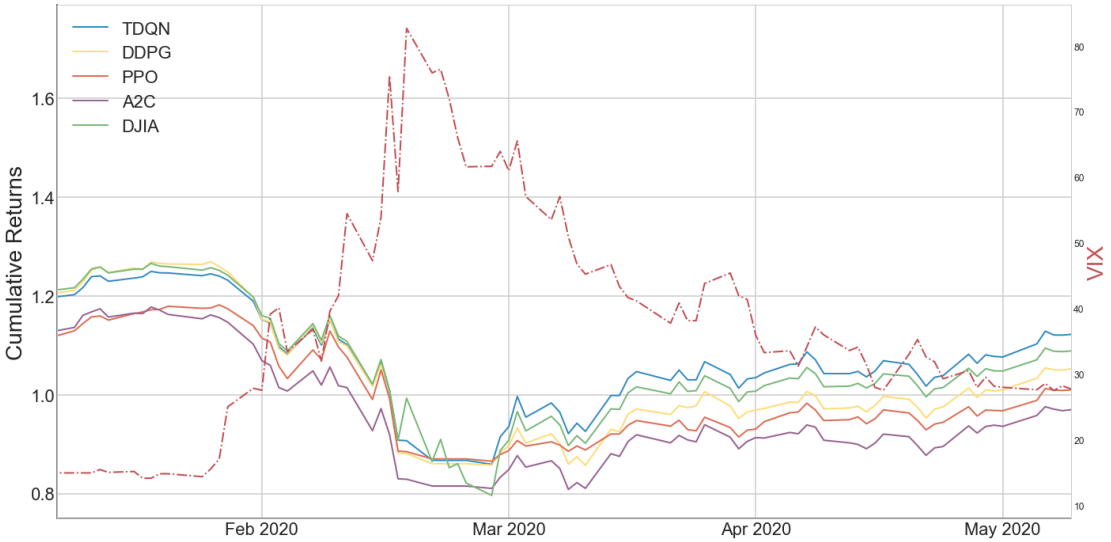


Figure 4.2: Volatile market comparison

This leaves us with DDPG, which behaves strongly in almost all conditions and is the only one that is able to match the reference DJIA index during the whole trading period. In practice DDPG produces the best all-round trading strategy, second only to our TDQN agent.

Concretely, our trading strategy is 14% better when considering risk-adjusted returns versus DDPG, or 12% in cumulative returns. Moreover, our TDQN agent obtains record returns during 2020 as shown in Table 4.4. This performance means that we can effectively trade both in upward and downwards trends, as well as high volatility periods.

The robustness of the TDQN trading strategy is also confirmed by top Sharpe and Calmar ratios when compared to the other deep reinforcement learning algorithms. Even if we fall slightly behind in volatility and maximum drawdown with respect to PPO, the returns obtained in exchange are definitely worth the risk according to these metrics.

Once again a more thorough analysis reveals that our algorithm is still ahead of the competition in every metric, including the previously cited gain/pain and payoff ratios. In particular, against the well-rounded DDPG strategy, we have higher expected returns in every timeframe (daily, monthly, quarterly and yearly) on average.

As highlighted in Figure 4.3 the TDQN agent is often ahead or on par in monthly returns, and the most significant difference between the two heatmaps is focused on the best and worst months in the entire trading window. In fact, the benchmark DDPG strategy has both higher best month and lower worst month.

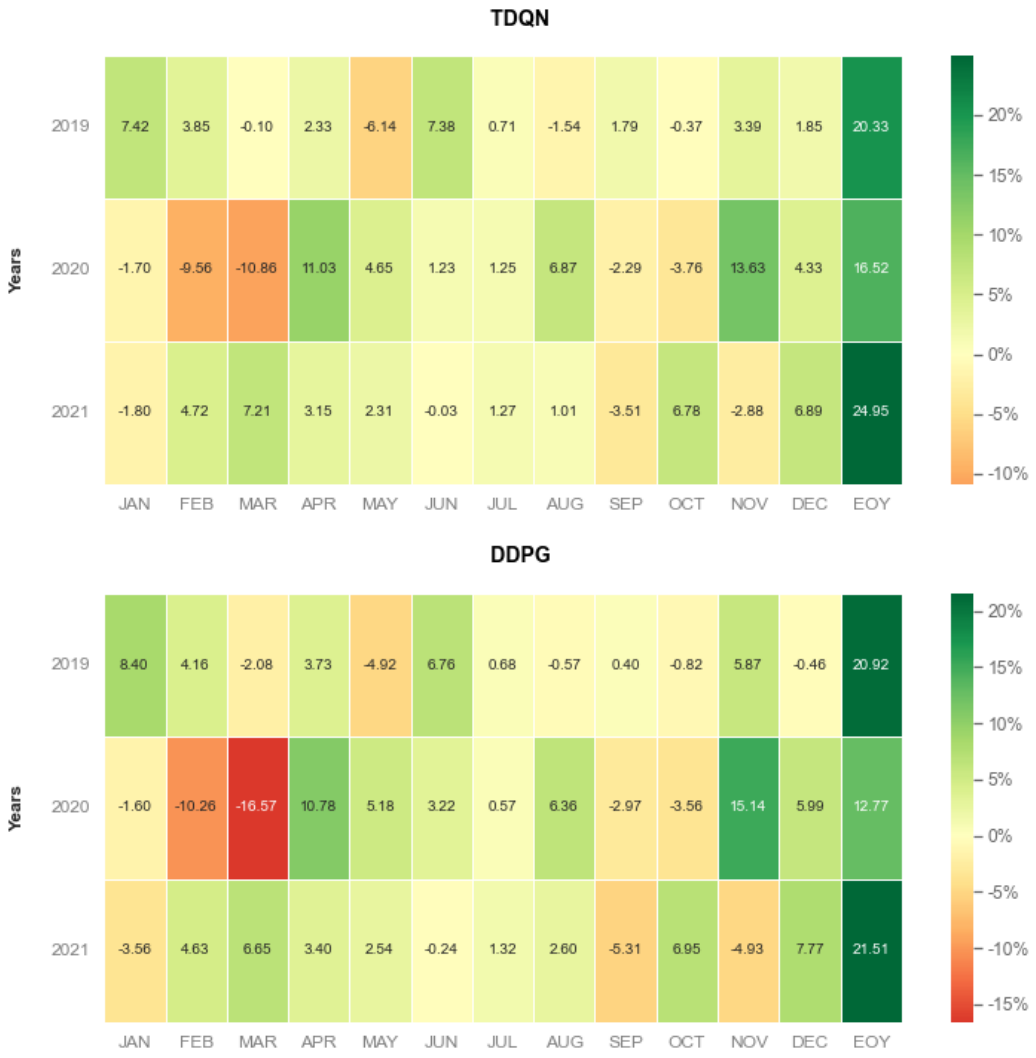


Figure 4.3: Monthly returns vs DDPG

This kind of behaviour does not pay off in the long run since TDQN still obtains a higher monthly average of 1.72%, compared to DDPG's 1.53%. Still, to put things in perspective, the DJIA average monthly returns are only 1.47%, so both agents are outperforming the market in the end.

The higher variability in monthly returns is also highlighted in the distributions pictured in Figure 4.4. The histogram of DDPG's monthly returns is narrower, and the graph shows more occurrences for lousy trading months. In contrast, TDQN strategy is more spread out and outlier months are rarer, this is undoubtedly reflected in the better stability score of TDQN as reported in Table 4.3.

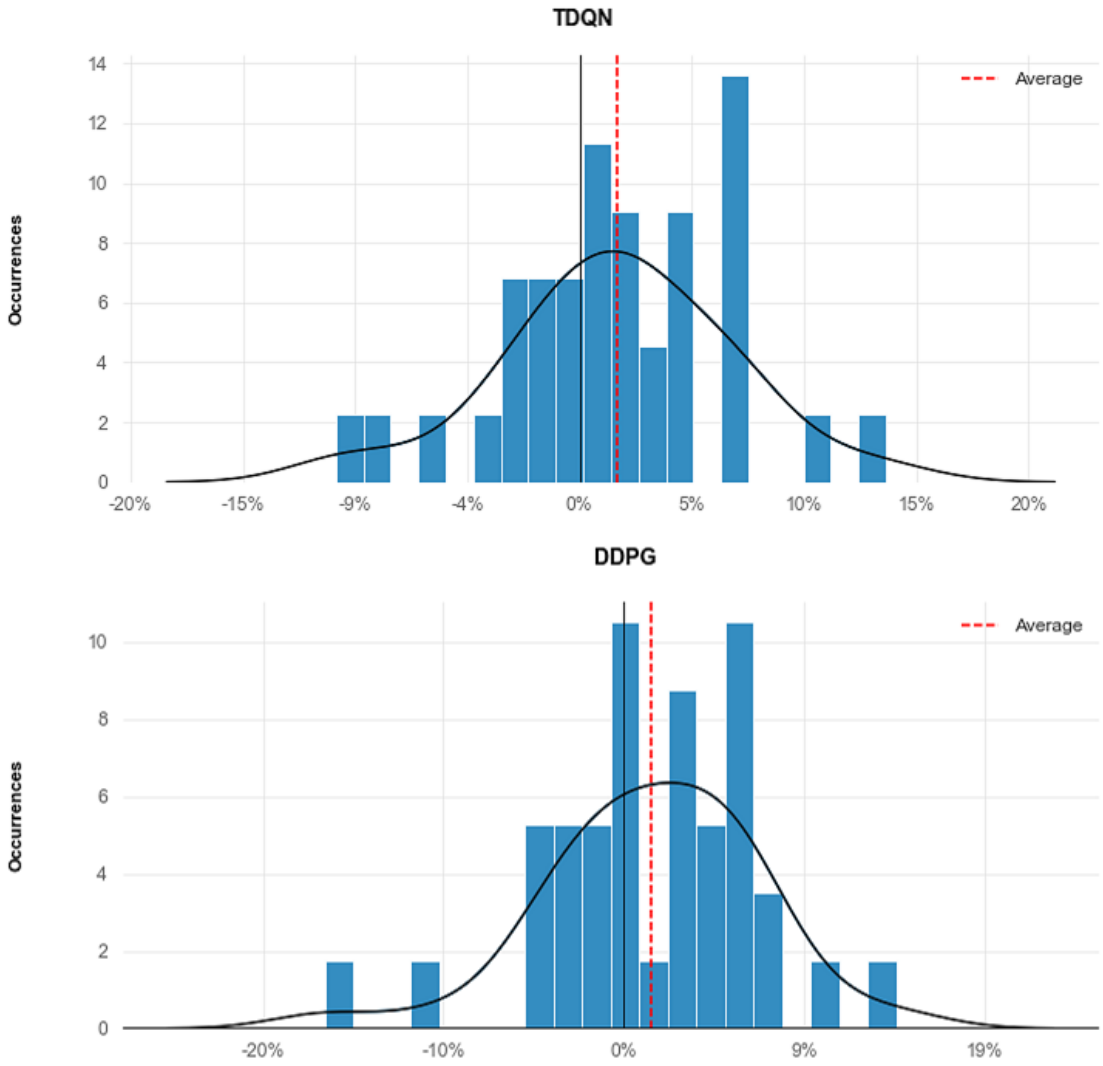


Figure 4.4: Distribution of returns vs DDPG

In conclusion, our DQN-based trading agent performs exceptionally in the stock trading market, obtaining great returns while managing relatively low risk. While it is true that most of the deep reinforcement learning algorithms were able to outperform the DJIA financial benchmark, none of them was able to obtain as stable and robust a strategy as TDQN. Consequently we uphold that the gap between the results of TDQN and the rest of the field in the stock trading environment is enough to validate our project methodology. Plus, given that our trading window is set purposely to capture a wide variety of market trends (including downward trends and high volatility periods), we claim that our TDQN agent can effectively trade in all market conditions with remarkable results.

4.4 Cryptocurrencies Trading

As an introductory notice consider that while cryptocurrency exchanges are booming in popularity, there are still many problems that we have to face when acquiring related data. In particular, it is not easy to find historical data since many coins are relatively new instruments. So while we were able to obtain daily records from 2018 onwards for our entire portfolio, it was not enough to train our algorithm effectively as the extremely limited dataset made our agent prone to overfitting the training set.

To solve this issue we used hourly data instead, which helped us obtain a big enough sample to train our agent. Still, since the objective of this project is to develop an algorithm that can trade in any financial market, we must acknowledge that less than a year and a half of data is not enough to capture an accurate representation of the market.

Specifically, the trading period for our experiment (last quarter of 2021) presented a significant downward trend that effectively limited the viable options for trading at a profit, especially considering that all agents must work with a long-only portfolio.

On top of this, whereas the stock environment is well established inside the trading framework used for this project, the cryptocurrency market is still in development and is subject to frequent changes. So, in the end, all of these issues must be considered when analysing experimental results in the cryptocurrency environment.

Introducing the results, we compare our TDQN agent's performance against an equal-weighted portfolio, which will act as our traditional financial benchmark inside of the cryptocurrency environment. In addition we also tested a bitcoin (BTC) only portfolio, but we quickly discarded it since it failed to obtain positive returns. In both cases the initial capital is spent on day one and we hold all assets until the end, following buy-and-hold criteria.

We also utilised the same three actor-critic reinforcement learning algorithms as before, but only DDPG returned positive results with our experiment's trading setting. The failure of PPO and A2C can probably be explained by the downward market trend of the trading window, that when paired with the extreme volatility of the cryptocurrencies market can quickly result in a total crash.

All considered, TDQN obtains an improvement of 43% over the equal-weight portfolio when balancing risk and return or a 129% improvement when only returns are taken into account, as reported by Table 4.5. These results may look unrealistic, but they are actually in line with our expectations and are comparable to related experiments. Regarding the deep reinforcement learning benchmark in DDPG the comparison is a wash. One algorithm edges the other by 0.1% and 0.2% in risk-adjusted and cumulative returns respectively, which is definitely not enough to distinguish the two agents.

Trading from Sept 1, 2021 to Dec 31, 2021

	TDQN	**DDPG**	**EW**
Cumulative Returns	99.9%	100%	21.4%
Annual Returns	6.21%	6.21%	1.70%
Annual Volatility	29.5%	29.5%	19.5%
Sharpe Ratio	0.29	0.30	0.19
Max drawdown	-18.0%	-13.9%	-36.7%

Table 4.5: Cryptocurrencies performance evaluation comparison

Both algorithms obtain higher volatility but lower max drawdown with respect to the equally-weighted strategy. The volatility is to be expected since the benchmark strategy does not involve any activity during the trading window. On the other hand, the maximum drawdown can tell us something about how the algorithms trade in such a volatile market.

Max drawdown is nothing but a measure of the largest loss during the trading window as explained earlier, therefore low values indicate to us that both agents do not suffer from significant losses. A more thorough analysis reveals that both algorithms trade with fewer assets per action and fewer shares for each purchase. This means that the agent mitigates the potential for major losses by minimizing the market exposure.

While this is an obvious conclusion in retrospect, especially considering the highly volatile nature of the cryptocurrencies market, this kind of behaviour is not at all granted for our algorithmic trading strategy. This kind of behaviour is what allowed both TDQN and DDPG to outperform the EW portfolio, and avoid a complete crash like the other DRL algorithms.

Nevertheless, while the overwhelmingly positive results over the financial benchmark are to be expected during a downward trend, we should still be confident in the performance of our TDQN agent in the cryptocurrencies market. We trust that our algorithm could implement a successful trading strategy even in other market circumstances and hope that, as more crypto data becomes freely available, we will be able to test this.

4.5 Ensemble Comparison

As briefly mentioned in Section 2.4, the obvious choice for the ultimate comparison is the ensemble agent[24]. It is the immediate choice not only because it is developed for the exact same environment that we use for our project, but because it is designed for the same objective: obtain a profitable trading strategy in a complex and volatile stock market.

In the simplest terms possible, the ensemble agent trades with only one of three algorithms (A2C, PPO, DDPG) for every quarter. The best performing agent is automatically chosen by using a 3-month validation rolling window and selecting the one with the highest Sharpe ratio. In this manner the authors determine the optimal algorithm for each market trend, effectively integrating and combining the best features of the three algorithms.

On top of that the authors made specific choices in their design to make the algorithm more robust and reliable, exactly as in TDQN. Related works in the field often tackle different subproblems because it is challenging to implement an all-rounder strategy, so it is exciting to find a similar project that is able to adapt and succeed in every market condition.

In the spirit of fairness we kept settings for this comparison as similar to the original as possible. In practice, this means that our backtests use daily data from 2009/01/01 to 2021/07/01 for the performance evaluation. The dataset is obtained from *Yahoo Finance* just as before and is also split into a training and testing split in a similar fashion.

As opposed to the previous example, the trading period is closer to two years since we actively trade only from 2019/04/01 to 2021/07/01. This choice is again made to keep settings as close to the original. The ensemble agent effectively begins the trading from 2019/01/01, but the first three months report no profits since they are used as a validation window, so with TDQN we start from April instead.

Furthermore, there are some slight changes in the data. First of all the ensemble agent works with a subset of the original technical indicators as it excludes the highly correlated attributes (only use MACD, RSI, CCI, ADX). Then, the high volatility cutoff threshold uses a computation of market turbulence instead of the VIX index. We mirror the same changes to keep the information equal between the two agents.

Once again, the strategy implemented by our TDQN algorithms tops the table and can comfortably outperform the ensemble agent as pictured in Figure 4.5. The DJIA benchmark, which is actually a solid strategy in this trading window, is also outclassed by a sizeable margin. In Tables 4.6 and 4.7 we report the numerical results that are not as one-sided as one might initially think. In particular the ensemble agent obtains the best score for volatility and maximum drawdown, underlying its risk-avoidant behaviour.

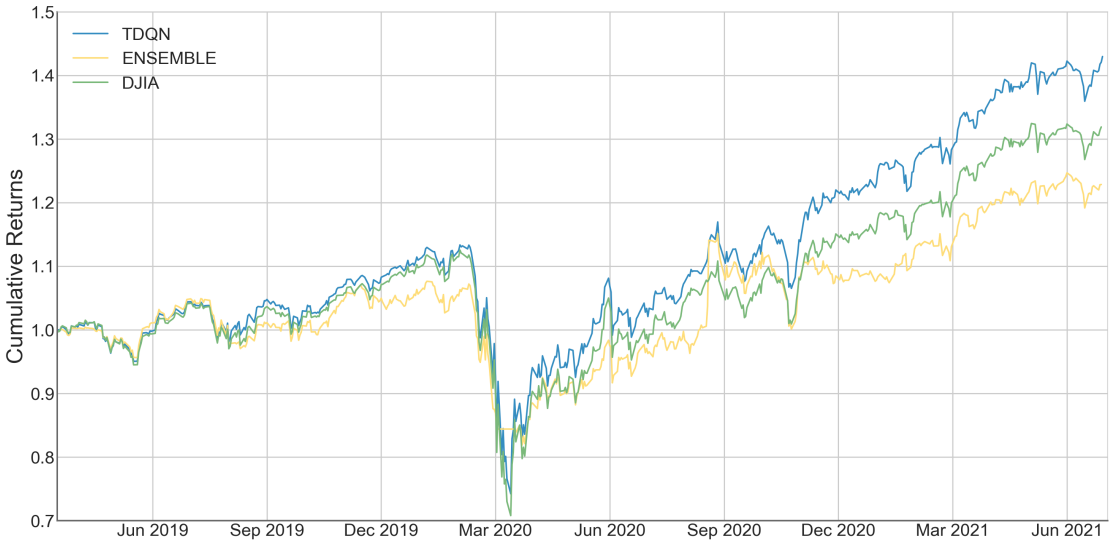


Figure 4.5: Cumulative returns comparison

Trading from Apr 1, 2019 to Jul 1, 2021

	TDQN	**ENS**	**DJIA**
Cumulative Returns	52.0%	22.9%	31.9%
Annual Returns	18.2%	9.58%	12.1%
Annual Volatility	24.3%	19.0%	25.2%
Sharpe Ratio	0.81	0.58	0.75
Calmar Ratio	0.53	0.40	0.46
Stability	55.5%	37.9%	48.4%
Max drawdown	-34.5%	-23.9%	-37.1%

Table 4.6: Performance evaluation vs ENS

	TDQN	**ENS**	**DJIA**
2019	24.5%	5.47%	9.94%
2020	7.91%	2.16%	5.16%
2021	19.7%	15.2%	16.8%

Table 4.7: Yearly returns vs ENS

Unfortunately though, yearly and cumulative returns suffer immensely from such low risk tolerance. This is also shown in the risk-adjusted returns as measured by the Sharpe and Calmar ratio, so in the end the higher safety attributes are not worth the vastly lower returns. TDQN obtains higher returns than both DJIA and the ensemble strategy. Specifically, it is 7% and 34% better in risk-adjusted returns respectively. Although it is not a clean sweep as in the previous experiment we can still be confident in our agent performance, whose robustness is still highest according to the stability metric.

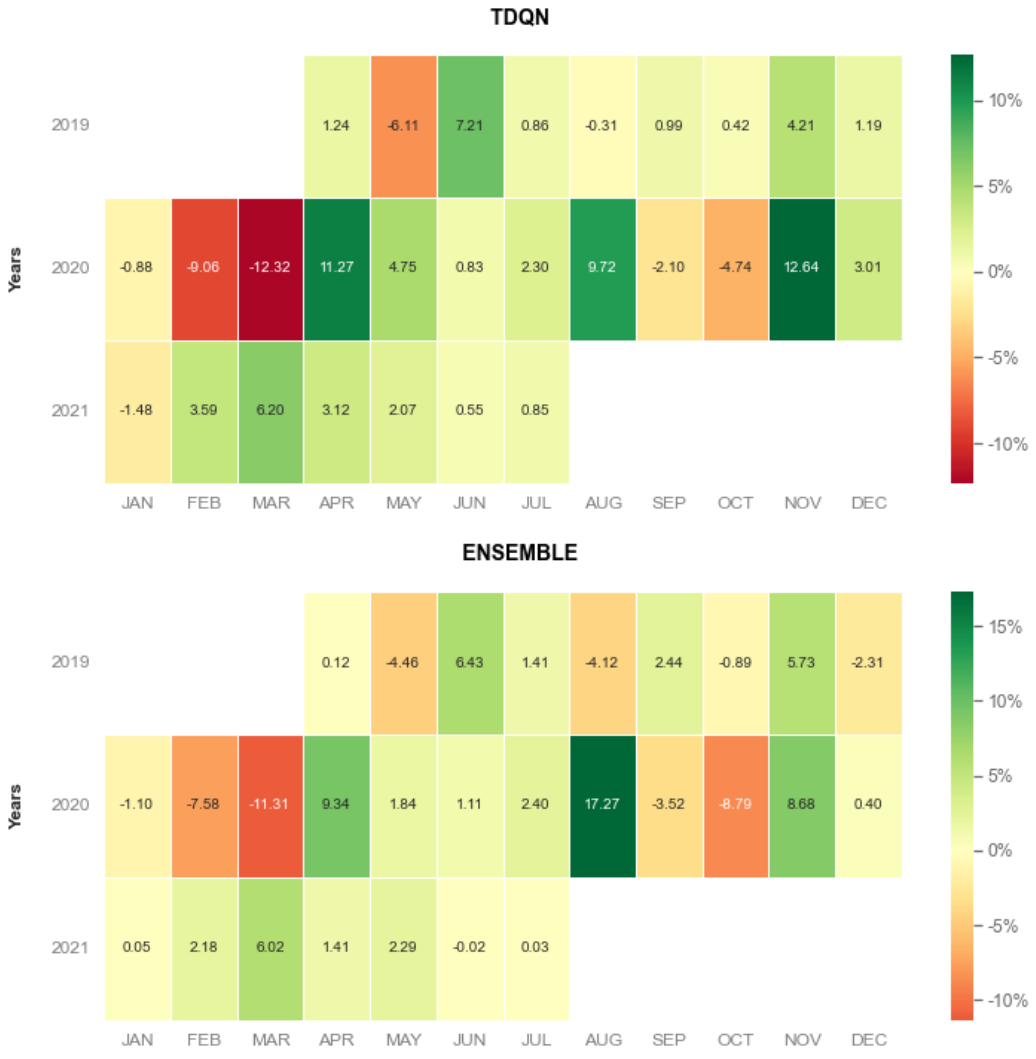


Figure 4.6: Monthly returns vs ENS

Let us briefly discuss TDQN performance versus the ensemble agent more in detail. Just as in the previous experiment our algorithm is still ahead of the competition in every metric: it shows a higher gain/pain ratio, higher payoff ratio and even higher profit ratio. This means that our agent makes fewer losing trades with respect to the ensemble agent. Plus, the loss is smaller on average when it fails, and the payoff is usually bigger when it wins.

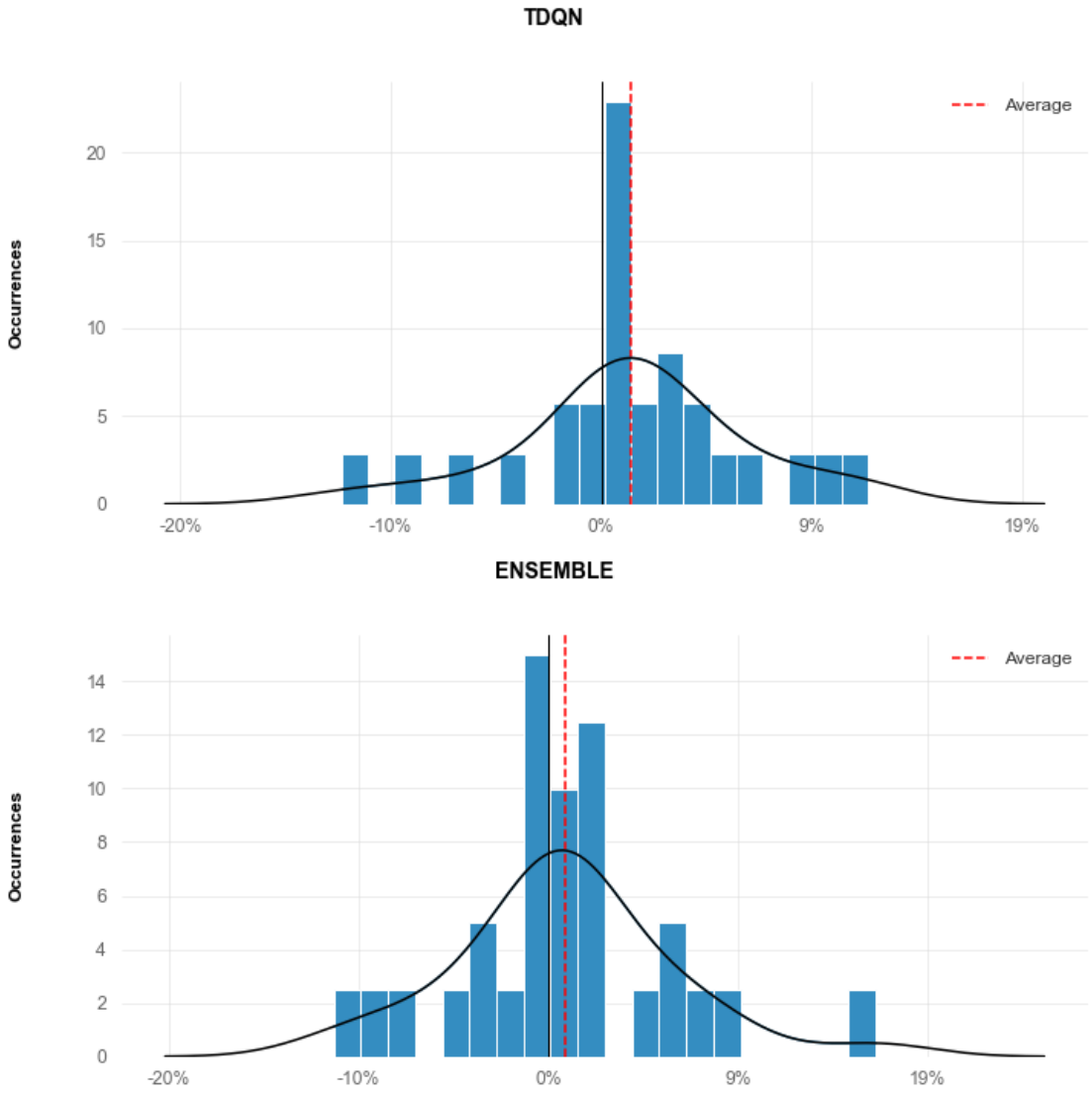


Figure 4.7: Distribution of returns vs ENS

As highlighted in Figure 4.6 our agent is often ahead of or on par in monthly returns, this is confirmed to be the case since TDQN has once again higher expected returns in every timeframe on average. As shown previously the strategy devised by our agent is more stable, its strategy obtains a higher monthly average of 1.43%, versus 0.89% of the ensemble strategy.

In this example the histogram of TDQN's monthly returns is more spread out and, admittedly, the worst trading month is not of the ensemble. In contrast, the ensemble strategy appears more focused, but negative months occurrences are still higher. This is reflected in an average down month score of -5.84% compared to TDQN's score of -5.07%. It is also important to notice that while the maximum drawdown of our algorithm is worse, the longest drawdown (in days) is actually shorter. Meaning that TDQN recovers from the stock market crash much better, as shown in Figure 4.5.

We also performed some additional experiments, including TDQN inside the set of algorithms used by the ensemble agent. In these trials we removed each of the three algorithms and used our agent instead at their place. The best resulting set is composed of TDQN, DDPG and A2C. The best performing algorithm overall is TDQN which is chosen as trading agent in 4 quarters, which means 45% of the trading window. The remaining quarters are split among DDPG and A2C, with our agent being the second choice (by Sharpe ratio) in all but one quarters. Ultimately the performance with respect to the original ensemble agent is closer than expected and sits between 3% to 5% improvement depending on the algorithms used. This is too close a result for our standards and therefore further testing is required before drawing conclusions of any kind.

Last but not least, we can now validate our previous claims about our performance in the stock market. At the end we improved the starting DQN algorithm into a well-rounded trading agent that can trade in all market trends and is able to outperform other state-of-the-art trading agents. As a result, we are confident that our agent's performance can be used as a benchmark for automated stock trading in the near future.

Chapter 5

Conclusions and future works

In this thesis, we compared different solutions to the automated stock trading task and discussed how to build an artificial environment to model the stock trading problem as in real-world exchanges. For this project's main objective we developed a trading agent specifically designed to automatically generate a profitable and reasonably safe trading strategy in any financial market. In addition, we built upon the basic algorithm in order to adjust to different market trends and implement a robust and reliable trading strategy.

At the beginning we worked to expand the agent's capabilities using the value distribution to increase stability, allowing the agent to better understand the intrinsic randomness of financial timeseries data. Simply put the distributional approach gives our agent a broader, more comprehensive picture of the financial market environment, which in turn makes our agent behaviour to be more risk-aware and learning more stable in highly-volatile markets. Finally we focused on using noisy network layers inside our automated trading system to introduce more variability in the agent's decisions and increase returns in the long run. This exploration technique is crucial to TDQN success in a complex and dynamic financial market, and is the single best improvement in our project.

Experimental results confirm our hypotheses and obtain top results compared to the DJIA financial index and other reinforcement learning algorithms. We obtain solid returns in a volatile, dynamic market under an extended trading window. Moreover, the trading strategy devised by TDQN is remarkable mainly for its robustness, based on Sharpe and Calmar ratios plus overall stability. Similar results are obtained in the cryptocurrencies market in a narrower trading window. We are confident that the methods and techniques behind TDQN are great tools for automated trading with reinforcement learning.

For future work, it will be interesting to integrate these solutions with more advanced models like IQN[39] and FQF[40]. These algorithm build upon the ideas of QR-DQN and distributional reinforcement learning in general, so it will be interesting to see how they stack up in the multi-asset trading problem. At the same time it will be interesting to explore how to learn and explore effectively with even larger state spaces such as S&P500 or STOXX600 constituent stocks.

Appendix A

Appendix

A.1 Environment setup

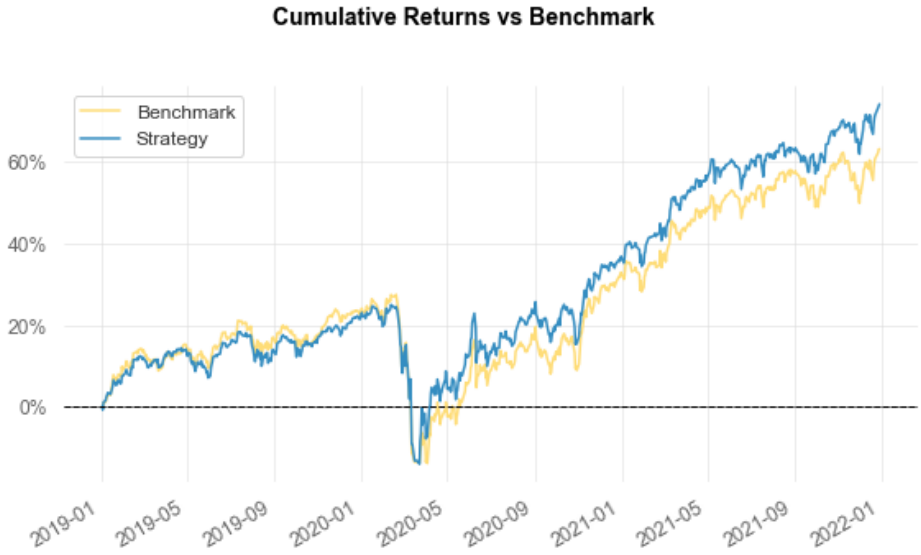
The entire project is implemented in Python 3.8. In our work we use several open-source packages, a shortlist of the primary packages and respective versions is reported below.

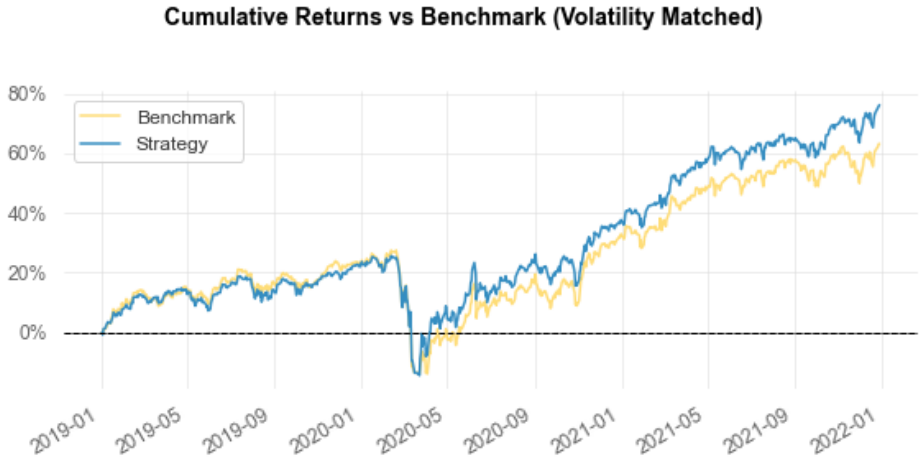
```
    pytorch: 1.9.0
    numpy: 1.19.2
    pandas: 1.3.2
    stockstats: 0.4.1
    quantstats: 0.0.50
    yfinance: 0.1.70
    pyfolio: 0.9.2
    matplotlib: 3.4.2
    seaborn: 0.11.2
    gym: 0.17.3
    stable-baselines3: 1.4
    sb3-contrib: 1.4.0
    tensorboardX: 2.2
```

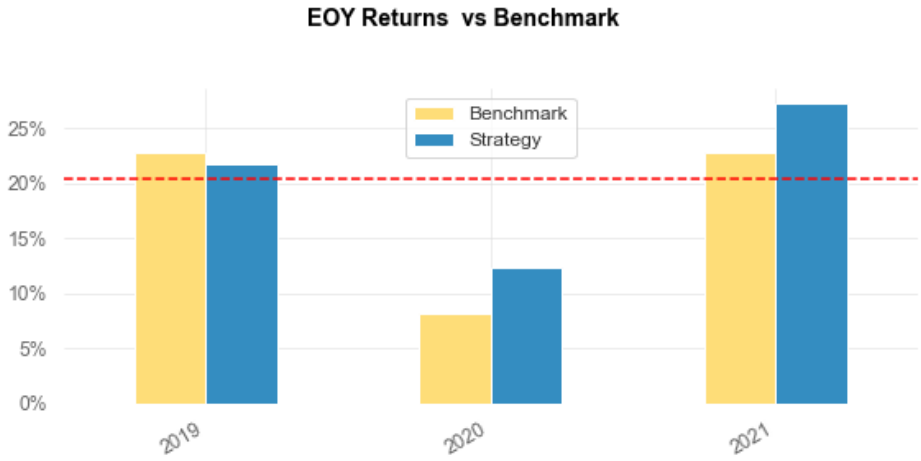
The complete list is available in the `Conda` environment file, distributed openly with the source code at *github.com/zappavignandrea*. To properly test and replicate our results we suggest to create a copy of the environment with the file, as different version of all packages makes results incomparable.

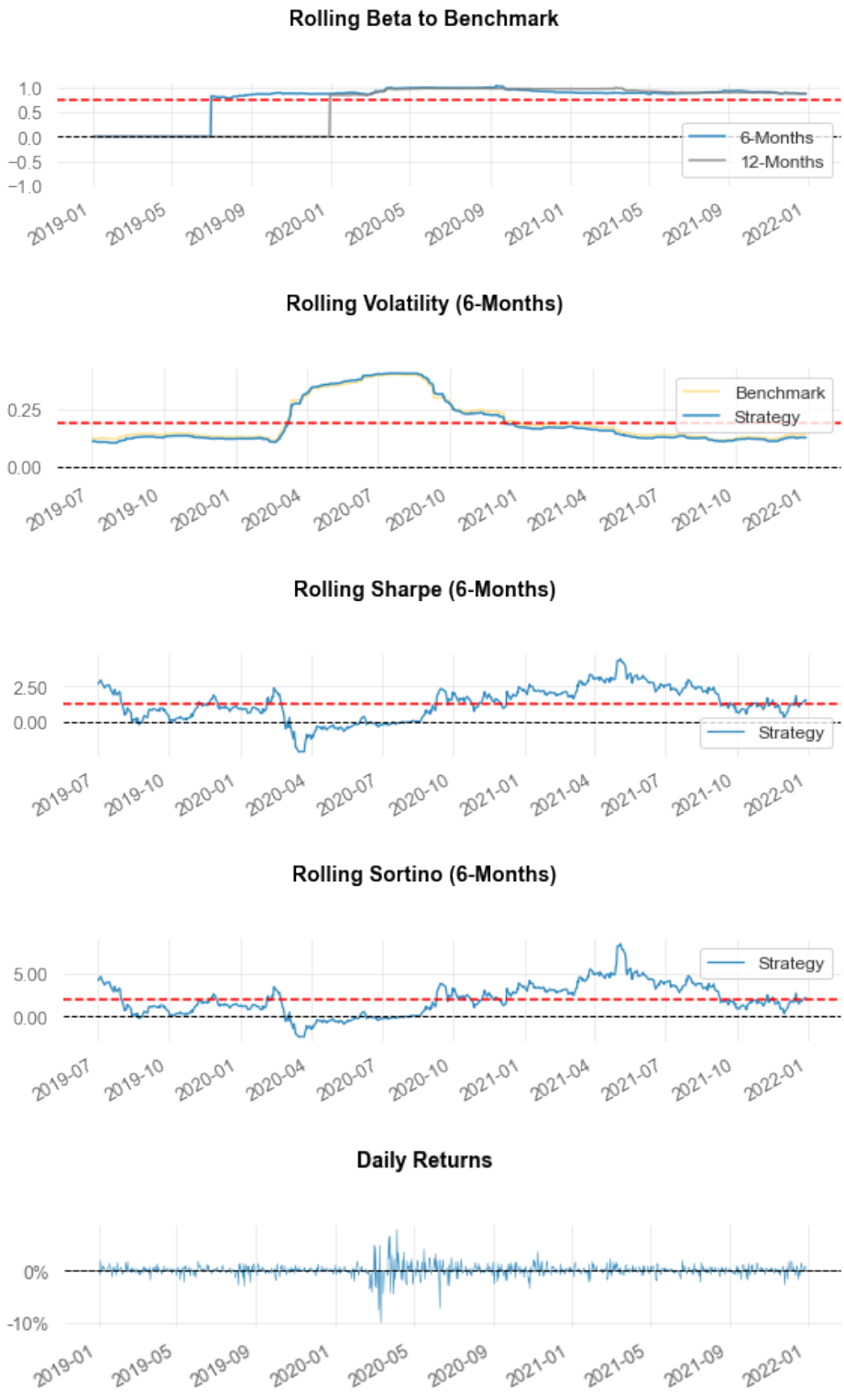
A.2 In-depth performance analysis

For additional analysis, we attach a detailed report generated by `quantstats`. In this tearsheet we compare TDQN's strategy to the DDPG benchmark as described in Section 4.3.


Cumulative Returns vs Benchmark
Benchmark
Strategy
60%
40%
20%
0%
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01


Cumulative Returns vs Benchmark (Volatility Matched)
Benchmark
Strategy
80%
60%
40%
20%
0%
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01


EOY Returns vs Benchmark
Benchmark
Strategy
25%
20%
15%
10%
5%
0%
2019
2020
2021


Rolling Beta to Benchmark
1.0
0.5
0.0
−0.5
−1.0
6-Months
12-Months
2019-01
2019-05
2019-09
2020-01
2020-05
2020-09
2021-01
2021-05
2021-09
2022-01
Rolling Volatility (6-Months)
0.25
0.00
Benchmark
Strategy
2019-07
2019-10
2020-01
2020-04
2020-07
2020-10
2021-01
2021-04
2021-07
2021-10
2022-01
Rolling Sharpe (6-Months)
2.50
0.00
Strategy
Rolling Sortino (6-Months)
5.00
0.00
Strategy
Daily Returns
0%
-10%

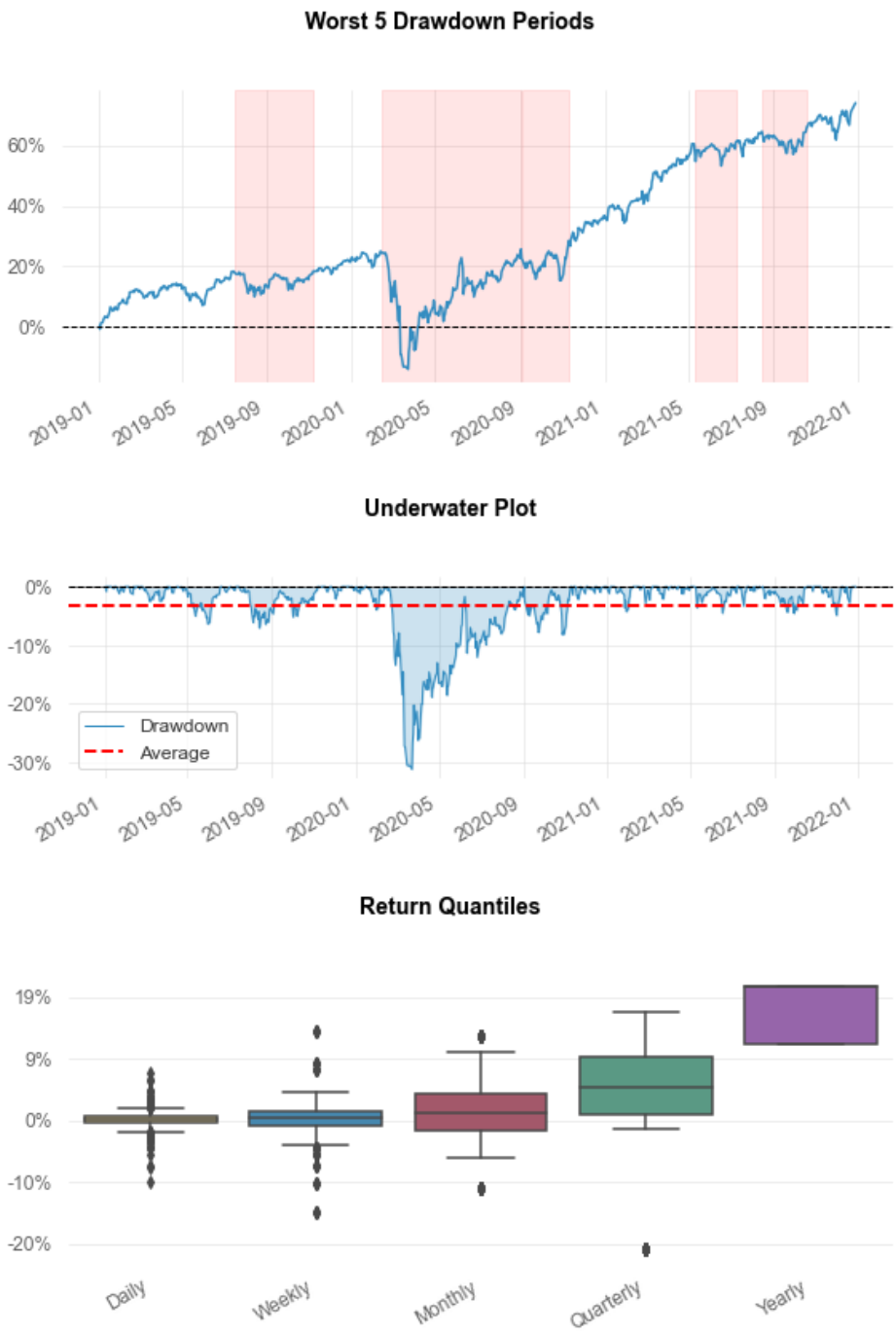


Figure A.1: TDQN vs DDPG tearsheet

Bibliography

[1] Tom M Mitchell et al. «Machine learning. 1997». In: *Burr Ridge, IL: McGraw Hill* 45.37 (1997), pp. 870–877 (cit. on p. 3).

[2] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020 (cit. on p. 3).

[3] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018 (cit. on p. 3).

[4] J. Ross Quinlan. «Induction of decision trees». In: *Machine learning* 1.1 (1986), pp. 81–106 (cit. on p. 4).

[5] Leo Breiman. «Random forests». In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on p. 4).

[6] Corinna Cortes and Vladimir Vapnik. «Support-vector networks». In: *Machine learning* 20.3 (1995), pp. 273–297 (cit. on p. 4).

[7] Sanford Weisberg. *Applied linear regression*. Vol. 528. John Wiley & Sons, 2005 (cit. on p. 4).

[8] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. Vol. 398. John Wiley & Sons, 2013 (cit. on p. 4).

[9] Stuart Russell and Peter Norvig. «Artificial intelligence: a modern approach». In: (2002) (cit. on p. 6).

[10] David Poole, Alan Mackworth, and Randy Goebel. «Computational Intelligence». In: (1998) (cit. on p. 6).

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 6, 7).

[12] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. Citeseer, 1994 (cit. on p. 13).

[13] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. «Action branching architectures for deep reinforcement learning». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018 (cit. on p. 16).

[14] Marc G Bellemare, Will Dabney, and Rémi Munos. «A distributional perspective on reinforcement learning». In: *International Conference on Machine Learning*. PMLR. 2017, pp. 449–458 (cit. on p. 17).

[15] Will Dabney, Mark Rowland, Marc Bellemare, and Rémi Munos. «Distributional reinforcement learning with quantile regression». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018 (cit. on p. 17).

[16] Meire Fortunato et al. «Noisy networks for exploration». In: *arXiv preprint arXiv:1706.10295* (2017) (cit. on pp. 18, 27).

[17] Saud Almahdi and Steve Y Yang. «An adaptive portfolio trading system: A risk-return portfolio optimization using recurrent reinforcement learning with expected maximum drawdown». In: *Expert Systems with Applications* 87 (2017), pp. 267–279 (cit. on p. 19).

[18] Hyungjun Park, Min Kyu Sim, and Dong Gu Choi. «An intelligent financial portfolio trading strategy using deep Q-learning». In: *Expert Systems with Applications* 158 (2020), p. 113573 (cit. on p. 19).

[19] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. «A deep reinforcement learning framework for the financial portfolio management problem». In: *arXiv preprint arXiv:1706.10059* (2017) (cit. on p. 19).

[20] Yuming Li, Pin Ni, and Victor Chang. «Application of deep reinforcement learning in stock trading strategies and stock forecasting». In: *Computing* 102.6 (2020), pp. 1305–1322 (cit. on p. 19).

[21] Zhuoran Xiong, Xiao-Yang Liu, Shan Zhong, Hongyang Yang, and Anwar Walid. «Practical deep reinforcement learning approach for stock trading». In: *arXiv preprint arXiv:1811.07522* (2018) (cit. on p. 19).

[22] Gyeeun Jeong and Ha Young Kim. «Improving financial trading decisions using deep Q-learning: Predicting the number of shares, action strategies, and transfer learning». In: *Expert Systems with Applications* 117 (2019), pp. 125–138 (cit. on p. 19).

[23] Quang-Vinh Dang. «Reinforcement learning in stock trading». In: *International conference on computer science, applied mathematics and applications*. Springer. 2019, pp. 311–322 (cit. on p. 19).

[24] Hongyang Yang, Xiao-Yang Liu, Shan Zhong, and Anwar Walid. «Deep reinforcement learning for automated stock trading: An ensemble strategy». In: *Proceedings of the First ACM International Conference on AI in Finance*. 2020, pp. 1–8 (cit. on pp. 19, 20, 31, 39).

[25] Zihao Zhang, Stefan Zohren, and Stephen Roberts. «Deep reinforcement learning for trading». In: *The Journal of Financial Data Science* 2.2 (2020), pp. 25–40 (cit. on p. 19).

[26] Chiao-Ting Chen, An-Pin Chen, and Szu-Hao Huang. «Cloning strategies from trading records using agent-based reinforcement learning algorithm». In: *2018 IEEE International Conference on Agents (ICA)*. IEEE. 2018, pp. 34–37 (cit. on p. 19).

[27] Souradeep Chakraborty. «Capturing financial markets to apply deep reinforcement learning». In: *arXiv preprint arXiv:1907.04373* (2019) (cit. on p. 19).

[28] Akhil Raj Azhikodan, Anvitha GK Bhat, and Mamatha V Jadhav. «Stock trading bot using deep reinforcement learning». In: *Innovations in Computer Science and Engineering*. Springer, 2019, pp. 41–49 (cit. on p. 19).

[29] Xiao-Yang Liu, Hongyang Yang, Qian Chen, Runjia Zhang, Liuqing Yang, Bowen Xiao, and Christina Dan Wang. «Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance». In: *arXiv preprint arXiv:2011.09607* (2020) (cit. on p. 20).

[30] Xiao-Yang Liu, Hongyang Yang, Jiechao Gao, and Christina Dan Wang. «FinRL: Deep reinforcement learning framework to automate trading in quantitative finance». In: *arXiv preprint arXiv:2111.09395* (2021) (cit. on p. 20).

[31] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. «Openai gym». In: *arXiv preprint arXiv:1606.01540* (2016) (cit. on p. 21).

[32] Volodymyr Mnih et al. «Human-level control through deep reinforcement learning». In: *nature* 518.7540 (2015), pp. 529–533 (cit. on p. 24).

[33] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. «Asynchronous methods for deep reinforcement learning». In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937 (cit. on p. 31).

[34] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. «Sample efficient actor-critic with experience replay». In: *arXiv preprint arXiv:1611.01224* (2016) (cit. on p. 31).

[35] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. «Proximal policy optimization algorithms». In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on p. 31).

[36] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. «Deterministic policy gradient algorithms». In: *International conference on machine learning*. PMLR. 2014, pp. 387–395 (cit. on p. 31).

[37] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. *Stable baselines3*. 2019 (cit. on p. 31).

[38] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. «Stable-Baselines3: Reliable Reinforcement Learning Implementations». In: *Journal of Machine Learning Research* (2021) (cit. on p. 31).

[39] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. «Implicit quantile networks for distributional reinforcement learning». In: *International conference on machine learning*. PMLR. 2018, pp. 1096–1105 (cit. on p. 45).

[40] Derek Yang, Li Zhao, Zichuan Lin, Tao Qin, Jiang Bian, and Tie-Yan Liu. «Fully parameterized quantile function for distributional reinforcement learning». In: *Advances in neural information processing systems* 32 (2019) (cit. on p. 45).