THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

REINFORCEMENT LEARNING APPROACH FOR ALGORITHMIC TRADING OF BITCOIN

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By
CHYRINE TAHRI
Norman, Oklahoma

2019

REINFORCEMENT LEARNING APPROACH FOR ALGORITHMIC TRADING OF BITCOIN

A THESIS APPROVED FOR THE
GALLOGLY COLLEGE OF ENGINEERING

BY

Dr. Theodore Trafalis, Chair

Dr. Dean Hougen

Dr. Sridhar Radhakrishnan

Dr. Andres Gonzalez

Abstract

Trading is in the heart of commerce in human history and its evolution is one of the most significant factors in the course of humanity. Consistently profitable traders take every negative or positive trade they make as an opportunity to improve themselves. Further, Reinforcement Learning (RL) is a framework where an agent performs actions over an environment and observes the immediate result; this feedback is used to improve the following action taken and the process starts again. We explore this principle (RL) as a plausible implementation for an algorithmic trader, implementing two different data representations throughout reinforcement learning-based trading scenarios. The first one is representing high, low, and close prices as percentages to the open price, in an attempt to learn price patterns. The second added technical indicators to the price observations, aiming to provide more sophisticated metrics that provide insights of market signals. This approach gives the opportunity to learn market analysis and signals spotting. Both agents learned to wait before selling their shares. The best result for the first agent using Bitcoin prices per minute as input data was to buy and hold rather than to do shorter trades. The second agent behaved similarly, but failed to make positive profit. We found out that the market understanding of both agents was still immature. The mappings of market states to actions is dictated by the policy, but the market does not always respond in the same way. The results show good potential for the approach but financial markets are quite large and complex and the modeling of this environment still presents a lot of challenges.

Contents

List of Figures

Chapter 1

Context

1.1 Trading

Brief history

Trading is at the heart and the principle of commerce in human history and its evolution is one of the most significant factors in the course of humanity. It is a way of bringing people together for mutual benefits, consisting of the act of buying, selling, or exchanging items. Humans would not have evolved over the centuries if they were limited to geographical boundaries. Trade has inspired innovation of techniques, which has led to faster and better communication between countries, creating a unified world for commerce.

In 1971, the National Association of Securities Dealers, an over-the-counter (OTC) traders association founded in 1939, created the first electronic stock market: the NASDAQ market. In 1976, the New York Stock Exchange (NYSE) introduced its Designated Order Turnaround (DOT) system, which allowed brokers to route 100-share order directly to specialists on the floor. These were not true electronic executions since the specialist still matched the orders, but it did bypass floor brokers. Then, in

1984, NYSE adopted a more sophisticated SuperDOT system that allowed orders up to 100,000 shares to be routed directly to the floor. More floor brokers were cut out.

In 1987, e-commerce made another leap forward, as NASDAQ expanded the Small Order Execution System (SOES), which allowed brokers with small transactions to enter their orders electronically rather than over the phone. This was done because, in the 1987 crash, many brokers simply stopped answering their calls. The rise of e-commerce took place between 1990 and 1995. In the next 5 years, e-commerce exploded as Internet traffic increased considerably. Small traders suddenly had the same access to real-time pricing as professional brokers. The word "day trader" entered the vocabulary.

The 2000s were marked by decimalization, Algorithmic Trading and high frequency trading [6]. This thesis focuses on algorithmic trading where powerful computers, combined with mathematical models and human oversight, are used to make decisions to buy or sell financial securities on an exchange.

The current era

Trading in the current era is perceived as an active style of capital market participation that seeks to outperform traditional buy-and-hold investments. Rather than trying to take advantage of long-term upward trends in the markets, traders are exploiting short-term price movements to profit from both rising and falling markets. A trader can be a person or an entity who buys and sells instruments in the financial market: stocks, bonds, commodities, derivatives, mutual funds, etc.

At the end of 2012, the size of the global stock market (total market capitalization) was approximately $ 55 trillion. By country, the largest market was the United States (about 34%), followed by Japan (about 6%) and the United Kingdom (about 6%). About 70% of US equities in 2013 were accounted for in automatic trading. Algorithmic

trading accounts for one third of the total volume of Indian cash equities and nearly half the volume of the derivatives segment.

Trading in today's financial markets generally takes place in what is known as a continuous double bidding with an open order book during an exchange. It's an elegant way of saying that there are buyers and sellers who are put in contact to exchange with each other. The stock market is responsible for matching them.

When developing trading algorithms, we have to choose the parameter to optimize. The obvious answer would be profit, but that is not enough. One should also compare the trading strategy to the benchmarks and compare its risks and volatility to other investments. Below, the most basic metrics traders use are presented.

Net profit and loss (PnL)

Simply how much money an algorithm earns (positive) or loses (negative) over a period of time, minus transaction costs.

Sharpe ratio (S)

The Sharpe ratio represents the difference in the profitability of a portfolio of financial assets (e.g, equities) relative to the rate of return of a risk-free investment (i.e. the risk premium, positive or negative), divided by a risk indicator, the standard deviation of the profitability of this portfolio: in other words its volatility.

Basically, it measures the excess return per unit of risk we take. This is essentially the return on capital relative to the standard deviation, adjusted for risk. It answers the following question: is the manager able to achieve a return above the benchmark, but with greater risk?

Thus, the higher will be the better. It takes into account both the volatility of the strategy and a risk-free alternative investment:

- If it is negative, the portfolio has performed worse than the benchmark and the situation is bad: the portfolio has a lower performance than a risk-free investment.
- If it is between 0 and 1, the over-performance of the portfolio in relation to the benchmark is made for risk taking too high. Or, the risk taken is too high for the return obtained.
- If it is greater than 1, the return on the portfolio outperforms the benchmark for an ad hoc risk taking. In other words, the outperformance is not done at the cost of a too high risk.

Maximum reduction

The maximum reduction, also called maximum loss, is a risk measure. This is the maximum difference between a local maximum and the subsequent local minimum. For example, a maximum levy of 50% means that 50% of the capital is lost at a given moment. We must then make a 100% return on the initial capital. Obviously, a low maximum reduction is better.

Importance of historical data

> "The stock market is human nature and crowd psychology on daily display, plus the age-old law of supply and demand at work. Because these two factors remain the same over time, it is remarkable but true that chart patterns are just the same today as they were 50 years ago, or 100 years ago." - *William O'Neil, How to Make Money in Stocks*

Historical price charts are the means by which a trader makes a decision. Looking at historical data may provide some insight into how a market has reacted to a variety of different variables, from regular economic cycles to sudden world events. The data

is incredibly valuable and there is no doubt about its importance to technical analysis which is the methodology used for forecasting the direction of prices through the study of past market data, primarily price and volume. The historical returns are often analyzed for trends or patterns that may align with current financial and economic conditions, hence the value of studying historical return trends.

Algorithmic trading has been around for decades and has, for the most part, enjoyed a fair amount of success in its various forms. Traditionally, algorithmic trading involves selecting trading rules that are carefully designed, optimized, and tested by humans. While these strategies have the advantage of being systematic and able to operate at speeds and frequencies beyond human traders, they are susceptible to all kinds of selection biases and are unable to adapt to changing market conditions. The main goal presented in this work is not to create a trading agent forecasting prices, but to see if trading could be seen as a learning problem where the agent is not told what to do but just what it needs to gain. In other words, the learner is going to try to learn the trading rules without human assistance.

1.2 Cryptocurrencies

The financial data of the stock exchanges are of a very high value. Hence, they are protected and difficult to access without having to pay for it. The cryptocurrency market is however an exception to this: the data is free and public.

Cryptocurrencies are simply currencies that do not have a centralized lender like a country's central bank. They are created using encryption techniques that limit the amount of currency units (or coins) created and then verify any transfer of funds after they are created. This technique of creation is called *mining* because of its theoretical similarity with the extraction of gold or other precious metals. To exploit the cryptocurrency, it is necessary to solve an algorithm or a computer puzzle more and more complex. The resolution of these algorithms requires a lot of computing power hence, electricity. In other words, it costs money to exploit them, so we can not create value from scratch. Therefore, these currencies and their value are guaranteed by the laws of mathematics, unlike any government administration or bank.

The underlying technology of a great number of cryptocurrencies is called the *Blockchain*.

> "The most critical area where Blockchain helps is to guarantee the validity of a transaction by recording it not only on a main register but a connected distributed system of registers, all of which are connected through a secure validation mechanism." - *Ian Khan*

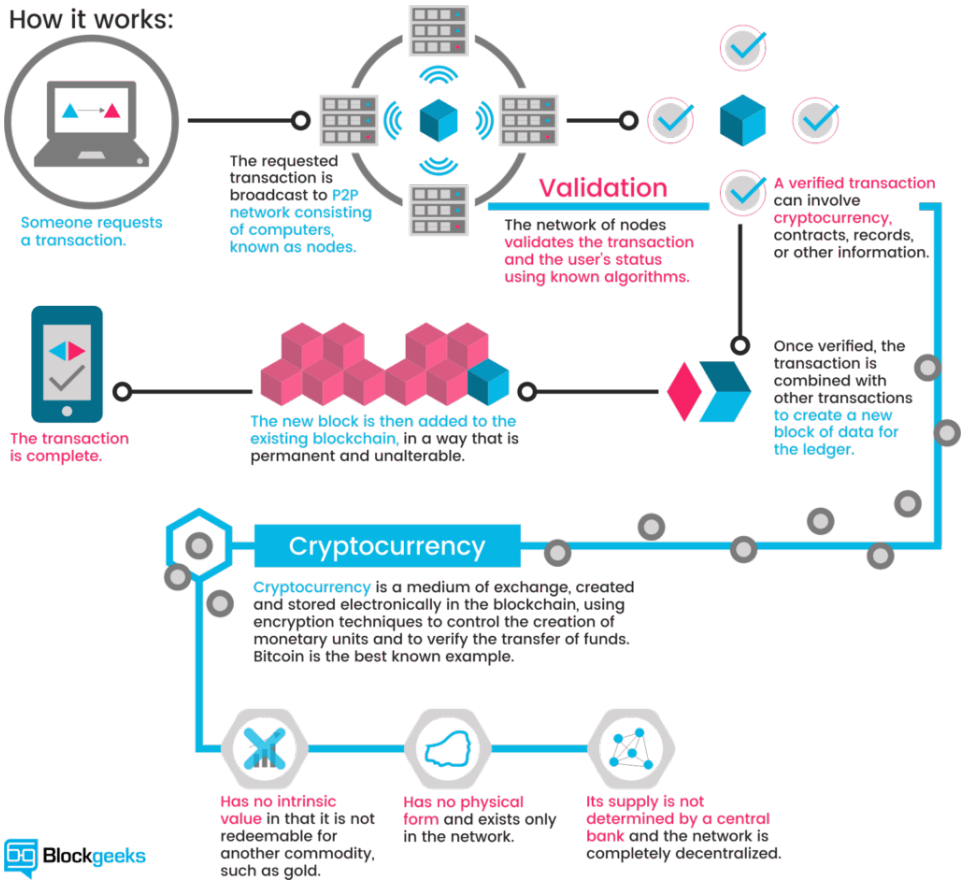

Figure 1.1: Cryptocurrency: how to make a transaction with Blockchain [1]

While the practical applications of cryptocurrencies date back to only 9 years ago, the technical aspects go back 30 years ago to the 80's. The cryptographer David Chaum was the first to theorize a cryptocurrency when he invented an encrypted algorithm allowing secure and unalterable exchanges between two parties. Chaum then founded DigiCash, one of the first companies to produce currency units based on its algorithm. However, only DigiCash could produce the currency. After encountering legal problems and rejecting a partnership with Microsoft that would have allowed DigiCash to be associated with all Windows operating systems, the company went bankrupt in the late 90's [7].

The modern cryptocurrencies we know today began with Bitcoin, which was first

described by an anonymous entity (the identity has never been confirmed as a single person or group) Satoshi Nakamoto. Bitcoin was made public in early 2009 and a large group of enthusiasts started to exploit, invest and exchange the currency. The first Bitcoin market was created in February 2010. At the end of 2012, the website hosting and development platform Wordpress became the first major distributor to support Bitcoin payment. This step was essential because it gave the currency real credibility and showed that big companies had confidence in it as a currency.

These currencies have a number of advantages over the currencies we know and use today. This makes them so attractive to long-term investors and short-term speculators. Of course, as with any investment, cryptocurrencies have potential disadvantages. As adoption increases, the number of actual uses also increases. Physical goods, gift cards and even hotel reservations can be purchased using cryptocurrency. Some bars and restaurants have also begun to accept it as a means of payment. A number of Non Governmental Organizations are now accepting donations in Bitcoin and other cryptocurrencies. There are also more illicit uses, with underground e-markets dealing with illegal products, such as the Silk Road.

The currency used for this thesis is Bitcoin (BTC), the number one in market capitalization: $114.05 billion in Q3'18 (third quarter of 2018).

1.3 Reinforcement Learning field

Reinforcement learning (RL) is about learning what to do - mapping situations (states) into actions - so as to maximize a digital reward signal. The learner is not informed of the actions to be undertaken, as in most forms of machine learning, but rather must discover what are the most profitable actions by trying them.

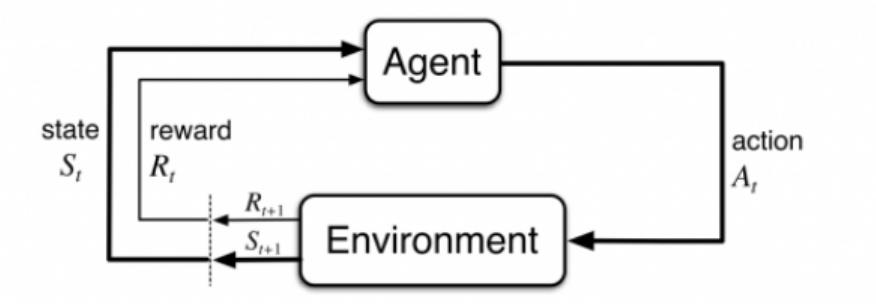

Figure 1.2: Agent interacting with the environment in RL

The agent observes a state and chooses an action to undertake. The environment transits to another state resulting of this action and sends back to the agent the reinforcement and the new state. Reinforcement is a scalar value that is generally negative to express a punishment and positive to indicate a reward. Unlike supervised learning techniques, reinforcement learning methods do not involve the presence of a teacher who can judge actions taken in a particular situation.

Application example: [5] A robot decides whether to enter a new room looking for more waste to collect or start trying to find its way back to its battery charging station. It makes its decision based on the speed and ease with which it has been able to find the charger in the past.

1.3.1 Reinforcement Learning for trading

The biggest problem traders face is their emotions [8] and financial markets are wild. They behave randomly, anything can happen and traders can face big losses at anytime. A good trader's goal is not to believe they can forecast the next layer of irrational reaction to unpredictable future news, but to develop a system, whether it would be driven by technical, sentimental or fundamental analysis, that can increase the probability of their success. A trader can increase their probability of winning either through fundamental or technical analysis but the best analysis can never produce a 100% certainty. In reality, the highest win rate that the best analysis can produce is far from 100%. However, as long as the trader has a trading plan that can produce positive expected value, they can expect consistent result over a reasonably large number of trades.

Since reinforcement learning is about learning policies so as to maximize the reward, and we can set the number of trades to as many as we want, a reinforcement learner would be perfectly capable to mimic, and perhaps outperform a human trader's mind. The approach presented in this thesis uses little analysis of the data itself in the common fundamental and technical ways. Instead, only few technical indicators are fed to the system as input parameters to the decision making process. The agent learns by itself the importance it should give to each input. In other words, it develops its own system of maximizing its probability of winning through error and trial, exploration and exploitation, tested on dense historical price data. Through the reward/punishment principle, the amount of "emotions" an agent should have is configurable, just like its learning rate. An agent would know that losing is bad, but would still be able to act "reasonably" even when facing an unknown pattern of big losses as long as we define how big is big for it or even better, defining the risk it should avoid. Imagine a group of human traders who place the exact same trade and all of them lose money. The first

one reacts with frustration, trades more aggressively to win its money back and end up losing more money by the end of the day. The second one becomes discouraged, curses the market and gives up for the day. The third one pauses its trading, reassess its emotions away from the market, comes back and reevaluates its strategy and waits for a signal of opportunity to place a good trade that would save a little bit the money they lost. What determines winners from losers is how they behave and control their emotions. When traders do well, they feel good. When they encounter loss, they become discouraged, doubtful and frustrated, questioning themselves and their strategies. Instead of dealing constructively with their losses, they react to the emotions triggered by their personalization of the events. Successful traders are those who focus on the perfect execution of a profit target or a stop loss level. *Consistently profitable traders take every negative or positive trade they make as an opportunity to improve themselves.* This is the core of reinforcement learning. The agent is by definition developed to make the best decision facing a particular situation based on its previous experience. Thus, the connection between trading and reinforcement learning is quite obvious. The goal of this thesis is not to build an algorithmic trader surpassing humans and becoming a Wall Street killer. Instead, we focus on what algorithms are good at: the ability to unemotionally spot a hardcoded pattern and act on it. The aim here is to prove that we can build a reinforcement trading agent that makes positive profit, as long as we have a good data representation of the environment. However, let us first explain the challenges facing us in this approach.

1.3.2 Challenges of applying RL to trading

Exploration-Exploitation trade-off

One of the challenges present in RL and not in supervised and unsupervised learning is the trade-off between exploration and exploitation. To get a lot of reward, an agent must prefer the actions it has tried in the past and found effective in producing rewards. But to discover such actions, it must try actions it has not selected before. The agent must *exploit* what it already knows to get a reward, but it must also *explore* to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without the task being in vain. This is the question every agent must learn to answer from a very early age; does it continue following this policy that is giving nice returns, or should it take some relatively sub-optimal actions now in case there is a possibly bigger payoff later? This problem is hard because there can be no right answer in general; there is always a trade-off. The agent must try various actions and gradually favor those that seem to be the best. On a stochastic task, each action must be attempted multiple times to obtain a reliable estimate of the expected reward. This is a general challenge of Reinforcement Learning, not only its application to the trading world.

Deep Reinforcement Learning baseline

There is relatively little work published on how to apply deep reinforcement learning to financial trading [9] compared to the work applied to robotics, control, video-games, operations research, or even human-computer interaction. There is no clear baseline, nor a model or an architecture of hyper-parameters. At the beginning of this research, important resources that we used were not available yet [3][4][10]. We had to link the problem and the solution piece by piece, through the fundamentals of each discipline.

Later, posts on internet as well as lectures about applying reinforcement learning to trading stocks became available and we built a clearer picture of what a model should include. The work presented here details the vision we had.

Timescale

We must decide on what time scale we want to act; days, hours, minutes, seconds, milliseconds or even nanoseconds. All require different approaches. Someone who buys an asset and retains it for days, weeks or months often makes a long-term bet based on an analysis, such as *Will Bitcoin be a success?*. Often these decisions are dictated by external events, news or a fundamental understanding of the value or potential of the assets. Since such an analysis generally requires an understanding of how the world works, it can be difficult to automate the use of machine learning techniques.

In contrast, we have high frequency trading (HFT) techniques, where decisions are based almost entirely on market microstructure signals. Decisions are made on time scales of nanoseconds and business strategies use dedicated exchange connections and extremely fast but simple algorithms running on FPGA hardware. Another way of thinking about these two extremes is that of *humanity*: the first requires an overview and an understanding of the functioning of the world, human intuition and high-level analysis, while the second concerns the simple, but extremely fast, matching of models. The ideal would obviously be to find a middle ground between the two approaches. We want to act on a time scale where we can analyze the data as quickly as possible, but also being smarter allows us to beat the *fast but simple* algorithms.

Partial observability of the stock market

There are two types of environments in which the agent behaves differently:

- Deterministic environment: In a given state, if the agent repeats a given action, it will always go to the same next state and receive the same reward.
- Stochastic environment: In a given state, if the agent repeats a given action, the new state and received reward may not be the same each time.

The stock market can be considered a non-deterministic and partially observable domain, because investors never know all information that affects prices and the result of an investment is always uncertain. Deterministic environments are easier to solve, because the agent knows how to plan its actions with no-uncertainty given the environment Markov decision process (MDP). Various elements drive prices at different scales:

- Opening and closing prices have their own patterns.
- News and rumors are the driving forces when it comes to multi-day horizons. Specific company news can happen at any time without any prior notice.
- High frequency trading and algorithmic trading are the main drivers of price at short intervals (less than a day).
- Value investing and economic cycles matter the most when it comes to price changes at a multi-year range.

Deterministic environments are easier to solve, because the agent knows how to plan its actions with no-uncertainty. However, in the financial markets no one really has a complete picture at any point in time. We do not know what will happen tomorrow and yet we still have to make a decision about our trade. The information we have

is quite minimal. At the same time, the distribution of data is constantly changing. Financial time-series is a partial information game that is really hard even for humans.

Figure 1.3: Stock market humor: partial observability [2]

In the next chapter, we present a detailed description of a reinforcement learning problem and the feasibility assessment of applying it to trading.

Chapter 2

Literature Review

2.1 Reinforcement Learning – Theory

2.1.1 Reinforcement Learning formalisms

Beyond the agent (the learner) and the environment, we can identify four main sub-elements of a reinforcement learning system: a policy, a reward function, a value function and, optionally, a model of the environment [5].

A policy defines how the learning agent behaves at a given moment. Basically, a policy is a mapping of the perceived states of the environment to the actions to be undertaken in those states. It is the nucleus of an agent in the sense that it is sufficient to determine the behavior.

A reward function defines the goal in a reinforcement learning problem. Roughly, it maps each perceived state (or state-action pair) of the environment to a unique number, a reward, indicating the intrinsic desirability of that state. For many problems, the consequences of an action are not immediately apparent after the execution of the action, but only after a number of other actions have been taken. In other words, the selected action can not only affect the immediate reward or punishment received by

the learner, but also the reinforcement it could get in later situations, namely rewards and delayed punishments.

While a reward function indicates what is good in an immediate sense, a **value function** specifies what is good in the long run. The value of a state is the total amount of reward an agent can expect to accumulate in the future, starting from that state. Rewards are somehow primary, while values, as the rewards' predictions, are secondary. Without rewards, there could be no value and the sole purpose of estimating values is to get more rewards. Nevertheless, it is the values that concern us the most when taking and evaluating decisions. Action choices are based on value judgments. The rewards are essentially given directly by the environment, but the values must be estimated and re-estimated from the observation sequences made by an agent throughout its lifetime. In fact, the most important element of all reinforcement learning algorithms is a method of efficiently estimating values [5].

The final element of some systems is a model of the environment. It mimics the behavior of the environment. Models are used for planning, which means any way of deciding an action by considering possible future situations before they are actually experienced. This is particularly interesting in the case of trading, since planning can only be beneficial if the investor seeks to minimize the risk associated with his strategy.

2.1.2 Markov Decision Process

The theoretical foundation of reinforcement learning is Markov decision processes [3]. To describe this, we will present it from the basis of Markov chains, and then add the reward and actions systems to lead to Markov reward processes and then Markov decision processes [11].

Markov chain

A Markov chain is a mathematical system that hops from one state (a situation or set of values) to another. The state of the system is what is observable but not influenceable. All possible states for a system form the *state space*. The observations over time form a sequence of states, also called a *chain*. If the future system dynamics from any state have to depend on the current state only, the system is said to fulfil the **Markov property**. The point of this property is to make every observable state self-contained to describe the future of the system and in such a case, the states have to be distinguishable from each other and unique. The Markov property ensures that the learner can behave optimally by observing only its current state (*i.e.* it is not necessary to follow the history) so that the learner does not need to know how that happened. Such a system is called a Markov process or Markov chain and we capture transition probabilities with a *transition matrix*, which is a square matrix of the size $N * N$, where N is the number of states in the system. Every cell in a row i and column j in this matrix contains the probability of the system to transit from state i to state j. In practice, we rarely have the opportunity of knowing the exact transition matrix but we have observations of the system's states which are called *episodes*. Sometimes we have uncertainty in the elements of the transition matrix. It is not complicated to estimate the transition matrix by the observations; we count all the transitions from every state and normalize them to a sum of 1.

Markov reward process

Reward can be represented in various forms. The most common way is to have another square matrix similar to the transition matrix with rewards for transitioning from state i to state j. Rewards can be negative or positive, large or small.

Next, we add a *discount factor* $\gamma \in [0, 1]$. In a Markov reward process, for every

transition we have a reward in addition to the chain of state transitions. Hence, all the observations now have a reward value attached to every transition of the system. For every episode, the **return** G_t at time t is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

For every time step, we compute the return as a sum of subsequent rewards, but more distant rewards are multiplied by the discount factor raised to the power of the number of steps we are away from the starting point at time t. The discount factor γ stands for the foresightedness of an agent. If $\gamma = 1$, G_t equals the sum of all subsequent rewards and corresponds to the agent with perfect visibility of any subsequent rewards. If $\gamma = 0$, G_t equals the immediate reward without any subsequent state and corresponds to absolute shortsightedness. These values, 0 and 1 are not useful and usually γ is set in between. This means that the agent looks into future rewards but not too far. The closer to 1, the more steps ahead of us we need to take into account.
In practice, return is not very useful, as it is designed for every specific chain we observe, so it can vary widely even for the same state. However, we can calculate the mathematical expectation of return of any state, the **value of a state** $V(s)$ denoted by:

$$V(s) = \mathbf{E}[G|S_t = s] \tag{2.2}$$

For every state s, the value $V(s)$ is the expected return we get by following the Markov reward process.

Markov decision process

Here we add actions that the agent can take to the Markov reward processes. The set of possible actions is called *action space*. Now we need to condition the transition

matrix with action, which means we need an extra action dimension and that turns the matrix into a cube. Now the agent is no longer passively observing state transitions, but can actively choose an action to take at every time. Hence, we have a matrix where the depth dimension contains actions that the agent can take and the other dimension is that the target state system will jump to after the agent performs the chosen action.

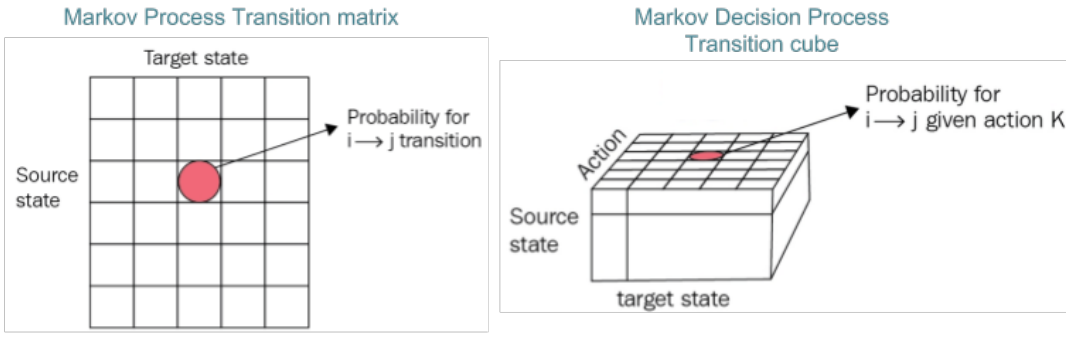

Figure 2.1: Transition in Markov Process (left) - Markov Decision Process (right) [3]

The agent therefore can affect the probabilities of target states by choosing an action. When adding actions to the reward matrix in the same way with the transition matrix, the reward will now depend not only on the state the agent ends up in but also on the action that leads to this state.

Finally, we introduce the central aspect for Markov Decision Processes: the **policy**. An intuitive definition is some set of rules that control the agent's behavior. The main objective in a reinforcement learning problem is to maximize the gain. Different policies can lead to different returns so it is important to find a good policy. Formally, policy is defined as the probability distribution over actions for every possible state:

$$\pi(a|s) = \mathbf{P}[A_t = a|S_t = s] \tag{2.3}$$

This is a probability, not a concrete action to introduce randomness into an agent's

behavior. To summarize, a Markov Decision Process (MDP) is a tuple of the following elements:

- $\mathbb{S}$: Set of states. At each time step the state of the environment is an element $s \in \mathbb{S}$
- $\mathbb{A}$: Set of actions. At each time step the agent chooses an action $a \in \mathbb{A}$ to perform
- $P[s_{t+1}|s_t, a_t]$: State transition model describing the changes in the environment state when the agent performs an action a at the current state s
- $P[r_{t+1}|s_t, a_t]$: Reward model describing the reward value that the agent receives from the environment after performing an action. It depends on the current state s and the action a performed
- γ: discount factor that controls the importance of future rewards.

A reinforcement learning task that satisfies the Markov property is a MDP.

2.1.3 Q-Learning

From a mathematical point of view, reinforcement learning is closely related to dynamic programming, which is a well-known method for solving Markov decision problems [12]. Dynamic programming is a collection of algorithms used to compute optimal policies given a perfect model of the environment as a MDP. The key idea of these algorithms, and of reinforcement learning in general, is the use of value functions to organize and structure the search for good policies [13][14]. Dynamic programming techniques are generally classified into two approaches: the value iteration approach and the policy iteration approach. Value iteration computes the optimal state value function by iteratively improving the estimate of the state value. Policy iteration redefines the policy at each step and computes the value according to this new policy until the policy converges. The same classification is used in reinforcement learning. This thesis focuses on value iteration methods.

Value iteration

The value function represents how good is a state for an agent to be in. Value $V(s)$ is defined as an expected total reward that is obtainable from the state:

$$V(s) = \mathbf{E}[\sum_{t=0}^{\infty} r_t \gamma^t | S_t = s] \tag{2.4}$$

Where r_t is the local reward obtained at the step t of the episode when observing state s.

Value is always computed with respect to some policy π that the agent follows. A formal description is as follows:

$$V_{\pi(s)} = \mathbf{E}_{\pi}[\sum_{t=0}^{\infty} r_t \gamma^t | S_t = s] \quad \forall s \in \mathbb{S} \tag{2.5}$$

Among all possible value-functions, there exists an optimal value function that has higher value than other functions for all states:

$$V^*(s) = \max_{\pi} V_{\pi(s)} \quad \forall s \in \mathbb{S} \tag{2.6}$$

The optimal policy π^* is the policy that corresponds to optimal value function:

$$\pi^* = \arg\max_{\pi} V_{\pi(s)} \quad \forall s \in \mathbb{S} \tag{2.7}$$

Value of action

The value of an action, $Q_{s,a}$, is the total reward we can get by executing action a in state s and can be defined via V_s. In other words, $Q_{s,a}$ is an indication for how good it is for an agent to pick action a while being in state s. The name Q-learning came from this quantity.

$$Q_(s, a) = \mathbf{E}_{s' \sim S}[r_{s,a} + \gamma V_{s'}] = \sum_{s' \in S} p_{a,s \to s'}(r_{s,a} + \gamma V_{s'}) \tag{2.8}$$

The value of Q for the state s and action a equals the expected immediate reward and the discounted long-term reward of the destination state s'. We can express $Q(s, a)$ as follows:

$$Q(s, a) = r_{s,a} + \gamma \max_{a' \in A} Q(s', a') \tag{2.9}$$

Since $V^*(s)$ is the maximum expected total reward when starting from state s, it will be the maximum of $Q^*(s, a)$ over all possible actions. Therefore, the relationship between $Q^*(s, a)$ and $V^*(s)$ is easily obtained as:

$$V^*(s) = \max_{a} Q^*(s, a) \quad \forall s \in \mathbb{S} \tag{2.10}$$

If we know the optimal Q-function $Q^*(s,a)$, the optimal policy can be easily computed by choosing the action a that gives maximum $Q^*(s,a)$ for state s:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad \forall s \in \mathbb{S} \tag{2.11}$$

Bellman Equation

The Bellman equation, using the dynamic programming paradigm, provides a recursive definition for the optimal Q-function. $Q^*(s,a)$ is equal to the summation of immediate reward after performing action a while in state s and the discounted expected future reward after transition to a next state s'.

$$Q^*(s,a) = r_{s,a} + \gamma \mathbf{E}_{s'}[V^*(s')] \tag{2.12}$$

$$\text{where} \quad \mathbf{E}_{s'}[V^*(s')] = \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')$$

With respect to (2.10) we have the equation:

$$V^*(s) = \max_a (r_{s,a} + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')) \tag{2.13}$$

The value-iteration algorithm allows us to compute the values of states and values of actions of MDPs with known transition probabilities and rewards by iteratively improving the estimate of V_s ($Q_{s,a}$ resp.). The algorithm initializes V_s ($Q_{s,a}$ resp.) to arbitrary random values. It repeatedly updates the V_s and $Q_{s,a}$ values until they converge. Value iteration is guaranteed to converge to the optimal value function if every state is visited an infinite number of times and every action is tried an infinite number of times in it. In other words, we need an infinite samples to learn from, and we need them everywhere.

Algorithm 1 Value-iteration; Value of states

1: Initialize values of all states V_i to some initial value (usually zero)
2: For every state s in the MDP, perform Bellman update:

$$V_s \leftarrow \max_a \sum_{s'} p_{a,s\to s'}(r_{s,a} + \gamma V_{s'})$$

3: Repeat from step 2 for some large number of steps or until changes become too small

Algorithm 2 Value-iteration; Value of actions

1: Initialize values of all $Q_{s,a}$ to zero
2: For every state s in the MDP and every action a in this state, perform update:

$$Q_{s,a} \leftarrow \sum_{s'} p_{a,s\to s'}(r_{s,a} + \gamma \max_{a'} Q_{s',a'})$$

3: Repeat from step 2

This is not practical at all. Instead we aim for an approximate solution, since a good policy does not take too long to achieve the goal (about 10 millions iterations, that is smaller than infinity).

In our case, there are limitations to this method. We do not know the transition probabilities for the actions and rewards matrix. In the real world setting, the market shifts according to the actions of buying and selling everywhere, that is a joint action of millions of traders (both human and electronic). In practice, we observe the state, decide on an action and only then we get the next observation and reward for the transition. Value iteration methods perform updates of each state value with a Bellman approximation. Q-value methods do basically the same, while storing values for every state and action. In Bellman's update, we need the reward for every transition and the probability of this transition. Supposing a discrete action space, which is the case for the trading problem, an obvious problem arises here: the count of environment states and the ability to iterate over them. If we look at the scalability of the Value-iteration approach (*i.e.* how many states could easily be iterated over in every loop), the memory

required for value tables does not seem like an insurmountable problem. The Value-iteration method wants to iterate over all of them just in case. Our environment, as it will be described later, is not very complex to iterate over all the state set. However, we do not know how the market shifts in reality, nor all of the elements contributing to that shifting. Our vision is too narrow and simply applying the classical Value-iteration approach is not enough to solve our problem.

The only accessible information in practice for us is the history from the agent's interaction with the environment. Therefore, we need to use this experience as estimation for the probability of transition and the reward we get. An agent has to try to learn the optimal policy from its history of interaction with the environment. This is done through the **off-policy** temporal difference control algorithm known as **Q-learning**. Q-learning learns an optimal policy no matter which policy the agent is actually following (*i.e.* which action a it selects for any state s) as long as there is no bound on the number of times it tries an action in any state (*i.e.* it does not always do the same subset of actions in a state). Because it learns an optimal policy no matter which policy it is carrying out, it is called an off-policy method. A one-step Q-learning is defined by:

$$\text{New}Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma \max Q'(s',a') - Q(s,a)] \tag{2.14}$$

The learned action-value function Q directly approximates Q^*, the optimal action-value function, independent of the policy being followed. The policy still has an effect, in the fact that it determines which state-action pairs are visited and updated. However, all that is required for convergence is that all pairs continue to be updated.

Deep Q-Networks

Even though the Q-learning method solves the issue with iteration over the full set of states, it still struggles with situations where the count of the observable set of states is very large. In the case of the trading environment, this count is not very large. But taking in mind the complexity of the financial markets, even a price change is important and costly. Both the trend and the direction of the price are important, as well as the change in price level. However, these are only low-level indicators on price movements. We do not take into account the more sophisticated metrics that combine those elements to get more insight. There are a lot of them and traders usually choose only a subset to focus on. We need to decide what ranges of parameters are important to distinguish as different states and what ranges could be clustered together. A solution to this is to map both the state and the action onto a value using a nonlinear representation, *i.e.* a regression. One of the most popular options for regression is to use a neural network. Neural networks are a means of doing machine learning, in which a computer learns to perform some task by analyzing training examples. Modeled loosely on the human brain, a neural network consists of thousands (or even millions) of simple processing nodes that are densely interconnected.

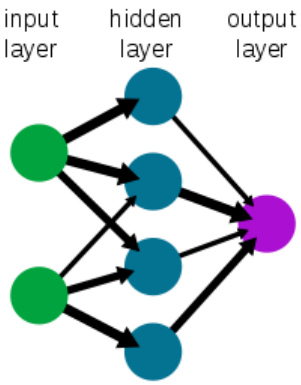

Figure 2.2: A simple neural network

To each of its incoming connections, a node will assign a number known as a "weight." When the network is active, the node receives a different data item — a different number — over each of its connections and multiplies it by the associated weight. It then adds the resulting products together, yielding a single number. When a neural network is being trained, all of its weights and thresholds are initially set to random values. Training data is fed to the input layer and it passes through the succeeding layers, getting multiplied and added together in complex ways, until it finally arrives, radically transformed, at the output layer.

The Q-learning algorithm would be:

Algorithm 3 Deep Q-learning

1: Initialize $Q(s, a)$ with some initial approximation
2: By interaction with the environment, obtain experience (s, a, r, s')
3: Calculate loss: $L = (Q_{s,a} - r)^2$ if episode has ended or

$$L = (Q_{s,a} - (r + \gamma \max_{a' \in A} Q_{s',a'}))^2 \quad \text{otherwise}$$

4: Update $Q(s, a)$ using stochastic gradient descent algorithm by minimizing the loss with respect to the model parameters
5: Repeat from step 2 until convergence

This version of the algorithm does not solve our problem yet. The data to train on is built from the interaction with the environment. In simple environments we could take random actions, but trading environments require a more sophisticated approach. We can use the Q-function approximation as a source of behaviour. If the representation of Q is good, then the experience received from the environment will show the agent relevant data to train on. However, sometimes this approximation is insufficient (at the beginning of the training for example). In such cases, the agent can be stuck with bad actions for some states without ever trying to behave differently. The agent needs to explore the environment to build a complete (and good!) picture of transitions and action outcomes. At the same time, it should not waste time by

randomly trying actions it already tried and learned their outcomes. This is a perfect illustration of the exploration-exploitation trade-off. Random behaviour in our case is good at the beginning of training when the Q-approximation is bad; it allows us to collect uniformly distributed information about the environment states. As the training progresses, random behaviour alone becomes inefficient and we would like to use the Q-approximation to decide how to act (*i.e.* learn from our experience). The Epsilon-greedy method switches between random and Q policy using the probability hyperparameter ϵ [15]. Varying ϵ allows us to select the ratio of random actions to be chosen. The usual practice is to start with $\epsilon = 1.0$ (100% random actions) and progressively drop it to a small value such as 5% of random actions. An epsilon-greedy method helps both to explore the environment in the beginning and to stick to a good policy at the end of the training.

There are still a few other practical issues for our problem, starting with the stochastic gradient descent optimization (SGD) used in the algorithm. We are trying to approximate a complex, nonlinear function $Q(s, a)$ with a neural network. To do this, we calculate targets for this function using the Bellman equation and then pretend we have a supervised learning problem at hand. A fundamental requirement for SGD optimization is that the training data should be independent and identically distributed (*i.e.* each variable has the same probability distribution as the others and all are mutually independent). This is not our case. Our time series is a minute data. The samples are not independent as they are very close to each other within the same episode; usually we will have the close price of an observation equal to the open price of the next observation. Another case is a steady price over several minutes as well. The basic feature of the financial time series is a high frequency of individual values. Nonsystematic factors can intensely influence the dynamism of these time series; the result is relatively high volatility which usually changes through time [16], thus they are not

identically distributed either. To deal with this, we use a replay buffer of the past experience and sample training data from it instead of just using the latest experience. The learning phase is logically separate from gaining experience, and based on taking random samples from the experience data. We want to interleave the two processes (acting and learning) because improving the policy will lead to different behaviour that should explore actions closer to optimal ones, and we want to learn from these. When adding new data to the buffer, we push the oldest experience out of it. This allows us to train on more-or-less independent data, but data will still be fresh enough to train on.

A second issue is bootstrapping, which is closely related to the distribution of the data as well. The Bellman equation provides the value of $Q(s, a)$ via $Q(s', a')$ where s and s' have both only one step between them. This makes them very similar and it is a hard task for neural networks to distinguish between them. When we train our agent, we update the weights accordingly to the temporal difference (TD) error, which is the difference between the maximum possible value for the next state and the current prediction of the Q-value. But the same weights apply to both the target and the predicted value. We move the output closer to the target, but we also move the target. Thus, we end up chasing the target and we get a highly oscillated training process. To stabilize training, we use a target network where we keep a copy of the network and use it for the $Q(s', a')$ value in the Bellman equation. This network is synchronized with the main network only periodically, for example once in N steps (usually quite a larger hyperparameter, such as 10k training iterations).

Additionally, the Markov decision process formalism that is the foundation of reinforcement learning methods. The environment has to obey the markov property: the current observation from the environment is all we need to act optimally. Financial markets are partially observable, which means we do not see other traders' portfolios,

therefore we do not know who chooses action A and who performs action B, etc. In theory, this setting is called Partially Observable MDP. To push our environment to the MDP domain, we cheat by maintaining several observations from the past and use them as a state, or by using technical indicators about variations in price over a past period of time within the current state. In the first case, the agent learns the dynamics of the current state, for instance, to get the change in price and its direction. In the second case, this information is given and the agent learns which are more relevant than others.

Deep Q-Networks (DQN) is the first deep reinforcement learning method proposed by DeepMind [17]. ϵ-greedy, replay buffers and target networks are improvements that allowed them to successfully train a DQN on a set of 49 Atari games. The original work without target network was published in 2013 where they used seven games to test. Later in 2015, an article [18] was published where they tested their methods with 49 different games. The algorithm for DQN is:

Algorithm 4 Deep Q-learning updated

1: Initialize $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights, $\epsilon \leftarrow 1.0$ and empty replay buffer
2: With probability ϵ, select a random action a otherwise $a = \arg\max_a Q_{s,a}$
3: Execute action a in emulator and observe reward r and the next state s'
4: Store transition (s, a, r, s') in replay buffer
5: For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step or
$$y = r + \gamma \max_{a' \in A} \hat{Q}_{s',a'} \quad \text{otherwise}$$
6: Calculate loss: $L = (Q_{s,a} - r)^2$
7: Update $Q(s, a)$ using stochastic gradient descent algorithm by minimizing the loss with respect to the model parameters
8: Every N steps copy weights from Q to $\hat{Q}$
9: Repeat from step 2 until convergence

The final piece to build our trading agent is to improve the architecture of the network. If for some reason the network overestimates a Q value for an action, that

action will be chosen as the go-to action for the next step and the same overestimated value will be used as a target value. In other words, there is no way to evaluate if the action with the max value is actually the best action. This would not be a problem if all actions were always overestimated equally, but this is not the case. If certain suboptimal actions were regularly given higher Q-values than optimal actions, the agent would have a hard time ever learning the ideal policy. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions. To do this, we use *Dueling DQNs* [19]. In this kind of network, $Q(s, a)$ can be divided into two quantities to approximate: the value of the state $V(s)$ and the advantage of actions in this state $A(s, a)$. This practical solution simplifies the relationship between $Q(s, a)$ and $V(s)$ to:

$$Q(s, a) = V(s) + A(s, a) \tag{2.15}$$

We interpret $A(s, a)$ as how much extra reward we can get by performing the action a in the state s. Here, *extra* means that the choice of one action over another can cost a lot of the total reward. The advantage could be positive or negative. Furthermore, the difference between a basic DQN and a Dueling DQN can be visualized as:

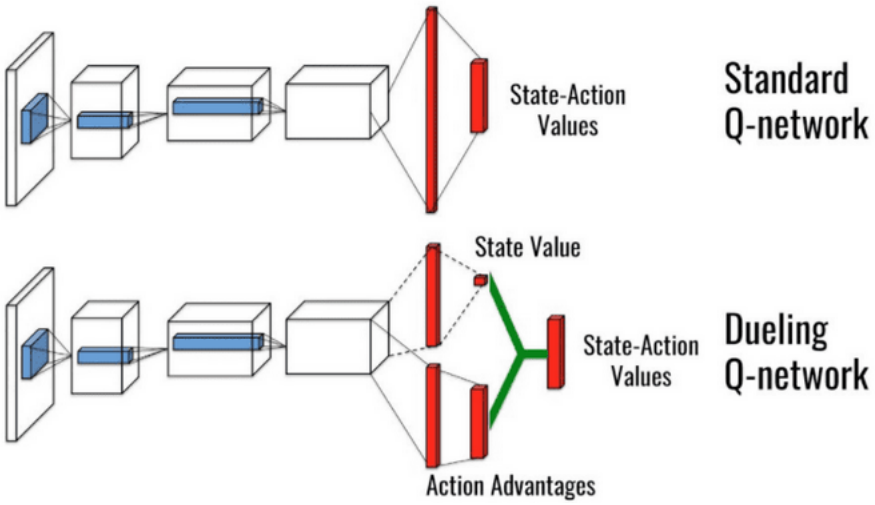

Figure 2.3: Basic DQN (top) - Dueling DQN (bottom)

The basic DQN (top) takes features from the convolution layer and, using fully-connected layers, transforms them into a vector of Q-values, one for each action. The Dueling DQN (bottom) takes convolution features and processes them using two different paths: the first one is responsible for $V(s)$ prediction and the second one predicts individual advantage values which have the same dimension as Q-values. There is a constraint to be aware of: the mean value of the advantage of any state has to be zero. Because the predictions are made separately, nothing prevents the network from making wrong predictions on one path and rebalancing on the other solely. This constraint helps prevent such cases. The solution proposed by Wang *et. al.* [19] is to subtract from the Q-expression in the network the mean value of the advantage which effectively pulls the mean of the advantage to zero. This dueling network architecture brings better training stability and faster convergence. The implementation is discussed in Chapter 3. In the next section, we compare the classical approaches of building algorithmic traders to a reinforcement learning based approach.

2.2 Development of trading strategies

A vast majority of Algorithmic trading comprises Statistical arbitrage / Relative Value strategies which are mostly based on convergence to mean, where the mean is derived from a randomly chosen sample of historical data. Algorithmic trading primarily has two components: policy and mechanism. The policy is chosen by the traders and the mechanism is implemented by the machines. We want to build a trading system which has cognitive properties that can discover a long term strategy through training in various stochastic environments. Let us compare a typical development of algorithmic trading and what a reinforcement learning approach would look like.

2.2.1 Typical development of strategy

In classic cases, the strategy boils down to the following steps:

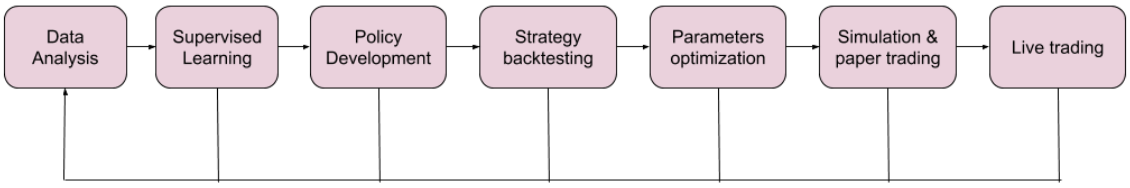

Figure 2.4: Typical strategy development [4]

Data Analysis

An exploratory analysis of the data is carried out to find trading opportunities. We can look at different graphs, calculate data statistics, etc. The result of this step is an *idea* for a trading strategy that needs to be validated. In other words, we decide on the strategy paradigm.

Supervised Learning

If necessary, one or more supervised learning models can be formed to predict the amount of interest needed to operate the strategy. The most common is price forecast.

Policy development

A rule-based policy is then created that determines the actions to be taken based on the current state of the market and the results of the supervised models. It should be noted that this strategy may also have parameters (such as decision thresholds) which must be optimized. This optimization is done later.

Strategy backtesting

A simulator is used to test an initial version of the strategy against a set of historical data. The simulator can take into account elements such as the liquidity, network latencies, fees, etc. If the strategy works reasonably well in backtesting, we follow through to optimize the parameters.

Parameters optimization

A search, such as a grid-search, can now be performed on the possible values of the policy parameters such as thresholds or coefficients, again using the simulator and a set of historical data. Here, overfitting to historical data is a big risk, and one must be careful using appropriate validation and test sets.

Simulation and trading on paper

The simulation is performed on the new market data, in real time before the strategy is put online. This is called paper trading and helps prevent overfitting. Only if the

strategy is successful in paper trading it is deployed in a real environment.

Live trading

The strategy now runs live on an exchange.

This is a complex process. This may vary slightly depending on the company or researcher, but this usually happens when new business strategies are developed. Yet this is not a very effective process for many reasons. First, the iteration cycles are slow. The steps Data Analysis, Supervised Learning and Policy Development are largely based on intuition and it is unclear if the chosen strategy will work until the Backtesting and Optimization steps are performed and optimized, which may require us to start over at zero. In fact, each step carries a risk of failure that could force us to start over. Furthermore, the simulation arrives too late. Environmental factors such as latencies, fees and liquidity are not explicitly considered until the Backtesting stage. These elements should directly influence the development of the strategy or the parameters of the model. They are elements that have an effect on the actions to take at a given state of the market.

Additionally, policies are developed independently of the supervised models, even though the two interact closely. Supervised predictions provide a contribution to policies. It makes more sense to optimize them together. The policies are simple too; they are limited to what humans can imagine. The financial data is complex and large, and there are a lot of patterns still to be found.

Lastly, the optimization of the parameters is inefficient. For example, suppose we want to optimize a combination of profit and risk and we want to find parameters that yield a high *Sharpe ratio*. Instead of using an efficient gradient-based approach, we perform an inefficient grid-search and hope to find something good (without overfitting).

A Reinforcement Learning approach is different and potentially solves some of these problems.

2.2.2 Reinforcement Learning based strategy

The problem we are trying to solve is to see how feasible it is to model trading as a reinforcement learning problem. But let us explain why we want to use it rather than supervised techniques. Developing trading strategies using RL looks like this:

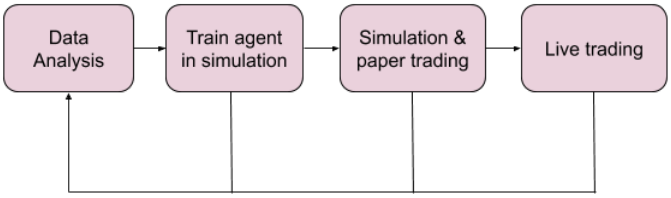

Figure 2.5: Reinforcement Learning based strategy

Train agent in simulation

This part is where the agent learns. This is done in a simulation environment on historical data to develop the policy. Previously, we needed separate stages of backtesting and parameter optimization because our strategies hardly take into account environmental factors such as the liquidity, fee structures and latencies during the supervised approach. It is not uncommon to find a strategy, but to discover later that it does not work, perhaps because the latency times are too high and the market is changing too fast so that we can not get the expected transactions. Since reinforcement learning agents are trained in simulation and the simulation can be as complex as desired, taking into account latency, liquidity and costs, we no longer have this problem. Bypassing environmental limits is part of the optimization process. For example, if we simulate latency in the learning environment and this results in an error by the agent, the agent

will receive a negative reward, which will force it to bypass the latencies. Indeed, by building an increasingly complex simulation environment that models the real world, we can train very sophisticated agents who learn to take into account environmental constraints.

This workflow has other advantages over the traditional process. In the traditional approach to strategy development, we need to go through several stages, a pipeline, before moving on to the metrics we really care about. For example, if we want to find a strategy with a *maximum reduction* of 20%, we must form a supervised model, develop a policy based rules, analyze the policy and optimize its hyperparameters to finally evaluate its performance by simulation. Reinforcement learning allows end-to-end optimization and maximizes rewards (potentially delayed). By adding a term to the reward function, one can for example directly optimize for this reduction, without having to go through separate steps. For example, one could give a significant negative reward each time a loss of more than 20% occurs, forcing the agent to look for a different policy. Of course, we can combine the maximum reduction with many other measures that interest us. It is not only easier, but also a much more powerful model.

Another benefit is policy learning. Instead of needing to code a rule-based policy manually, reinforcement learning directly learns an optimal policy. We do not need to specify rules and thresholds such as *buy when we are more than 80% sure the market will progress*. This is part of the RL policy, which optimizes the metrics we care about. This amounts to removing a complete step in the strategy development process. Since the policies can be parameterized by a complex model, such as deep neural networks, we can learn more complex and powerful policies than the human trader could possibly propose. As we have seen above, policies implicitly take into account parameters such as risk, if it is something we optimize.

Intuitively, some strategies and policies will work better in some market environ-

ments than others. For example, a strategy may work well in a bearish environment, but lose money in a bullish environment. This is partly due to the simplistic nature of the policy, which does not have a parameter setting powerful enough to learn to adapt to changing market conditions. The agent can learn to adapt to various market conditions by seeing them in historical data, as it is trained over a long period of time and has sufficient memory. This allows it to be much more robust in the face of changing markets. In fact, we can optimize it directly to make it resistant to changing market conditions, by putting appropriate penalties in the reward function.

In the next section, we present the creation and implementation of trading as a reinforcement learning problem.

Chapter 3

Methodology

3.1 Modeling

There are a lot of challenges in building a good representation of the stock market's environment. Financial markets are quite complex and contain a lot of information. The basic elements needed to formulate trading as a Reinforcement Learning problem are the observation of the environment, the action set and the reward system. Throughout this study, we aim to model and implement the elements shown in the following diagram:

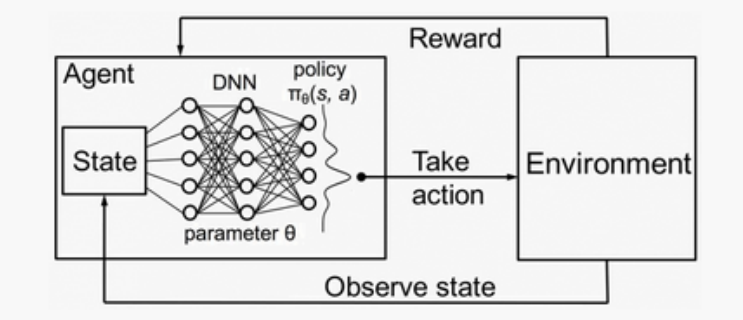

Figure 3.1: Deep Reinforcement Learning diagram

The main challenge is deciding what the learner sees as we can not really feed it all

possible information on the market's actual state. This will be discussed in details in the next section (3.2). In the meantime, the rest of the elements needed to construct the problem are the same, no matter what observations the agent takes.

Action set

The agent can take any of the following actions at each step:

- **Hold**: Skip the actual state without taking actions *i.e.* do nothing
- **Buy a share**: If not holding one already, buy a share and pay the commission fee (0.3% of the current price for Coinbase Pro, a US based digital currency exchange)
- **Close position**: If holding a share, sell it by paying the commission fee, otherwise nothing happens

A step done by the agent is as follows: an action is chosen, either randomly or from experience, the reward is received, a done flag is given, and the new observation is set. Here we make the assumption that there is no *price slippage*, which means the trade order is executed immediately at the current close price and not on a different price which would be the case in a real world setting.

Reward function

The reward is the feedback by which we measure the success or failure of an agent's actions. Therefore it is critical to design this function in a way that is significant for the problem at hand. The agent's judgment of its actions is based on this function, and one can choose from several possible reward functions. Since trading, and reinforcement learning tasks, are about maximizing the gain, the intuitive reward function would be

how much money the agent makes or loses when doing a trade. The net profit of this trade can be positive or negative. This is the reward signal. As the agent maximizes the total cumulative reward, it learns to trade profitably. This reward function is technically correct and leads to the optimal policy. We chose the reward function as:

$$r = \frac{soldPrice - boughtPrice}{boughtPrice} = \frac{close - open}{open}$$

where the open price is the price it was bought at and the close price is the close price for the current observation. An example of this as follows; if the agent bought a share at \$100 and sold it at \$110, the reward he would get is: $\frac{110-100}{100} = 0.1$. This means that the agent made a positive profit of 10% on this trade.

In the real world setting, rewards are rare because buying and selling stocks are relatively rare compared to doing nothing. In theory, the agent must learn without receiving frequent returns. In our case, we allow the agent to take random actions throughout the learning process. This randomness helps rebalance the occurrence of actions in a way. As the training progresses and ϵ decreases, the replay-buffer is really useful to get unbiased data. However, we want the agent to learn how to hold onto shares for a longer time if the market is in an up-trend, so the timeline for executing the hold action is something the agent needs to learn too.

Finally, remember that the goal is to maximize the gain. The agent needs to maximize the sum of the profits it makes from all trades. The reward function presented above does not take into account the buying action. In our case, the agent does not hold a portfolio (a grouping of financial assets as well as their fund counterparts, *i.e.* cash, Bitcoin). We make the assumption that the agent is capable of buying at all times, in other words that it has the money for the trade at all times. We limit it to only a single share trades though to avoid confusion. This is a very simplistic approach

compared to the real world where traders have limitations. However, the dynamics of the problem stay the same. The agent pays the 0.3% commission fee when buying. Prices do not vary much within close episodes. The reward it would get, even from successful trades, would be around the same magnitude.

In reality, a trader may want to minimize the risks. A strategy with a slightly lower return but a significantly lower volatility is preferable to a very volatile strategy but only slightly more profitable. Using the Sharpe ratio is a simple way to take risk into account, but there are a lot of other indicators. We can also consider the maximum reduction that could be particularly interesting with the behavior of Bitcoin, since this currency has proved since the end of 2017 that an investor could become a millionaire in one month, and return to normal the following month. Roughly, one can imagine a wide range of complex reward functions that make a trade-off between profit and risk. We will use the reward function presented above for the sake of convenience, since this is still an early stage of modeling a reliable trading environment.

State set

This set is the most complicated part of the modeling process. In the case of a trade in the market, we do not observe the complete state of the environment. There is a lot of information and traders choose to follow those that are more relevant to their strategies. By the time one observes carefully the current state of the market and studies its elements, the market would already have changed its state. The elements we take into account are presented in details in the next section (3.2), where we try to ensure that the Markov property is respected. However, the general idea is to have observations about price fluctuations, either by recording several past observations as one, or by using metrics of technical analysis that hold information about the past state but are more challenging to interpret for the agent.

Agent

The agent is the element trading on the environment. At a given time step t, it observes the state of the environment, selects the action from the action set and receives the reward. It has a memory and a brain. The memory stores its experiences of trades and it becomes larger as the training progresses. The Agent will store this information through iterations of exploration and exploitation. The memory contains a list in the format: (state, action, reward, next_state). The brain is the Neural Network which will train from the memory (*i.e.* past experiences). Given the current state as input, it will predict the next optimal action. As we discussed in section (2.1.3), the brain will be divided in two networks, one to compute state values and the other one to compute action advantages.

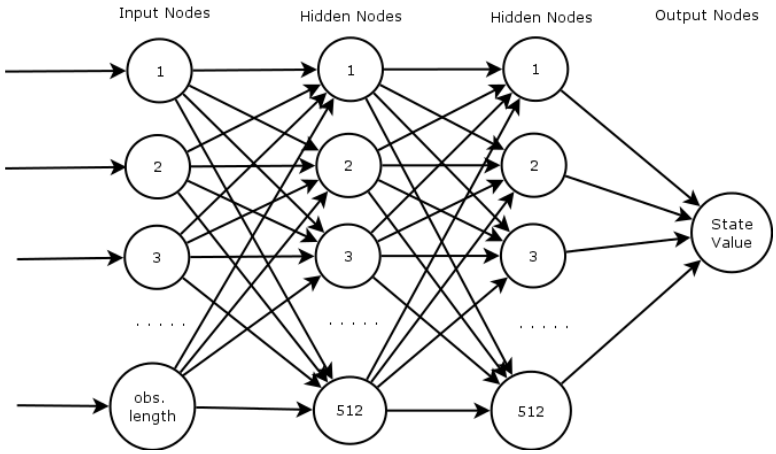

Figure 3.2: First network diagram - State value computing

Obs.length stands for observations length: how many observed values we have in the state. This number depends on the data representations that we will be using.

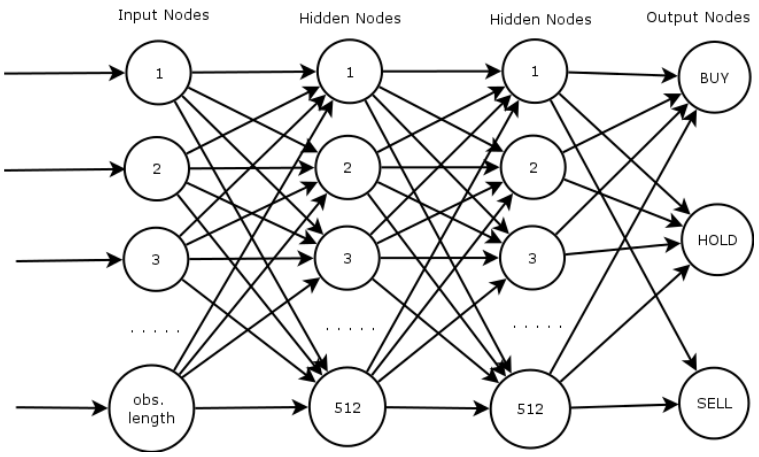

Figure 3.3: Second network diagram - Action advantage computing

The two computed values are afterwards combined back into a single Q-value. This value describes how useful the chosen action is in gaining some future reward.

3.2 Data

It is necessary to take into account the nature and the form of the data which one possesses and to try to keep the problem as a Markov Decision Process (*i.e.* ensure the Markov property holds). The recoverable data from any financial data platform all have the same format and are summarized in the following graph:

Figure 3.4: Bitcoin prices August 2017 through August 2018

This graph, retrieved from Quandl, shows Bitcoin's price fluctuation between August 2017 and August 2018. We distinguish the opening (open) and closing prices (Close) as well as the highest prices (High) and the lowest (Low) for each day of this period, and the total volume (Volume) of executed orders during the tick period for buy and sell actions. The left vertical axis denotes the price in USD. The right vertical axis refers to the traded volume per 24h.

Unix Timestamp	Date	Symbol	Open	High	Low	Close	Volume
1514764740	2017-12-31 23:59:00	BTCUSD	13825	13825	13804.7	13820.3	2.61072
1514764680	2017-12-31 23:58:00	BTCUSD	13815.4	13825	13815.4	13825	3.54759
1514764620	2017-12-31 23:57:00	BTCUSD	13775	13815.4	13775	13815.4	18.4373
1514764560	2017-12-31 23:56:00	BTCUSD	13771	13775	13771	13775	7.53843

Figure 3.5: Bitcoin prices per minute at the end of 2017

We now have to decide how to represent these prices in our environment observations. We would like our agent to be independent on actual price values and take into account relative movement such as *Bitcoin has grown 2% during the last few observations* or *Bitcoin has lost 1% over the last period*. This helps the agent to discover the patterns in price movements. To achieve this, we present two different approaches. The first one is representing high, low, and close prices as percentages to the open price. This is a low-level data representation and it can be very noisy as it includes lots of small price movements. The second one is to add to the observation's technical indicators, that are basically more sophisticated metrics that provide insights on market signals.

Bear in mind that the Markov property ensures that every state is self-contained and therefore the learner can behave optimally by observing its current state. However, the trader needs to observe how the stock moved recently: in which direction and by how much. The current price alone is not enough to draw these information. The second approach does not require further adjustments as the information is contained within the technical indicators themselves. For the first approach, however, we have to group observations together. The agent would observe the last N instances as one state of the environment.

3.2.1 First approach: Prices as percentage of variation

In this case we group multiple price observations as one observation of the state. We will call the observations bars, composed of {Open, High, Low, Close, Volume} elements. The observed state vector looks like:

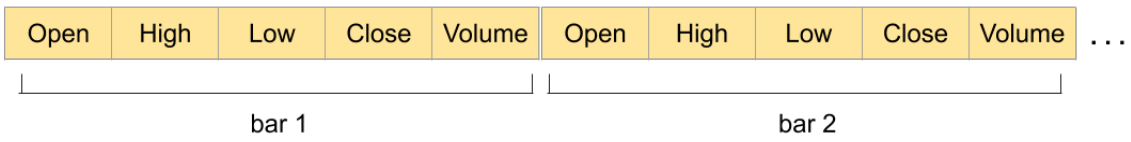

Figure 3.6: Data as one array with multiple bars

To achieve this, we first organize the different fields as arrays themselves. This representation is convenient for the network architecture. For this, we perform a rotation:

0	float32	(305559,)	[13825.	13815.37	13775.	...	974.55	970.	974.55]
1	float32	(305559,)	[13825.	13825.	13815.37	...	974.55	974.55	974.55]
2	float32	(305559,)	[13804.68	13815.37	13775.	...	974.5	970.	970.]
3	float32	(305559,)	[13820.26	13825.	13815.37	...	974.5	974.55	970.]
4	float32	(305559,)	[2.6107187e+00	3.5475934e+00	1.8437304e+01	...	9.2993602e-03	5.1968802	...

Figure 3.7: Data rotated

The first row describes all the opening prices of the year 2017. The second row refers to the high prices, third row to low, fourth to close, and fifth describes the volume. The circled elements (column in red) are what the first bar would contain. However, as we explained before, we are more interested in learning the patterns and therefore represent the High, Low, Close as percentages to the opening price:

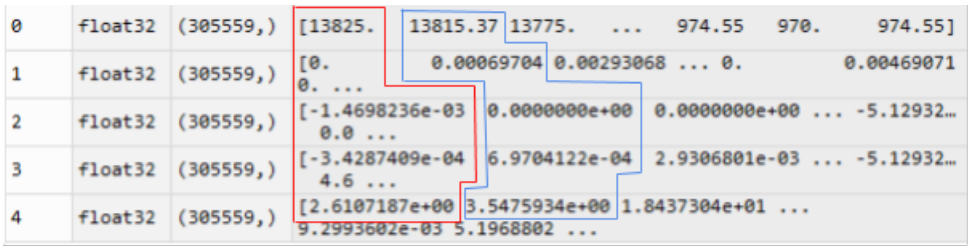

Figure 3.8: Rotated data as relatives to opening price

The first bar contains the elements of the column in red, the second bar contains the column in blue, and so on. This table forms a *price* tuple.

3.2.2 Second approach: Prices with technical indicators

In this case we leave the prices as they are, {Open, High, Low, Close, Volume}, and we add to them 5 technical indicators: {r, r_1, r_2, rsi, atr}:

- r: return over the last timeperiod
- r_1: return over the last two timeperiods (lagged return)
- r_2: return over the last three timeperiods (lagged return)
- rsi: Relative Strength Index, shows how strongly a stock is moving in its current direction over the last fourteen timeperiods
- atr: Average True Range, measures a stock's volatility over the last fourteen timeperiods

There are roughly more than 30 technical indicators traders use for market analysis. Each of them is more suitable for some type of strategy than others. The indicators can be classified into four groups: trend (direction, distribution), oscillator (momentum), volatility and volume. Since we are more interested in price patterns, we chose those

five elements. RSI is a momentum indicator while ATR is a volatility indicator. As an illustrative example of how these indicators might work, the agent may learn that rising prices along with steady volume is a bullish sign and adjust its weights so that it invests longer (holds longer) onto the share.

The observed state vector looks like:

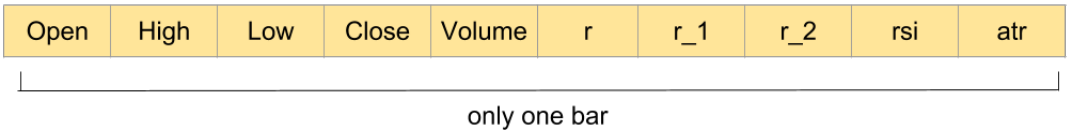

Figure 3.9: Data as one array with one bar

We use the same representation as the first approach for convenience for the network architecture.

0	float32	(525600,)	[13825. 13815.37 13775. ... 974.55 974.55 974.55]
1	float32	(525600,)	[13825. 13825. 13815.37 ... 974.55 974.55 974.55]
2	float32	(525600,)	[13804.68 13815.37 13775. ... 970. 974.55 974.55]
3	float32	(525600,)	[13820.26 13825. 13815.37 ... 970. 974.55 974.55]
4	float32	(525600,)	[2.6107187 3.5475934 18.437304 ... 0.41767928 0. 0.]
5	float32	(525600,)	[0. 0.00034292 -0.00069681 ... 0. 0.00467975 0. ...
6	float32	(525600,)	[0. 0. -0.00035389 ... 0. 0.00467975 0.0 ...
7	float32	(525600,)	[0. 0. 0. ... 0. 0.00467975 0.00467975 ...
8	float32	(525600,)	[0. 0. 0. ... 3.7102466 4.038229 4.038229]
9	float32	(525600,)	[0. 0. 0. ... 0.77680665 0.8519419 0.7778339 ...

Figure 3.10: Data with technical indicators

Similarly, the first row describes all the opening prices, the second refers to the high prices, the third to low, the fourth to close, and fifth describes the volume. The following rows refer to the technical indicators. The observations are done in the same

way. The table forms a *price* tuple.

We will use 2017 Bitcoin prices per minute data as training and testing sets, and 2018 minute prices as validation data for both approaches.

In the next section, we discuss how the elements of this reinforcement learning problem are implemented.

Chapter 4

Results & Discussion

The two approaches are compared to a baseline where ϵ does not decrease. In such case, we behave randomly at all times and the length of episodes (holding duration before selling) is constant. The reason we use this baseline is to demonstrate that it is worthwhile to use an RL agent as a trader and not simply a random one that is easier to implement and execute. To do this, we have to answer the following questions:

- Does the RL agent generally achieve higher performance than a random agent? This is referred to as the algorithm effect.
- Is the influence of training on the performance dependant on the algorithm? This is called the interaction effect.

First, we run the approaches through statistical tests for performance curves to answer these questions. Second, we evaluate them on previously unseen data to see if the strategies do well on different market situations.

4.1 First approach results: Prices as percentages of variation

The following chart shows the average length of the episode for the last 100 episodes:

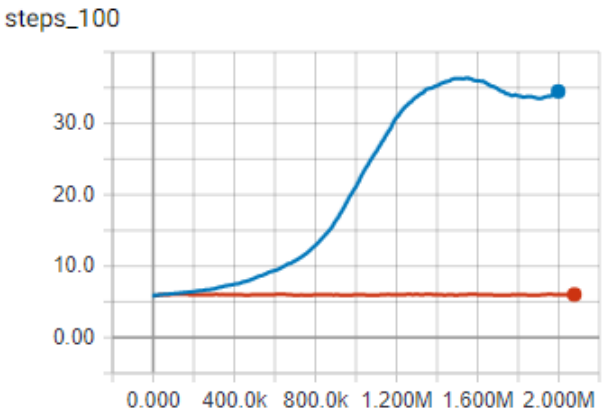

Figure 4.1: Average length of episode over last 100 episodes for agent (blue) and baseline (red)

This shows that the length increased over the training time. The episode lasted 5 bars at the beginning and grew to 36 bars after 1.5 million steps and then decreased and adjusted to 35. This means that the agent learned to hold the share for a longer time to increase the final profit over the first 1.5 million steps, then decreased that time a little bit, which means it learned not to hold too long either.

The next chart shows the average reward of the episode over the last 100 episodes:

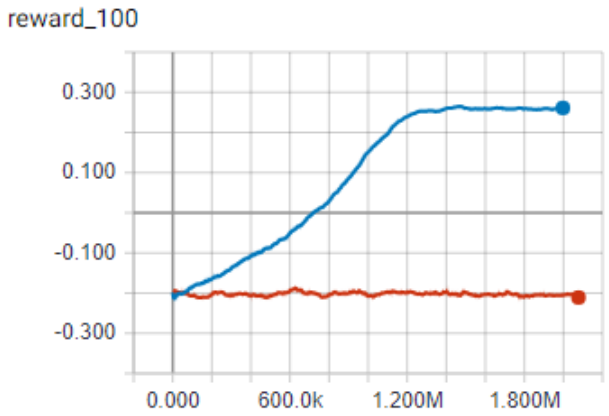

Figure 4.2: Average reward over last 100 episodes for agent (blue) and baseline (red)

This chart shows that the agent was able to learn how to make positive profit overtime. Since the commission fee (0.1% of the price) is paid after each trade, either buying or selling, a random action has a value of -0.2% reward. The agent still learned how to get positive reward overtime. It began at -20% and ended at 25% profit in 2017.

The apparent differences between these curves (baseline Vs. our agent) are statistically significant according to a randomized ANOVA test for performance curves, with $p < 0.001$ for both algorithm and interaction effects [20]. We can therefore say with 95% confidence that the mean performances of our agent and the baseline are not the same, and that the relationship between training and performance depends on the algorithm.

Now, let us evaluate how profitable the agent will be in the future. To check the strategy of the agent, we need to validate its training on previously unseen data for a different time period. First we test it on the training data, 2017 data and then we validate it on 2018 data.

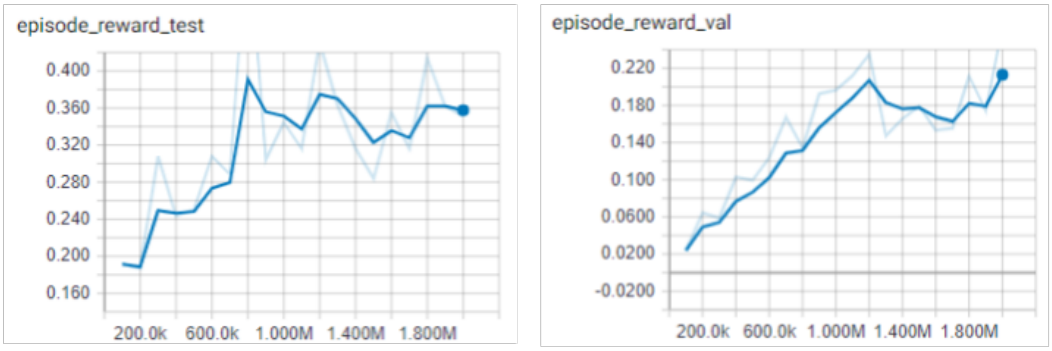

Figure 4.3: Reward per episode on test and validation data

On the test chart, we can see that the reward is positive and mostly growing over time which confirms the same positive dynamics seen during the training. The reward starts at 19%, and reaches a maximum of 40%. On the validation chart resulting from the unseen data, the reward is also positive and growing. It does not have the same magnitude as the test rewards however, it starts at 2% and reaches a maximum of 22% in 2018. This is still very good though. It means that the agent's strategy is leading to positive reward on previously unseen data. Let us look into the length of episodes:

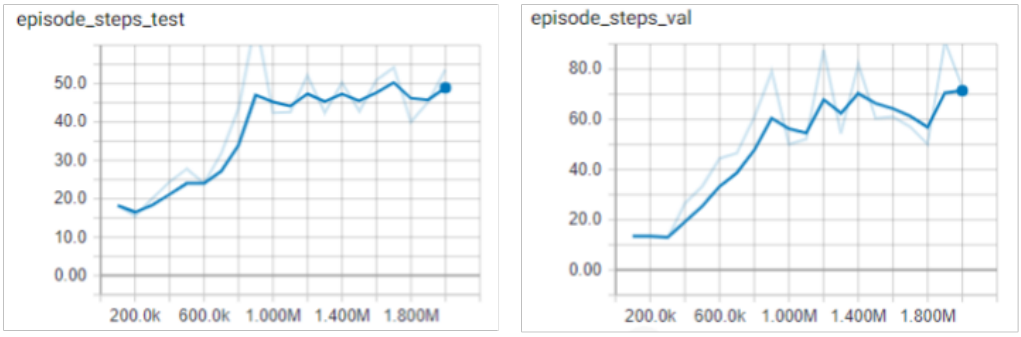

Figure 4.4: Steps per episode on test and validation data

On the test chart, the length of the episode is also growing as the agent learns to hold the share longer and longer. On the validation chart, the same dynamic is present

and the agent even has longer episodes than during the testing process. The strategy is doing well both on testing and validation.

Even though our agent was capable of making positive profit facing 2018 prices, it struggles on 2019:

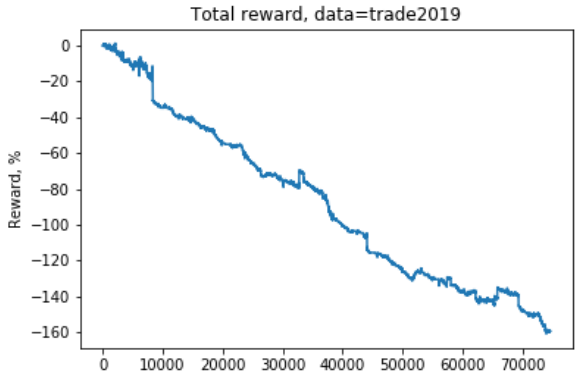

Figure 4.5: Total reward on 2019 trading

The agent lost 160% trading on the period January 1st through April 1st 2019. Note that this chart represents the evolution of the total reward on the trading period not per episode. As we have seen in section (2.2), the strategy is deployed in a real environment only if it is successful in paper trading. Evaluating the total reward is a simple and effective way to say whether or not the strategy is doing well. In this case, we can see that this strategy is not ready yet.

4.2 Second approach results: Prices with technical indicators

Let us see how this agent behaves based on the same dynamics as the first one. The average length of the episode for the last 100 episodes is visualized in the following chart:

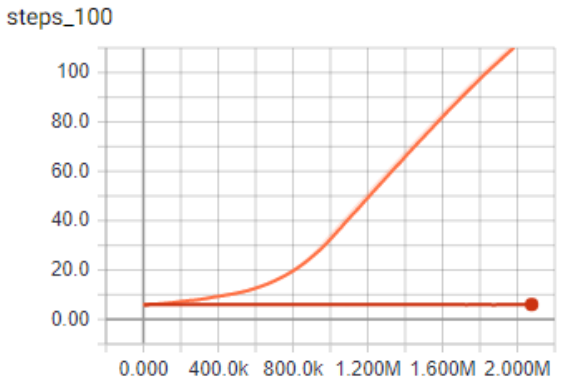

Figure 4.6: Average length of episode over last 100 episodes for agent (orange) and baseline (red)

Like the first agent, this chart shows that the length increased over the training time. The episode lasted 5 bars at the beginning and kept growing to more than 100 bars. This agent did not adjust the length of episodes, which means it did not learn not to hold for too long.

The next chart shows the average reward of the episode over the last 100 episodes:

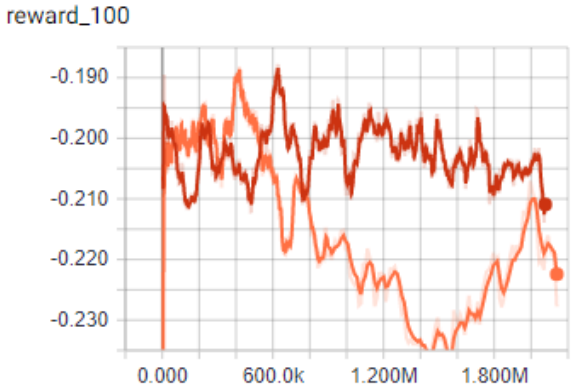

Figure 4.7: Average reward over last 100 episodes for agent (orange) and baseline (red)

This chart shows that the agent was not consistant in making profit. The average reward began at -20%, improved to -19%, fell down to -23% for a vast majority of the training period, went up to -21% and down again at -22%. This strategy is failing to make positive profit overtime like the first one, even though it places some successful trades.

The apparent differences between these curves (baseline Vs. our agent) are also statistically significant according to a randomized ANOVA test for performance curves, with $p = 0.02$ for algorithm effect and $p < 0.001$ for interaction effect. We can therefore say with 95% confidence that the mean performances of our agent and the baseline are not the same, and that the relationship between training and performance depends on the algorithm.

Let us evaluate this agent on previously unseen data for a different time period. We use the same test and validation data sets; 2017 and 2018 price data.

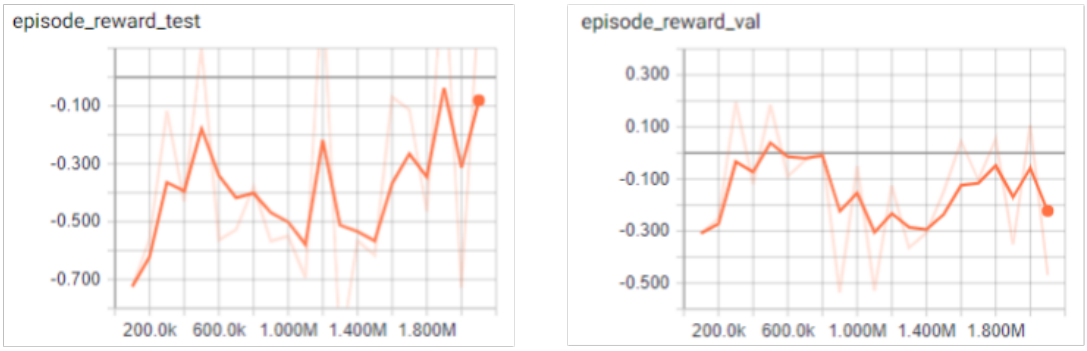

Figure 4.8: Reward per episode on test and validation data

On the test chart, the reward is still negative. As it starts at -70%, it struggles but improves significantly to -10%. On the validation chart resulting from the unseen data, the agent succeeds at going up from -30% to 0% at the beginning, drops again to -30% and improves to -10% before finishing at -22%.

Let us look into the length of episodes:

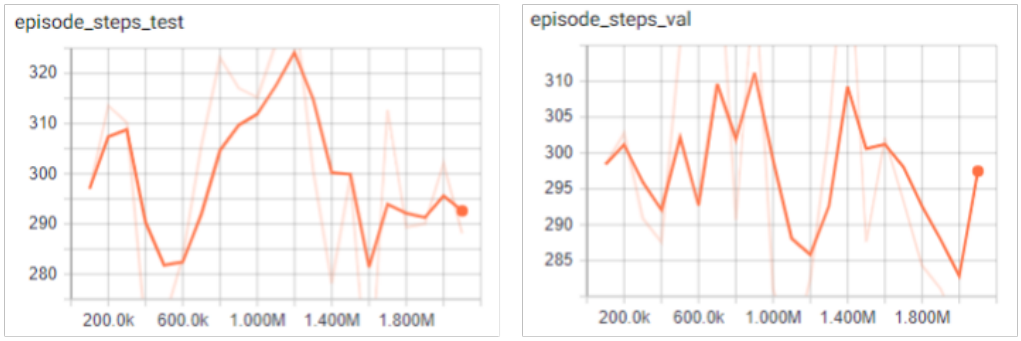

Figure 4.9: Steps per episode on test and validation data

On the test chart, the length of the episode is tremendously bigger with a maximum of 325 bars. The direction (wait longer or less) changes multiple times and is sign of hesitation. The validation chart shows the same dynamic which means the agent cannot decide on holding for longer or shorter periods of time.

If we want to try this agent on further unseen data, for example on 2019 data:

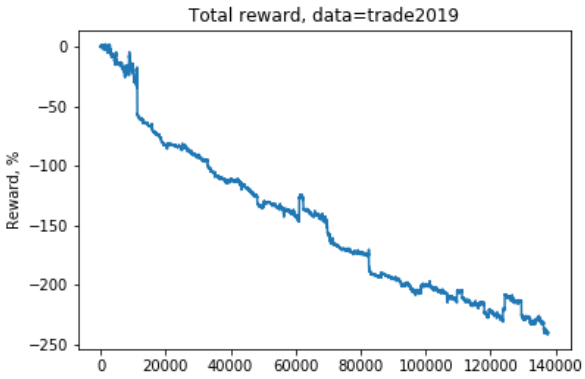

Figure 4.10: Total reward on 2019 trading

The agent lost 250% trading on the period January 1st through April 1st 2019. This strategy, like the first one, is not ready yet.

4.3 Discussion

The ANOVA test on performance curves has shown with great confidence that the RL algorithms had a strong influence on the results. The training data looks like:

Figure 4.11: BTC prices for 2017

We saw that both approaches learned to hold onto position for longer and longer as the training progressed. Seeing this data confirms that holding the shares longer is a great contributor to a good policy as the whole market was in an up-trend until November 2017.

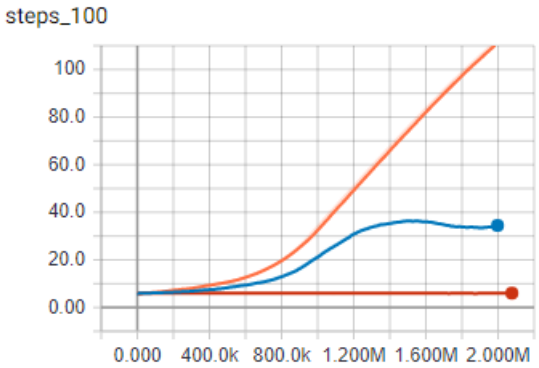

Figure 4.12: Length of episodes for first agent (orange), second agent (blue) and random agent (red)

The first agent was capable of understanding that holding the share longer increased profit. The late down-trend that comes towards the end of the time series explains the decrease in the length of episodes too. The best result for this agent was to buy and hold rather than to do shorter trades during this time frame, and this is a good strategy on 2017 data, as the price only kept increasing. We can say that this agent learned to adjust its policy and find an appropriate strategy. Note that there is not an optimal length of episode because at each time step we are unable to tell what will happen next. If the price is at a local maximum, the optimal action would be to sell now, but we only know we reached the maximum once we passed it. What traders do in the real world is the inverse strategy, they care about loss stop; when to stop holding when the value of the share is decreasing.

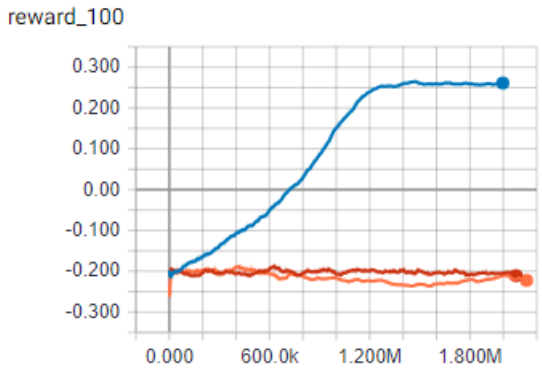

Figure 4.13: Average reward per 100 episodes for first agent (orange), second agent (blue) and random agent (red)

The second agent is holding for longer time, but it still failed to make positive profit. The reward charts looks like it behaves randomly all the time but we know that that is not the case as it has been demonstrated by the ANOVA test. In fact, according to this test, we can say that the agent adjusted its policy but ended up with an inappropriate one. There may be a lot of issues that could explain this behavior, such as the reward not being accurate enough. To demonstrate this, let us see how the

agent behaves in 2019 if there was no commission fee:

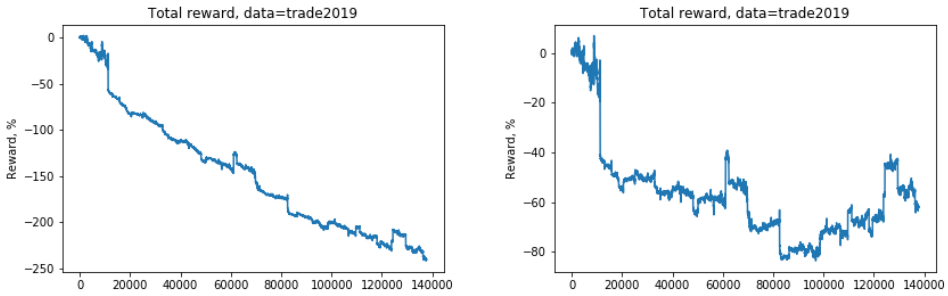

Figure 4.14: Total reward with commission fee (left) and without commission fee (right) - Second agent

Even though the agent still fails to produce positive profit, it has a slightly better performance when there is no commission fee. While it lost 250% when paying 0.1% on each trade, it lost 80% when there is no fee. This is normal since it paid less each time. However the behavior is better as we can see there is some resistance and attempts of improvement when training progresses. Since in the real world there has to be a commission fee, this does not help much. There could be several reasons why this agent is not making positive profit. From what we saw for the first agent, we can do well on some years and fail on others, with a big difference. For example:

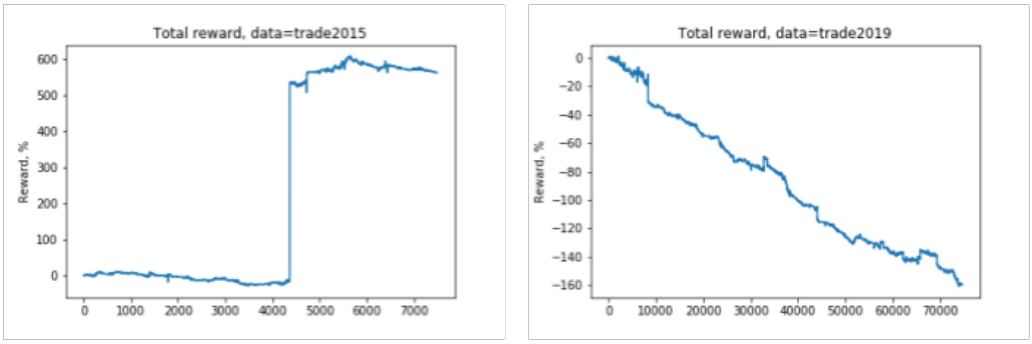

Figure 4.15: Total reward for 2015 (left) and loss for 2019 (right) - First agent

This is the same strategy applied to 2015 prices and 2019 prices. The agent used the same market understanding that allowed it to make a huge positive profit, 600% in 2015, and lose 160% in 2019. The mappings of market states to actions happened the same way in both cases, but the market does not always respond similarly. Let us look at these two different market situations:

Figure 4.16: BTC prices in 2015

- 2015:
  - First quarter: down-trend for January, initial price recovery at the beginning of February then a small but continuous up-trend through March and price drop again by April
  - Second quarter: market resistance observed May through mid-June then up-trend then down again to price at resistance level in August
  - Third quarter: 100% increase in price from September to beginning of November, 50% decrease in November and then 40% increase in December

Figure 4.17: BTC prices in 2019

- 2019: at the time of writing, data is only available until April 1st. The market is having lasting episodes of resistance each week.

The total reward chart of 2015 for the first agent shows that it had a little period of very successful trades (the jump in reward towards the 4200 steps) and the rest of the year was poor in profit. For the first quarter, if it bought at the beginning, there is no point where it would make a positive profit, hence the slightly decreasing reward below 0%. Then, the agent looks like it made the most of the up-trend in a very short time by successfully placing trades. One would say how is it possible to make 600% profit in such a short time then. Think of it this way: it learned to hold the share for about 45 minutes. Let us take the following example:

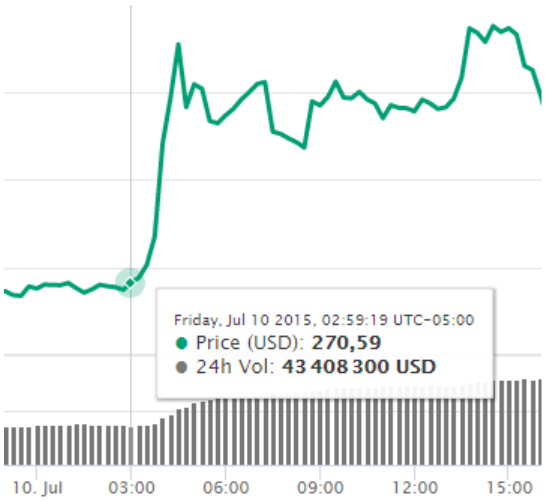

Figure 4.18: BTC prices July 10th, 2015

Suppose the agent buys a share for $270.59 at 02:59:19, July 10th, 2015. It sees the up-trend, waits for about 50 minutes and sells the share for $279.03 at 03:49:19, July 10th, 2015. That is 3.1% profit in less than an hour. Now let us suppose it waits for 5-10 minutes before buying again for $283.29 at 03:59:19. It then sells the share for $288.79 at 04:34:19, that is 1.9% profit. In 2 hours, the agent made 5% positive profit. It would take 240 *similar* hours to make 600% profit. The policy of the agent tells it to replicate that kind of state observation to the same actions to be undertaken. Keep in mind the up-trend we are in; the market is going up for the next 2 months. If we disperse those 240 hours (10 days) on the following period, the 600% profit can be achieved even in the first two weeks. Of course this is not a realistic assumption as the patterns are not the same on a minute-scale, but this gives us an idea how it is possible to make such a profit. The question here is, why did it stop making positive reward afterwards? This is not a training process; the agent applies its strategy only. It simply saw the same trend that happened at the beginning of 2015 for the rest of the year so it had the same behavior that did not generate positive income.

Now, looking at 2019 data, we see episodes of resistance everywhere with more often price drops than ups. Looking closer, one Bitcoin is no longer priced at about $300, but $3000. Since price would vary about $10 in a very active day, that would be 0.3% variation in price. In fact, resistance seems to happen even on an hourly-scale:

Figure 4.19: BTC prices January 23-25th, 2019

This resistance in price, coupled with the insignificance of price variation makes it hard for the agent to place successful trades. The agent applies its understanding of the market again, but the market is different this time. Therefore, we can say that its understanding is immature. This might be overcome if we used more training data. Indeed, 2017 data is not enough to capture all market situations. Ideally, we should train our agents on Bitcoin prices since it started to be on the market. Several years of data would ensure a broader vision of all the possible situations in the financial markets. As we said before, the stock market is wild and complex. One year of data is not enough to *see it all*.

Another surprising finding is that the first representation of data is more powerful than the second, which is counter intuitive. The first representation is just price movements, while the second one was a chance for the agent to learn complex patterns

and analysis by mixing indicators. It is definitely not perfect, but it was presupposed to be more powerful. There is a large number of ways technical indicators could be incorporated into the observation. Maybe the ones we chose were not enough and there is a better representation. Or maybe all we need is solely stock prices. Different data representations should be tested. The network architecture could be too simple for this representation as well. We might need to use more sophisticated networks. The following chapter presents some of the ideas that could be applied to our learner's brain and observations.

Chapter 5

Future work

Financial markets are large and complicated and there is useful information everywhere with no time to analyze them. The DQN method we presented in this work shows that we can indeed see trading as a Reinforcement Learning problem, as long as we have a good representation of the environment. There are a lot of elements that need to be revised in our approach in order to achieve better performance. Furthermore, there are other elements of different nature that could be added.

Broader environment observations

The results have shown that a good environment representation is critical to develop a good and lasting strategy. We need information to be more complete and there are a lot of indicators we should take into account too, such as price levels. However, this is not all we need. External news is critical, and most traders wake up before the market opens to do research on stocks news because they know that it has an observable impact on day prices. We should add a factor to our observations, a sentiment-analysis kind of factor that would classify news as bad or good and keeps looking for media updates during the trading process. This has already been tried and developers are

using Natural Language Processing techniques to do so.

There are other approaches in Deep Reinforcement Learning that could also fit in the trading environment and tackle other problems.

Actor Critic Method

This is a policy based method and is kind of more intuitive as an application to our trading problem. Value estimation is suspicious in finance, unless we define well what the returns are. If we are simply using returns or risk adjusted returns for a reward, then we have a noisy signal. The main idea behind the Actor Critic method is that we do not attempt to optimize for the choice of action; we do not ever know what the objectively correct action would have been, but rather the degree of certainty about each action, given its resulting reward and the degree of surprise about each reward, given the state. We do not use the Bellman equation anymore but a neural network to calculate state values, and optimize it just like we optimize the main action-selecting (policy, actor) neural network.

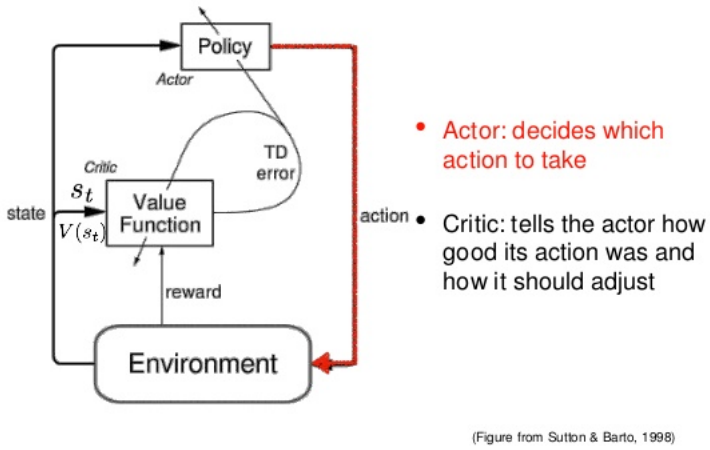

Figure 5.1: The actor-critic architecture [5]

With this kind of architecture, we could for example use a more realistic view of

a trader's portfolio and try to manage it. We could build an agent that observes a universe of n stocks and at each step decides how much of its capital to allocate to each of those n stocks. The actions would be the weights of a portfolio over n stocks and cash. Think of the actor as the person holding capital and the critic as the broker telling them what to do. This is more intuitive than our DQN method.

Conclusion

It has never been easy to make money by stock trading. It requires a lot of time and effort to answer a lot of questions and make a good trading strategy, but implementing real-time data into analysis is the difficult part. Investment, and trading in general, is an iterative process. We make our bets, learn something new and try again. During this struggling process, we improve our own decision making by constant trial-and-error. That is the reason we use Reinforcement Learning, a framework where an agent is trained to behave properly in an environment by performing actions and adapting to the results. The connection between trading psychology and the formalisms of RL are the reason we started this research. We wanted to demonstrate that trading could be seen as a reinforcement learning problem although there are a lot of challenges to this approach.

As a proof of concept, we designed and implemented a trading system for Bitcoin as trade data is readily available. We represented data in two different ways: the first aimed learning the price patterns so we represented the High, Low, Close prices as percentages to the opening price. The second implied the use of technical indicators that could be used by the agent to learn how to analyze the market and see opportunities such as rising prices along with steady volume is a bullish sign and it has to invest longer. To evaluate the efficacy of our reinforcement learning agents, we studied the dynamics after training was completed, such as learning to hold onto shares, total

profit, and how the strategy performs on previously unseen market situations. The best result for the first agent was to buy and hold rather than to do shorter trades during the year 2017. The second agent was not able to make positive profit although it learned to buy and hold. This could be explained by the inaccuracy of the design of the problem elements, such as the reward function. It can also be due to the fact that we did not train on a data set richer in different market situations. The performance of the first agent proved this point, as it did not always manage to make positive profit.

Both approaches are still very basic. Real traders use much more sophisticated tools than price variations and the few technical indicators we used. However, the work presented here shows the potential of applying deep reinforcement learning to trading. The next step is to rework the data representation and take into account real-life situations such as price slippage in order execution. We can also experiment with the network architecture and find a more powerful and faster model. There are still a lot of approaches in Reinforcement Learning to explore too. We can try a policy-based method such as an Actor Critic approach, or we can shift to risk management and try to apply categorical DQNs. Deep Reinforcement Learning applications in finance are still largely unknown. However, this is an active area of research and is showing a lot of potential at early stages.

Glossary

bearish

a condition in which securities prices fall 20 percent or more from recent highs amid widespread pessimism and negative investor sentiment 39

bullish

a condition of a financial market of a group of securities in which prices are rising or are expected to rise 39

cash equities

a type of trading executed primarily by large, institutional investors 3

decimalization

a system where security prices are quoted using a decimal format rather than fraction 2

derivative

a contract between two parties which derives its value/price from an underlying asset 3

high frequency trading

a program trading platform that uses powerful computers to transact a large number of orders in fractions of a second 2

liquidity

refers to a market's ability to allow assets to be bought and sold easily and quickly 35

price forecast

using historical price data as inputs to make informed estimates that are predictive in determining the direction of future trends 35

security

refers to any form of financial instrument, but its legal definition varies by jurisdiction

volatility

the degree of variation of a trading price series over time as measured by the standard deviation of logarithmic returns 3

volume

the amount (total number) of a security (or a given set of securities, or an entire market) that was traded during a given period of time 3

Acronyms

ATR

Average True Range

BTC

Bitcoin

DQN

Deep Q-Network

DRL

Deep Reinforcement Learning

HFT

High Frequency Trading

MDP

Markov Decision Process

NYSE

New York Stock Exchange, Wall Street

OTC

Over The Counter

RL

Reinforcement Learning

RSI

Relative Strength Index

Bibliography

[1] "What is cryptocurrency: Everything you must need to know!." https://blockgeeks.com/guides/what-is-cryptocurrency/, 2017.

[2] "Trading journal blog." https://www.trademetria.com.

[3] M. Lapan, *Deep Reinforcement Learning Hands-on*. Packt Publishing Ltd, 2019.

[4] D. Britz, "Introduction to learning to trade with reinforcement learning." http://www.wildml.com/2018/02/introduction-to-learning-to-trade-with-reinforcement-learning/, 2018.

[5] R. S.Sutton and A. G.Barto, *Reinforcement Learning: An introduction*. MIT Press, 1998.

[6] "The evolution of trading: Barter system to algo trading." https://www.quantinsti.com/blog/evolution-trading-barter-system-algo-trading, 2017.

[7] S. Satoshi, *Cryptocurrency: Beginners Bible - How You Can Make Money Trading and Investing in Cryptocurrency like Bitcoin, Ethereum and altcoins*. CreateSpace Independent Publishing Platform, 2017.

[8] A. Aziz, *How to Day Trade for a Living: A Beginner's Guide to Trading Tools and Tactics, Money Management, Discipline and Trading Psychology.* CreateSpace Independent Publishing Platform; 3 edition, 2016.

[9] Y. Li, *Deep Reinforcement Learning: An Overview.* arXiv: abs/1701.07274, 2017.

[10] ADL, "An introduction to q-learning: reinforcement learning." `https://medium.freecodecamp.org/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc`, 2018.

[11] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley Series in Probability and Statistics, Volume 414, 2009.

[12] D. Bertsekas, *Dynamic Programming and Stochastic Control, Mathematics in Science and Engineering (Book 125).* Academic Press, 1976.

[13] D. Bertsekas, *Neuro-dynamic programming.* Athena Press, 1996.

[14] W. B. Powell, *Approximate Dynamic Programming: Solving the Curses of Dimensionality.* Wiley, 2011.

[15] I. Dabbura, "Bandit algorithms: epsilon-greedy algorithm." `https://imaddabbura.github.io/post/epsilon_greedy_algorithm/`, 2018.

[16] M. A. Joseph Arlt, *Financial Time Series and Their Features.* Acta oeconomica pragensia 9: (4), str. 7-20, VŠE Praha ISSN 0572-3043., 2001.

[17] V. M. et al., *Playing Atari with Deep Reinforcement Learning.* arXiv: 1312.5602, 2013.

[18] V. M. et al., *Human-Level Control Through Deep Reinforcement Learning*. Nature, Volume 518 Pages 529-533, 2015.

[19] W. et al., *Dueling Network architectures for Deep Reinforcement Learning*. Proceedings of the 33rd International Conference on Machine Learning, Volume 48 Pages 1995-2003, New York, NY, USA, 2016.

[20] J. H. Piater, *A Randomized ANOVA Procedure for Comparing Performance Curves*. Proceedings of the 15th International Conference on Machine Learning, pp. 430–438, Madison Wisconsin, 1998.

[21] J. Francis, "Introduction to reinforcement learning and openai gym." `https://www.oreilly.com/learning/introduction-to-reinforcement-learning-and-openai-gym`, 2017.

Appendices

Coding

The implementation of the elements presented in this work respects the conventions for OpenAI Gym's *Env* class API [21]. All sections are coded in Python.
Note that the handling of price data time series is not described in this section. The purpose here is to explain the dynamics and the implementation of the elements discussed previously.

The environment

Actions are encoded as an enumerator fields.

```
class Actions(enum.Enum):
    Hold = 0
    Buy = 1
    Close = 2
```

To create the instance of our environment, we construct it directly:

```
def __init__(self, prices, bars = DEFAULT_BARS,
commission_fee = DEFAULT_COMMISSION, reset_on_close = True,
reward_on_close = False):
```

The arguments for this constructor have the following interpretations:

- *prices*: dictionary of prices mapping the quotes to the tuple, the key is the asset's name and the value is a container object that holds price data arrays
- *bars*: The number of bars in observation
- *commission_fee*: the percentage paid as a fee for the trade, on CoinbasePro it is 0.3%
- *reset_on_close*: if True, the episode stops every time the agent asks to close a position. Otherwise, the episode continues until the end of the time series
- *reward_on_close*: If True, the agent receives the reward only when closing the position. Otherwise, it receives a small reward every bar, corresponding to the price movement during that bar. It is by default set to False, because we want our agent to learn when and for how long to wait before closing a position. This way it studies the evolution of the share value and can follow trends when the price is going up or down.

Looking into the environment constructor:

```
1     assert isinstance(prices, dict)
2     self._prices = prices
3     self._state = State(bars_count, commission, reset_on_close,
    reward_on_close = reward_on_close)
4     self.action_space = gym.spaces.Discrete(n = len(Actions))
5     self.observation_space = gym.spaces.Box(low=-np.inf, high=np.inf,
    shape=self._state.shape, dtype=np.float32)
6     self.seed()
```

First we make sure that the prices are in a correct format. We then create the state (constructor description below). We construct the discrete action set where we have 3 actions. The observation_space is where our observations would be stored. We need to

store floats, and we set the low and high limits to infinity. The shape of the space is the same shape as the state. We then perform one step in our price (step within the state), check for the end of prices and handle the position change. We take into input the action to perform, and we return the reward.

```
def step(self, action):
    assert isinstance(action, Actions)
    reward = 0.0
    done = False
    close = self._cur_close()
```

We first check that the action to take is valid. We initialize the reward of this trade to 0.0 and we set the close price to the current closing price.

```
    if action == Actions.Buy and not self.have_position:
        self.have_position = True
        self.open_price = close
        reward -= self.commission_fee
```

If the action to take is buy and we do not have a share already, we now have a position whose open price, the price we buy it at, is the close price. We pay the commission fee.

```
    elif action == Actions.Sell and self.have_position:
        reward -= self.commission_fee
        reward += 100.0 * (close - self.open_price) / self.open_price
        self.have_position = False
        self.open_price = 0.0
```

If the action to take is sell and we have a share to close, we pay the commission fee, we get the reward, close the position and readjust the open price to 0.0.

To create a state:

```
class State:
```

```
  def __init__(self, bars, commission_fee, reset_on_close,reward_on_close
    = True, volumes = True):
    assert isinstance(bars, int)
    assert bars > 0
    assert isinstance(commission_fee, float)
    assert commission_fee >= 0.0
    assert isinstance(reset_on_close, bool)
    assert isinstance(reward_on_close, bool)
    self.bars = bars
    self.commission_fee = commission_fee
    self.reset_on_close = reset_on_close
    self.reward_on_close = reward_on_close
```

The arguments for the constructor are the same as for the environment. We make sure that they all are in a correct format. To reset the state:

```
def reset(self, prices, offset):
    assert isinstance(prices, data.Prices)
    self.have_position = False
    self.open_price = 0.0
    self._prices = prices
```

We set the have_position to False, the open price to 0.0 and we feed the new prices to the environment.

Again, the interaction between the agent and the environment happens as follows:

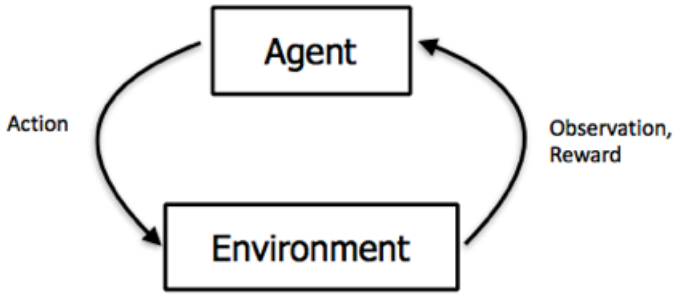

Figure 5.2: Step in the environment

To perform a step in the environment:

```
def step(self, action_idx):
    action = Actions(action_idx)
    reward, done = self._state.step(action)
    obs = self._state.encode()
    return obs, reward, done
```

We make a step by performing the action, we receive the reward and the done flag. We then encode the observation. The encode function converts the current state into a Numpy array. We then return the new observation, the reward and the done flag.

The model

The model has a dueling architecture:

```
class FFDQN(nn.Module):
    def __init__(self, obs_len, actions_n):
        super(FFDQN, self).__init__()
```

One part, three layers, predicts the value of the state $V(s)$:

```
        self.fc_val = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
```

```
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )
```

and the other , also three layers, the advantage of action $A(s, a)$:

```
        self.fc_adv = nn.Sequential(
            nn.Linear(obs_len, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, actions_n)
        )
```

Remember that the goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer (equation 2.14). As it was discussed in 2.1.3, we want the mean for advantage of any state to be zero so we substract it from the Q expression.

```
    def forward(self, x):
        val = self.fc_val(x)
        adv = self.fc_adv(x)
        return val + adv - adv.mean(dim=1, keepdim=True)
```