The Effectiveness Of Reinforcement Learning For Cryptocurrency Trading Actions

Adrian Ronayne
School of Computer Science
National University of Ireland, Galway

Supervisors
Dr. Frank Glavin

In partial fulfillment of the requirements for the degree of
MSc in Computer Science (Data Analytics)
September, 2020

Abstract

Cryptocurrencies have been an increasing focus of attention in recent years. Demand from retail and institutional investors has resulted in the creation of many cryptocurrency exchanges. Paying for goods using cryptocurrencies rather than Fiat [a government issued currency] is also experiencing increased demand. On the investment side, cryptocurrencies can be highly volatile, but over time their value has been unwaveringly increasing, which offers investment opportunity. As a traditional form of currency, cryptocurrency transaction costs are typically lower than Fiat-based currencies.

There are caveats to the use of cryptocurrencies. Among these are that it's not widely supported. Cryptocurrencies are built upon blockchain technology, therefore are built on the idea of decentralised trust. Conceptually this is a very different way of thinking whereas generally, trust is centralised to a single entity. Transaction completion times of cryptocurrencies can take time in comparison to Fiat, case in point is Bitcoin with a transaction completion time of up to 10 minutes. The quintessential bitcoin paper (Nakamotoet al. 2019.) is a detailed and concise outline of how a currency can exist without the need for a centralised trust location. While not all of today's cryptocurrencies will survive, it is highly unlikely none will and Bitcoin is on its way to becoming a store of value comparable to gold.

With this research, the aim is to discover a methodology for applying reinforcement learning techniques to time series financial data. Reinforcement learning models are explored and briefly explored is an architecture for deploying the models as part of a software deliverable component.

The results of this research are promising but inconclusive. Playing a game of tic tac toe, the model successfully learns via self-play to improve against a player performing random moves. For trading, results on a subset of the experiments suggest the training and test sets converge on trading actions that produce a positive reward.

Declaration

I, Adrian Ronayne, do hereby declare that this thesis entitled The Effectiveness Of Reinforcement Learning For Cryptocurrency Trading Actions is a bonafide record of research work done by me for the award of MSc in Computer Science (Data Analytics) from National University of Ireland, Galway. It has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification.

_____

Adrian Ronayne

September 2020

Table of Contents

List of Figures

List of Tables

1. Chapter 1

1.1. Introduction

CoinMarketCap (CoinMarketCap. 2020.) lists prices for over 5500 cryptocurrencies. The combined market capitalisation of these cryptocurrencies is over $273B. As an investment vehicle, cryptocurrencies are now difficult to ignore. Towards the end of 2015 Bitcoin traded between "$395 – $504" (History of Bitcoin. 2020.), today its worth over $8'000. The level of growth is unprecedented, and more recently has attracted institutional investment. The drivers of many cryptocurrency price increases are diverse. Quantitative easing has increased in recent years, leading to increased inflation of Fiat currencies. Some investors now view cryptocurrencies, in particular, Bitcoin as a hedge against inflation.

Machine learning trading strategies are not new. The trading firm Renaissance technologies (Renaissance. 2020.) are one of, if not the most successful trading firms in existence today and utilised computer models as early as the 1990s to determine market trading actions.

Historically there have been widely publicised cases which suggest AI is on the cusp of overtaking human intelligence, also known as the "singularity". A recent example of this is the publicity surrounding a machine expertly playing Go, named AlphaGo (Mastering the game of Go without human knowledge. 2020.) . It is highly unlikely AlphaGo could have successfully executed within an acceptable time frame on available hardware ten years ago. The "Bitter Lesson" blog post (The Bitter Lesson. 2020), written by one of reinforcement learning's most prominent proponents, describes the advances in AI mainly being attributed to increases in hardware performance. Sutton suggests in the "Bitter Lesson" that as ML practitioners, we should allow the machine to determine where possible the process parameters and not codify the process with our ideas and intuitions. This raises an important question, what if any is the distinction between search and intelligence. Understanding and exploring the difference between search and intelligence is a profound philosophical question, and while outside the scope of this thesis is useful as a guiding principle in developing RL algorithms.

Associated with trading is the creation and loss of capital for retail and hedge funds. Proprietary trading firms such as "SMB Capital" (SMB Capital. 2020) are firms that enter the market, make trades and leave the market with an expectation of a positive return. Many successful traders cite the trading chart as the primary source of information for making trades. The reinforcement learning model that has been developed for this research will be referred to as "market agent". Cryptocurrency price data is utilised by the market agent model, which leads to the research question:

- Can a facet of trader behaviour be emulated using the underlying data which generates pricing charts?

The objectives are :

- Analyse research cryptocurrency data.
- Identify currencies that contain the most signal.
- Train an RL algorithm on cryptocurrency data to predict trading actions that result in profitable trades.

The motivations for this work are to gain an understanding of a facet of RL, and its application to the trading domain. Currency pricing data is easily accessible from various cryptocurrency exchanges which allows the initial step in developing machine learning driven trading strategies. The availability of data allows to quickly begin research on the RL model with little preprocessing of data and building of data infrastructure.

Chapter 2 begins with a brief overview of RL and crucial definitions, a literature review is provided in chapter 3. Chapter 4 and chapter 5 focus on the RL models explored and describes the pre-processing data methods. It is useful to test RL models on a game that is clearly understood and intuitive as it forms a base to gain intuition in developing the model. For this reason, this research will include solving the game of tic tac toe using RL. A more advanced version of RL using a deep Q network applied to a trading environment is then explored. Chapter 6 provides source code to reproduce the results for each implemented

model with the results presented in chapter 7. Chapter 8 concludes the thesis with some final thoughts on improving the market agent model.

A web application is developed, which allows the user to play a game of tic tac toe against the RL agent as well as graphically view model predicted actions.

This thesis makes the following assumptions :

- Signal exists within pricing data.
- An RL model can utilise signal to initiate profitable trades.

2. Chapter 2

2.1. Reinforcement Learning Overview

Here an overview of RL is provided. The RL techniques explored in this thesis are :

1. Tabular learning
2. Deep Q-Network

2.2. Definitions

Provided is a list of some commonly referred to RL and finance definitions.

Action: The methods or functions the RL agent performs within an environment.

Agent: Acts on an observation received from the environment.

Activation Functions: Compute the output for a given neuron in a neural network. Typically the output is based on several inputs to a given neuron. There are many types of activation functions, Sigmoid and ReLU being among the most common in use.

Sigmoid: Also known as the logistic function due do its use in logistic regression. The output of sigmoid is generally used as a probability where the output is a label of the function parameters, referred to as theta.

$$h_\theta(x) = \frac{1}{1 + e^{-(\theta^T x)}}$$

ReLU: ReLU returns the function parameter if it's positive or returns zero if the parameter is negative. Typically ReLU has overtaken Sigmoid as the choice of the loss function. The reasons are:

- Lower risk of the vanishing gradient problem. For activation functions such as sigmoid when taking the derivate of a substantial input value, the gradient tends to zero. Glorot et al, 2011 refer to this as "there is no gradient vanishing effect due to activation non-linearities of sigmoid or tanh units".
- Less computationally expensive.

Bias: High bias occurs when a model is consistently producing similar predictions regardless of the training data. In the context of RL, a high bias policy function approximator always predicts the same actions independently of the state. The policy function approximator should have low bias.

Bootstrapping: Update Q values based in part on other Q values without waiting for an outcome. TD(0) is a bootstrapping method. Bootstrapping is generally referred to in the context of dynamic programming methods where "All of them update estimates of the values of states based on estimates of the values of successor states. That is, they update estimates on the basis of other estimates. We call this general idea bootstrapping." (Sutton and Barto, 2018, p87) .

Cost Function: Also known as a loss function, the cost function provides a measure of the accuracy of model predictions.

Examples of cost functions are cross-entropy and mean square error (MSE). Typically for classification problems, the cross-entropy cost function is used, for regression problems MSE is used.

An example cost function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

In RL, the policy network uses the MSE cost function.

Epsilon Greedy: A greedy algorithm is one that always takes the action which is interpreted as providing the best reward. The value ϵ which is contained in the interval [0,1] manages the trade-off between exploration and exploitation.

Setting ϵ to .1 the agent selects a random action with probability 0.1 . The selection of a random action is also referred to as exploration in the context of RL. With probability 0.9 the agent selects the greedy action. The selection of an action deemed to return the highest reward is also referred to as exploiting the given state.

Epsilon Greedy Decay: To reduce model exploration over time, reduce epsilon by some value for each episode. As epsilon is in the interval [0,1] the decay value is typically set to less than 0.

Experience: This term is often used to describe some aspect of an RL model, for example, experience transition, experience replay and past experiences. Experience can be generally defined as how an agent processes the data it has access too. Goodfellow et al (2016) refer to experiences in the following contexts of unsupervised and supervised learning:

- Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset.
- Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target.

Fiat: Currency issued by a government.

Genetic Programming: A method of optimising model parameters. In the context of policy approximation, genetic programming discovers the optimal hidden neural network layer weight configurations.

Index Fund: An index fund is a type of mutual fund with a portfolio constructed to match or track the components of a financial market index, such as the Standard & Poor's 500 Index (S&P 500). (Chen, 2020)

Log Function: Log functions are widespread in applying ML algorithms. Applying the log function to values in computation reduces the risk of computational inaccuracies due to loss of precision, also known as underflow. As the log function is a monotonically increasing and decreasing it can safely be applied to computing the gradient during gradient descent in the minimising of a cost function.

Markov Property: If the Markov property is satisfied, the given state is dependent only on its previous state.

Model Free: The agent utilises just action and reward to infer the best action. The best action is determined via adjusting the policy parameters in order to gain more reward. The agent learns from experience only.

On Policy and Off Policy: Frequently RL algorithms are referred to as off policy or on policy. ϵ greedy can be used in both on policy and off-policy learning.

Q Learning is an example off policy while SARSA is an example of on policy.

Off Policy: The agent learns the value of the optimal policy independent of the agent's actions. "off-policy methods evaluate or improve a policy different from that used to generate the data." src: Barto The generated data is the observations about states, actions and rewards.

On Policy: The policy the agent uses to determine actions is also the policy being learned. "On-policy methods attempt to evaluate or improve the policy that is used to make decisions" src: Barto

Pump and Dump: Used to increase the price via many large buy orders to attract other buyers, and ultimately the aim is to sell the asset once an increase in price exceeds a threshold.

Policy Gradient: An optimal policy for a specific MDP corresponds to the strategy that, if followed, is guaranteed to provide the highest amount of reward in the environment it operates in.

Policy gradient methods explicitly parameterise the policy. The parameters of the policy are differentiable and represent the weights of a neural network. For each step, the neural network predicts the probability of each action.

Portfolio Weight: The percentage of an investment portfolio that a particular holding or type of holding comprises. The most basic way to determine the weight of an asset is by dividing the dollar value of a security by the total dollar value of the portfolio. (Investopedia. 2020. Portfolio Weight.)

Q Value: For a given state action pair Q(S,A) the Q value provides an estimation of its value. Using the temporal difference update rule Sutton, Richard S., and Andrew G. Barto (2018) the value of Q is continually updated.

Reward: A scalar value returned to the agent from the environment after taking action.

State: A representation of an observation of an environment.

Temporal Difference Learning: The temporal difference is the measurement of changes in the value of two states at time t and t+1. Temporal difference learning utilises the temporal difference to discover a better value estimate of a given state.

Trading Volume: The quantity of a cryptocurrency that has been traded during a time interval.

Variance: High variance occurs when a model consistently produces predictions which have a high variability with minor changes in the data supplied to the model. In the context of RL, a high variance policy function approximator predicts different actions with small changes in the state attributes. The policy function approximator should have a low variance.

2.3. Introduction

RL is not a fully supervised or fully unsupervised method of machine learning. For RL, ground truth is not defined as in supervised learning. RL utilises a reward signal in determining the utility or value of a given state. The reward signal is a scalar value and designates how good the RL model is performing for a given time step. The methods of RL provide a framework to train ML models that are concerned with decision making. Therefore the applications of RL are wide and varied. The applications include game playing, choosing when to invest in a stock and the control of physical robots.

2.4. Markov Decision Process (MDP)

A Markov decision process or MDP defines the environment in which the agent operates and includes the transition and reward probabilities. Continually the agent is interacting with some MDP via taking actions and receiving a positive, negative or neutral reward. A Markov decision process assumes the Markov property, which states all future states are dependant only on the current state. However, the current state can encapsulate a history which spans more than a seemingly single state. For example, Mnih, Volfourdymyr, et al. state as part of the DQN model "the last 4 frames" of images are used to form "a history" which are then stacked "to produce the input to the Q-function."

A state transition probability matrix defines the transition probabilities from all states s to all successor states s'. Sampling from an MDP can be used to determine the state transition matrix values. Each row characterises the transitions. For a Markov chain diagram, a square represents the terminal state.

A Markov reward process is defined as a tuple (S,P,R,γ):

- S: the state space.
- P: the state transition probability matrix.
- R: A reward function.
- γ: the discount factor.

γ controls if the reward is preferred in the immediate or long term. If γ is set close to zero, then the future reward is less important than an instant reward. Using a low γ value, the RL agent will try to focus on actions which will return immediate rather than future rewards.

The following diagram an example of an MDP process where the terminal state is "Sleep". Once in the terminal state, the agent cannot move to any other state. From each node, a probability determines the likelihood of moving to another connected state. Generally, the model will be unaware of the probabilities of moving between states and are learned via sampling from the MDP process.

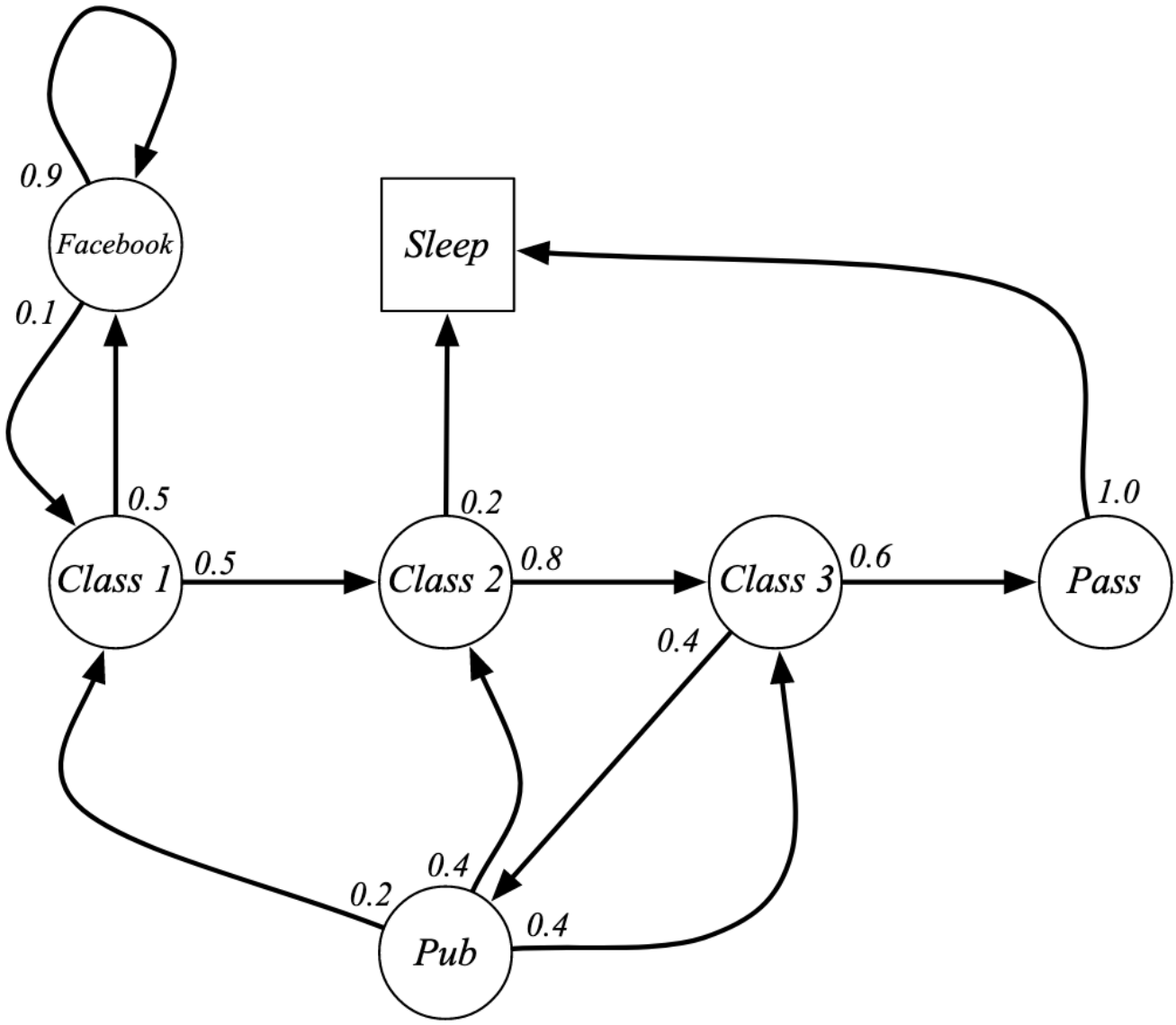

Figure 1. Example MDP Process (src: David Silver, Student Markov Chain, 2020)

2.5. Planning by DP.

We are interested in determining for a given MDP how much reward is achieved, known as policy evaluation. Dynamic programming is a form of mathematical programming and is used to optimise a function. It differs from divide and conquer techniques, for example, merge sort. Dynamic programming caches partial solutions to a problem, each partial solution can be reused to arrive at the final optimisation, for example, a policy.

The Bellman equation tells us how to break down the optimal value function into two steps.

Step 1: the optimal behaviour for the current step.

Step 2: the optimal behaviour for the next step.

The value function behaves like a cache for computed values. This function returns the reward quantity for any state in the MDP. For a given policy its value function is improved by acting greedily with respect to the value function for that policy.

For an MDP, there exists at least one deterministic optimal policy. An example of policy evaluation is Jacks Car Rental problem which is described as "*Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location* to rent *cars. If Jack has* a *car available, he rents it out and is credited* $10 by *the*

national *company. If he is out* of *cars at that location*, then the *business is lost. Cars become available for rent the day after they are returned. To help ensure hat cars are available where they are* needed, Jack *can move them between the two locations overnight*, at *a cost of $2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables* " (Sutton and Barto, 2018, p81).

Policy iteration converges to the following policy:

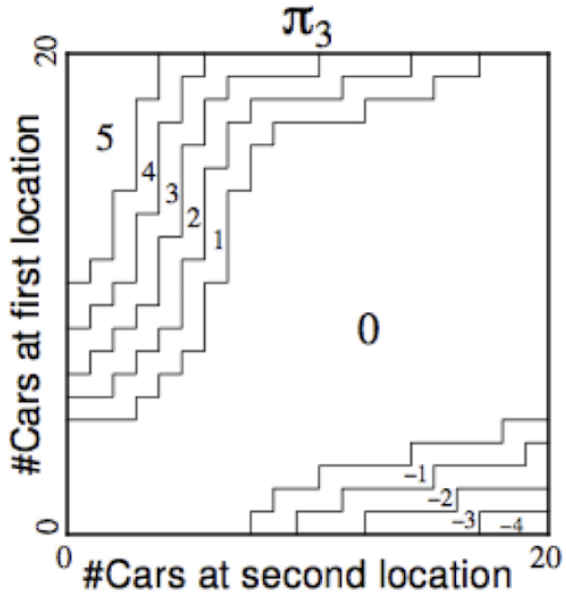

Figure 2. Optimal Policy src: Barto

To interpret the optimal policy diagram defined above if there are 0 cars at the second location and approximately between 10 and 20 vehicles at the first location then we should move 5 cars from the first location to the second location to maximise rental returns. This problem focuses on planning. Therefore the dynamics of the system are known.

A greedy policy measures the value of an action in a particular state and subsequently follows that policy. The goal is to select actions which return the max action value. Put another way q is the action value, and the model searches for the policy that yields the most q. q is the immediate reward upon taking action summed with the subsequent value of the next state as a result of that action.

Once the Bellman equation has been satisfied, then the optimal policy for an MDP has been found. A simple method of stopping before convergence when policy improvement is "close enough" is to stop iterating when the Bellman updates ϵ that are changing the value function becomes negligible. This is known as ϵ convergence of the value function. Policy improvement can be considered as a partial ordering over policies. If the value function has higher values than the other policy in all states, this provides a partial ordering over policies.

In policy iteration for every step, we construct a value function for a particular policy. In contrast, value iteration does not build the policy as an intermediary step.

2.6. Model Free Control

When solving real-world RL problems, typically we don't know the dynamics of the MDP. Therefore we use model free prediction and model-free control. On policy learning refers to learning about a policy by following that policy, the model is sampling actions from some policy π and in parallel evaluating the action taken. Off policy learning estimates expected values from one distribution "given samples from another", the sampling technique is known as "importance sampling". (Sutton and Barto, 2018, p104)

For model-free control, a Q function is used instead of the V function to improve the policy. Q is model-free, i.e. it does not require to know the dynamics of the MDP. The Q function requires the current state and action to determine the state value, while the V function requires the value of the next state and is not model free.

Monte Carlo policy evaluation takes the mean of trajectory values returned by the value function.

It can be proved that any ϵ greedy policy improves as a result of taking an ϵ greedy step. After taking each step using ϵ greedy, a new policy is created dictated by its parameters which determine the action to take. Policy improvement does not indicate the expected time to explore all states. It is not always necessary to fully explore a policy before evaluating it. We want to update our policy at some point before the episode ends to update it to a better policy. The policy should converge to a greedy policy to ensure the action with the max Q value is selected for a given state. Decaying ϵ in ϵ greedy is a method to achieve greedy policy convergence.

Temporal difference control is favoured over monte carlo control for several reasons. Temporal difference is online. Therefore the agent does not need to wait until the end of an episode to update the value function. At the end of each step, the value function is updated. Temporal difference typically has lower variance in achieving convergence. Temporal difference learning utilises the Q(state,action) function to perform model-free policy iteration.

2.7. Policy Gradient

Actor critic methods use both value functions and policies. Policy gradient methods optimise the policy directly and do not use value functions. The gradient in "Policy Gradient" translates to how the model adjusts the policy in the direction of the gradient.

Policy gradient methods use gradient ascent. Gradient ascent is used to maximise an objective function such as the amount of reward to access. If trying to maximise the likelihood of some action, then follow the gradient of the defined score function.

The softmax policy is an alternative to ϵ greedy policy and works well for a discrete set of actions.

3. Chapter 3

3.1. Literature Review

In this literature review, we provide a detailed overview of RL and relevant research to the domain of this thesis, namely the financial markets. Deepmind [1] is referred to frequently as part of this literature review as has contributed to many highly cited RL papers in recent years, including Wang et al (2015), Zhang et al (2019), Mnih et al (2013) and Schaul et al (2016). Many of the papers in the RL field utilise a gaming environment in which to test their hypotheses. Gaming environments can enable many plays to discover what can be learned and exploited to win a given game. Much of the publicised success of RL applies to game environments where key RL concepts such as state, actions, and reward have been successfully implemented. This has led to the advancement of RL.

It is not yet verified if RL can be applied to the subject of this thesis, trading the financial markets. The mechanism of discretisation of state, actions, and reward from a continuous space is not precise. The RL for trading literature seems to overlook modeling how the agent influences the environment as a result of its trading actions, utilising a varying action size instead of immutable leads to more complex action space. Publicly available research with repeatable results that returns a profit in RL applied to the trading environment is not widely available or is difficult to recreate. Upon discovery of a trading edge, it is typically not shared publicly with the aim being to extend the edge lifetime.

Implementing an RL agent in a trading environment needs to address several potential issues. The transaction cost of placing a trade is not deterministic, and it is subject to the environment. Transaction costs can severely discount any profit-taking and cause the selected action to be loss-making as a result. The state-space of pricing data can grow extremely large. Representing the state space in a lower-dimensional structure using an autoencoder is considered.

1. https://deepmind.com/

3.2. Representing the Problem

Mnih et al (2013) advance combining RL with deep learning to extract features instead of utilising domain knowledge to handcraft features. The Bellman equation is likely used as the cost function, although not explicitly stated. For a deep learning model, the cost function provides a measure of how accurate the model and its current weight configuration are at mapping training input values to its corresponding outputs. A deep learning network is utilised to approximate Q values. Q values are a numerical representation of the values for a set of state → action pairs. Representation of Q values is via a table lookup data structure. The model described is off policy and model-free.

Dai et al (2019) provide an overview with terms and definitions of the domain this research encompasses. The domain explored is FX trading, which involves trading currency pairs to maximise profit throughout multiple trades. Brokers act as intermediaries between the currency sellers and buyers. Brokers offer a bid and ask price for a currency pair. The bid price is the price the broker will pay for an amount of currency. The asking price is the price of purchasing an amount of a given currency. The Dai et al (2019) dataset consists of the bid and ask prices for 8 currency pairs. The bid-ask spread is a measure of the difference between the bid price and ask price; using this information, the model described attempts to make optimal trades to maximise profit-taking. The reward attained by the model is set to a negative value and therefore penalised if the currency bought represents the ask price * 'units purchased'. The reward attained by the model is set to a positive value and therefore incentivised if the amount sold is the bid price * 'units sold'. In conclusion, the agent is rewarded for selling and penalised for buying. Trading the currency pairs AUD-USD, EUR-USD, GBP-USD is undertaken over three weeks. A summary of the results is as follows:

Table 1. Dai et al (2019) mean return

Currency Pair	Week 1	Week 2	Week 3
AUD-USD	-0.25	-0.18	-0.21
EUR-USD	0	0.3	0.072
GBP-USD	-0.12	-0.69	-0.413

The dataset is high frequency as it contains trades at second interval time steps. To further increase the likelihood of profit-taking, the bid-ask spread is calculated from the pricing data, and the minimum bid-ask spread is selected as the model dataset. This model requires 'human intervention' during periods of high volatility as implied by "Human intervention is required when the market experiences some strong deviations from expected.". A further implication is that during periods of high volatility, model performance decreases. Incorporating features into the model that are also utilised by "professional traders" is suggested. Feature selection is an area to improve upon, features described are "technical indicators," of which examples are:

- Moving averages
- Relative strength index
- Bid-ask spread

The EUR-USD currency pair performs slightly better than other currency pairs, and it is not clear why this is the case. Reasons for successful trading are attributed to selecting data that contains a minimum bid-ask spread, a short trading duration of one hour, and including transaction costs into the decision making process.

Jiang, Zhengyao et al (2017) argue that if an ML model produces price predictions but does not select actions based on those predictions, then the "approach is not fully machine learning". An issue with applying RL to trading is that although market actions can be discretised, they are nevertheless continuous. discretisation is described as being a "major drawback" as discrete actions are coupled with "unknown risk". This paper suggests utilising information from previous episodes as "portfolio weights from the previous trading period are also input". The portfolio weights determine the number of assets bought. Assets with increased target weights are bought, while assets with decreased target weights are sold. Assets with increased target episodes are not clearly defined in this paper, suggesting this problem is applied as a continuous RL problem. A distinction is made between cumulative and immediate reward. The immediate reward is defined as rt/tf where rt is reward at time t and tf is the "length of the whole portfolio management process". Therefore as the length of the portfolio management process increases, the penalisation of immediate rewards also increases. Higher trading volume assets are selected as higher volume implies higher liquidity. High volume also implies actions have less impact on the environment, but this is less of a concern for this thesis as the size of trades is expected to be small as not to have a significant environmental impact. Similar to Jiang, Zhengyao, and Jinjun Liang (2017), the selected cryptocurrencies in Jiang, Zhengyao et al (2017) are based on the volume traded with Bitcoin being included in the selection. This model does consider the effect of transaction costs when making investment decisions.

Hasselt et al (2015) provides the following useful definitions which are frequently referred to in RL literature:

- Model-free: Solves the RL task directly using samples from the emulator E, without explicitly constructing an estimate of E
- Off-policy: Learns about the greedy strategy a = max_a Q(s, a; θ), while following a behavior distribution that ensures adequate exploration of the state space. In practice, the behavior distribution is often selected by an ϵ greedy strategy that follows the greedy strategy with probability 1 − ϵ and selects a random action with probability e.

3.2.1. Previous Work

This section describes previous work and results for similar problems in RL for the trading domain.

Zhang et al (2019) suggest financial data contains a "low signal to noise ratio". An implication is to extract a signal from financial data; a large dataset size is required. Any signal's found effectiveness varies throughout time. If the model discovers a predictive signal, then the signal value may decrease over time. The signal decrease depends on how the signal is exploited. The paper does not focus on making predictions, such as how a stock price direction will change. Instead, the model focuses on outputting trade positions. The problem is treated as a continuous learning task. Therefore episodes are not defined. Representation of state includes pricing and returns data. The space of actions are "short position", the agent holds shares for a short period with the expectation they will decrease in price. "Long Position" shares are held for a more extended period with the expectation they will increase over the period. "No holding" translates to no shares are held. The breakdown of the allocation of shares within the portfolio is not clear.

The metrics for the results of Zhang et al (2019) are also applicable for this thesis. An obvious metric is measuring the trade returns, which is the amount that is gained or lost as a result of a trade. Also, the σ is a valuable metric as a high return with a low σ implies high return with low risk, which is ideal. Basis points (bp) represent the trading transaction costs. For example, if the transaction cost is 1bp and 1bp = .0001 then to purchase €1000 (be in a futures contract or currency), the transaction cost is €.10 (€1000 * €.0001).

Jiang, Zhengyao, and Jinjun Liang (2017) describe training a network on just over six months of trading data and backtesting the network on a trading period of 30 minutes. Twelve cryptocurrencies are traded using this model, and a 10x fold return on investment is claimed. A "riskless asset" is utilised to represent data, but it is not clear how a riskless asset can be defined. All assets are associated have some risk associated. Pricing history data "is used to represent the state of the market". The model employed by Jiang, Zhengyao, and Jinjun Liang (2017) assumes the generated investment decisions "will not affect the price of the asset".

Xiong, Zhuoran, et al (2018) utilise 30 trading stocks with prices and compare the model performance to the Dow Jones Industrial average. The model is similar to a common approach in measuring investment fund performance. For a given fund, if performance is negligible compared to an index fund, then an investor will typically move to the index fund as fees are much lower. Return on investment is utilised using Deep Deterministic Policy Gradient (DDPG) algorithm, which is an "improved version of Deterministic Policy Gradient (DPG) algorithm ". DDPG is similar to DQN in that state transitions are stored in an experience replay buffer. An experience replay buffer is used to store transitions. DDPG reduces the correlation between experience samples. The model performance is measured using:

- Final portfolio value
- Annualised return
- Annualised standard error
- Sharpe ratio

The authors report a high Sharpe ratio which indicates this "strategy beats both Dow Jones Industrial Average and min-variance portfolio allocation in balancing risk and return". From this Xiong, Zhuoran, et al (2018) conclude that a trading strategy can be developed which outperforms the Dow Jones Industrial Average.

3.2.2. Experience Replay

Experience replay refers to the storing of the RL agent experiences, defined as $e_t = (s_t, a_t, r + t, s_t + 1)$, Mnih, Volodymyr, et al (2015). Q learning samples from the stored agent experiences during the update process.

Mnih et al (2013) define a method for performing experience replay, similarly to (Mnih, Volodymyr, et al) this is described as "to perform experience replay we store the agent's experiences $e_t = (s_t, a_t, r + t, s_t + 1)$ at each time-step t in a data set $D_t = e_1, e_2, \ldots, e_t$ ". Learning is then a separate process from gaining experience. During the Q learning phase updates are made by sampling from the experience replay memory. This paper describes a mechanism for replacing handcrafted feature selection with deep learning feature discovery. This paper led to more research in this area with other deep learning architectures being explored, such as CNN and LSTM.

Schaul et al (2016) describe instead of experience transitions being sampled from replay memory, Mnih et al (2015), experience replay allows agents to "remember and reuse" past experiences. This paper describes a framework for prioritising important transitions for experience replay. Transitions with high expected learning progress are prioritised. High expected learning progress is a measure of the temporal difference learning error. This error is measured as the difference between the actual and estimated reward. The error can be a poor estimate if the rewards are noisy, this raises the question, can we measure noisy rewards? Greedily prioritising, where the temporal difference learning is high, has consequences. Among the consequences are transitions with a low TD error as once these transitions are visited, they may not be replayed for a "long time". The selection of experiences is coupled with greedy prioritisation and random sampling to overcome the low transition visitation rate.

3.2.3. Environment exploitation

Ziyu et al (2017) mention the need to reduce the cost of an "expensive simulation step" during the exploitation of the environment. A similar advantage function is described in Wang et al (2015) and is utilised to provide a relative measure of the value of each action. A new "trust region policy optimisation" is introduced, which maintains a running average of the past policies and, on average, forces the updated policy to not "deviate from this average".

3.2.4. Learning Q Values

Wang et al (2016) describe using two separate architectures, one for learning state values, the other for learning action values advantages. For each architecture, the same Bellman equation is utilised. The Q value measures the value of choosing an action for a given state. The advantage function is the difference between the state value and Q function. In other words, the Q function values are subtracted from the state value. The result is the "relative measure of the performance of each action". The relative measure of the performance, also referred to as its importance becomes more advantageous as the difference increases. The Q value produced by this model combines the values and advantage function to update the state-action value.

3.2.5. Representation of State

Mnih et al (2013) define the 'Q network' as the neural network function approximator. Utilising a neural network model to approximate the value of a given state is a typical pattern in deep RL. The learning agent is not aware of the internal state of its environment. The environment is defined as an Atari emulator. The agent relies on the "raw pixel values", which maps to the current representation of the game screen. The exclusion of the internal state in this model is relevant as intuitively, the external state is noisier than the internal state. Of course, this depends on how an internal state is constructed. For example, instead of relying on emulator images for the state, utilise player moves, for example, up, down, backward, and forward.

An autoencoder is utilised to represent a lower-dimensional time series state-space. We want to maximise the likelihood of our training data being generated, which in turn is we are maximising the likelihood of pricing data being generated. We want the encodings to represent the most important and meaningful features in the data. Two networks are used to train a model to reconstruct original data, an encoder, and a decoder. The encoder maps to a lower-dimensional representation. The decoder upscales the dimensionality to the same dimensions as initial training data. The loss function determines how inaccurate the decoder representation is from the training data. If we utilise an Autoencoder, the idea is that some states will be revisited as a result of their encoding. Similar states will be based on a distance measure, for example, euclidian distance. Similar states will be stored in buckets based on the encoding of raw pricing data. Hsu, Daniel (2017) describe using an autoencoder to represent time series data. A recurrent neural network of type long short term memory model is used as the autoencoder architecture. The data is pre-processed into partitions where each partition has a different length but similar variation in its values. The difference is a pre-defined value and is manually set. The data compression ratio's achieved vary between 0.03 and 0.08. The compression ratio is defined as $Compression\,Ratio = \frac{Un\,Compressed\,Size}{Compressed\,Size}$ (Data compression ratio. 2020). Therefore the compression achieved by this method varies

between 3% and 8% of the original data size.

State can also be represented as a function:

$$state = (getSeeOpponent()\text{Unknown character}\text{Unknown character}25) + (getCurrentAmmo()\text{Unknown character}\text{Unknown character}5) + getCurrentHealth()$$

Glavin, Frank G. Towards Inherently Adaptive First Person Shooter Agents using Reinforcement Learning. Diss. Doctoral Dissertation, 2015.

3.2.6. Model Optimisation

Suganuma, Masanori, Shinichi Shirakawa, and Tomoharu Nagao (2017) employ a genetic programming approach for designing the architecture of a CNN model. In this model, the validation accuracy of the trained model is used as the fitness value. Models with high validation accuracy are progressed to later generations. The authors report that this method is comparative with state of the art results where hyperparameters are manually tuned. The computational resources required for this method are significant, evidenced by the optimal configuration requiring "800 GPUs for the architecture search".

Conclusion

Further research is required to determine the applicability of RL to the trading environment. There is a wide array of implementation choices to be made when implementing an RL model. Coupled with this is a constant stream of research offering new and better methodologies. In quick succession, these newfound methodologies are also improved in a recursive process. An example of this is utilising experience replay, Mnih et al (2013), to improve Q learning, Schaul et al (2016) built on this idea for prioritising the transitions to be selected.

The discretisation of actions is described as being a "major drawback" by Jiang, Zhengyao et al (2017). discretisation of reward may also prove to be a drawback as it is challenging to assign rewards that describe the outcome of agent actions. If the agent selects an action which results in a €300 return on investment and a subsequent action which results in a €600 return on investment how should the agent be rewarded in each case ? intuition may guide to assign reward of 1 in the first case and 2 in the second to emulate the monetary amounts but the time horizon of reward attained may not be considered. It is challenging to target which trade execution in a sequence of trades should be assigned the highest reward as an initially negative losing trade over time can become a profitable one. discretisation has the drawback of potentially not assigning adequate reward values.

Jiang, Zhengyao, Dixing Xu, and Jinjun Liang (2017) describe risk with the discretisation of continuous action spaces, but this should be controllable if we define our actions carefully. Extreme action is defined as "investing all the capital into one asset, without spreading the risk to the rest of the market.", but this action should not be defined in the creation of a trading algorithm.

Measures of the accuracy of models appear to be at a macro rather than micro level. For example, the following criteria are not utilised as a measurement of model performance by authors of papers in this literature review :

- Sensitivity
- Specificity
- Confidence Interval
- Precision
- Recall

3.2.7. Issues to consider

Several issues will need careful consideration while creating the model and utilising domain data.

Q Learning

Ziyu et al (2017) highlight an issue with Q learning that is also relevant for this thesis, namely action selection for "large action spaces". The data for this research contains millions of trades from 6 cryptocurrencies. To be decided is if the RL agent will utilise exploration for all six cryptocurrencies or some subset. Regardless the action space will be large. The RL agent will need to provide a trading decision in minimum time to take advantage of any edge (though perhaps not in all cases); therefore, any performance improvement is worth exploring.

Limited Memory

Hausknecht and Stone (2015) describe addressing the "limited memory issue" issue in Markov Decision Processes. More specifically, this refers to addressing MDP's not being able to map from more than a limited number of past states. The Q network is trained on full observations and inferences are then made on partial observations. It is not clear how the inclusion of a Long Short Term Memory model affects the accuracy of inferring partially observable MDPS's. In the context of this thesis, the topics discussed could prove useful. Each state is defined as a sequence of bid/ask prices, and the ideal number of attributes to be included in each state is unknown. As the state size increases, more processing is required to determine the highest Q value. At some point, the state size may cause the decision-making process to lag, such that any trading decision is redundant. Therefore to increase decision making performance, reducing the state size and inferring partial observability will improve decision-making.

Q Network

Dai et al (2019) and Xiong, Zhuoran, et al (2018) conclude that a Deep Q-network does not work well for trading. Issues with Deep Q Networks (DQN) are highlighted and include training instability and variability. The DQN agent learns to take neutral positions only and break even on the test set. The discretisation of the action set is difficult in allowing the agent to learn. Other difficulties are 'the indirect learning objective, and confusing features', Dai et al (2019). Confusing features may be eliminated by using neural network architectures for feature generation such as LSTM or CNN, depending on the availability of data.

Representing Episodes

Discrete, rather than continuous episodes, is described as being "mathematically easier because each action affects only the finite number of rewards subsequently received during the episode." (Sutton, Richard S., and Andrew G. Barto) . Cryptocurrency exchanges are open 24 hours per day. Such exchanges are not subject to typical trading periods, unlike the New York Stock Exchange (NYSE), which trades between 9:30 a.m. to 4 p.m. Therefore, incorporating continuous, rather than discrete episodes as described by Zhang et al (2019) may require further examination.

Yuqin et al (2019) describe designating an episode to be "one hour long" and the agent learns to make "decisions to maximise the reward" for that episode. For this research, decisions are taken at each time step to maximise reward instead. The guiding intuition here is if the reward is computed at each time step then the RL algorithm will achieve better convergence properties as a result of learning at smaller time intervals rather than an episode length.

4. Chapter 4

This chapter describes the methodology for the collection and preparation of the pricing data. The process for calculating reward is also described. Price data from the Coinbase exchange is utilised by the RL agent.

4.1. Data Overview

The figure 'Sample Coinbase cryptocurrency trading screen' below is a Candlestick chart and represents the price movement of a given cryptocurrency. Each candlestick represents the movement within a five minute interval. A new bar is added to the chart when the interval ends. The interval is not static and can be increased or decreased to suit a more macro or micro view of the trading price for a given cryptocurrency.

The figure 'Five minute interval price chart' is for illustrative purposes to assist with providing an intuition of trading charts. The colour scheme of each bar is determined based on the price history.

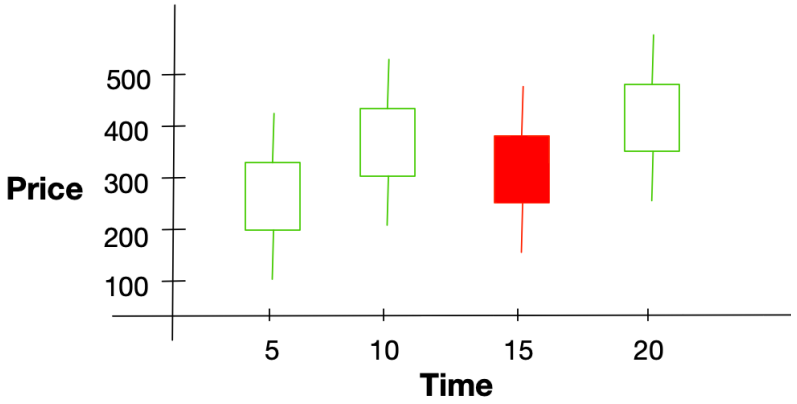

Figure 3. Five minute interval price chart

If the closing price is less than the opening price for subsequent bars then this is reflected with a red or orange solid filled colour:

Figure 4. Five minute interval: Close < Open

If the closing price is greater than the opening price for subsequent bars then this is reflected with this is reflected with a green unfilled colour:

Figure 5. Five minute interval: Close > Open

For example, the figure 'Five minute interval price chart' represents a simplified synthetic chart of four bars where the third bar closing price is less than the opening price:

4.2. Acquiring data

Pricing data is acquired from the Coinbase Pro cryptocurrency exchange (Coinbase Pro). The pricing data is the only source of signal for the RL model. Each data instance is the price for a given unit of currency at time t.

From the Coinbase Pro cryptocurrency exchange 45 days of price data for the following currency pairs are downloaded: 'BTC-EUR' , 'XLM-EUR' , 'ETH-EUR' , 'XRP-EUR' , 'LINK-ETH' , 'LTC-EUR' , 'ZRX-EUR' , 'BCH-EUR' , 'EOS-EUR'. Coinbase exposes an API which allows downloading the most recent 200 trades for a given cryptocurrency. Custom infrastructure is required to store data for more than the most recent 200 trades. Empirically it appears the length of the time interval for the most recent 200 trades decreases as volatility of price increases. A price window defines n prices within a time interval where n_1 is the mean price for $minute_1$ of a given currency.

4.3. Data selection

From the set of downloaded data, the selected currency is based on trading price history that has experienced the most extreme deviations between subsequent prices. Intuitively, measuring the σ for each set of prices to measure volatility seems a logical solution. Still, this method is erroneous, where currencies contain differing min and max price values. For example, for a currency c_1 the σ of currency prices 1 & 100 is 49.5, for a currency c_2 the σ of currency prices .1 & 10 is 4.95. Selecting for the currency with the highest volatility, then c_1 seems correct as 49.5 > 4.95 but the price changes in terms of magnitude are equal, each has risen by 100 orders of magnitude. 'ZRX-EUR' is selected due to it's high volatility in its trading price.

Data is processed such that the agent can select buy, hold or sell actions from $t+1$ to $t+49$ time steps in the future. Therefore, there are a total of 147 actions the agent can choose from at each time step.

4.4. Preprocessing

In its raw format from the Coinbase exchange, prices are at the sub-minute level. I calculate the mean of prices for each minute and convert to a pricing window of n minutes. If data is not available for a minute at time t then the mean of price at $t-1$ is imputed for minute t.

An improvement is instead of imputing the mean price for minute t use a confidence interval to determine if a sample mean price is representative of the true mean for a given interval as follows:

4.4.1. Experiment Definition

The collection of cryptocurrency prices for a given time interval is defined as t. Null hypothesis: The mean price for BTC-EUR for a given sample from the rest service is equal to the mean price from the web socket for t. Alternative hypothesis: a significant difference in mean price for BTC-EUR exists for t.

4.4.2. Experiment Implementation

Using the realtime API receive a price attribute for BTC-EUR approx every 10ms. A sample of 350 prices over 3.5 seconds for BTC-EUR from the Coinbase exchange is accessed using the realtime API.

The population for a given price interval is 1 minute and as the σ of the population of price data is not known, the confidence interval is determined using a sample standard deviation.

A random sample of 10 prices is sampled from 350 prices to produce the following:

$\{10415.8510422.2610512.8410423.1910422.2810422.2110424.610414.7510433.4410423.72\}$

which has μ 10431.514. The confidence interval based on this sample is $(10415.813, 10447.214)$. As 10431.514 is within the confidence interval then 10431.514 could be used to impute the mean for the given interval.

This experiment assumes access to realtime data, which for this research was not available.

Experiment source code [1]

1. https://gist.github.com/aronayne/538eed4d41fcf596faebd39925afe0e9

4.4.3. Normalisation

DQN approximates the state action mapping using a neural network where the network inputs are the state attributes and its outputs are the actions selected by the network. Normalisation results in an error surface that is more suited for backpropagation. Geoffrey Hinton refers to normalisation as "Scaling the inputs" in order to achieve unit variance, unit variance refers to a σ of 1

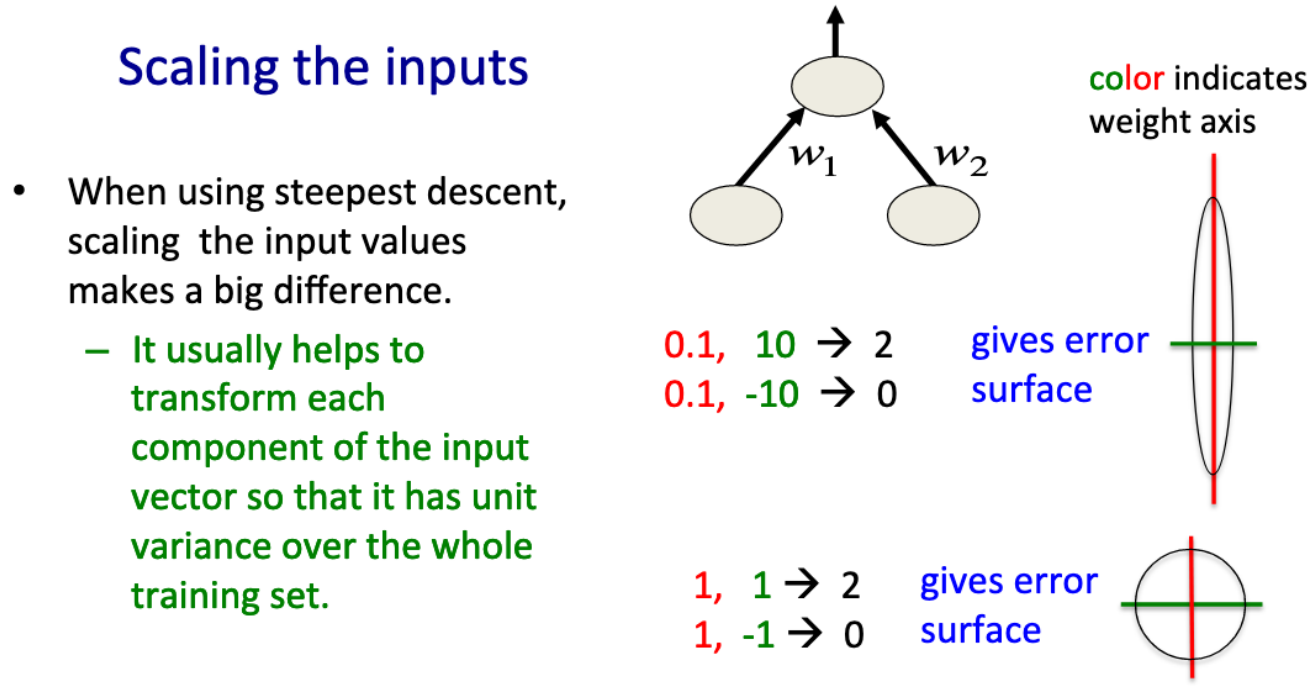

Figure 6. Scaling the inputs (src: Geoffrey Hinton, Scaling the Inputs, 2020)

4.5. Reward calculation

Pricing data is processed such that the model can select the action buy, hold or sell from $t+1$ to $t+49$ time steps into the future. Therefore, there are a total of 147 actions the agent can choose from at each time step.

Rewards for buy, sell and hold actions are each calculated for $\{t+1, t+2 \ldots t+49\}$ as follows:

- Buy reward for time $\{t+1, t+2 \ldots t+49\}$ is the price at time $t+1$ subtracted from price at time t
- Sell reward for time $\{t+1, t+2 \ldots t+49\}$ is the price at time t subtracted from price at time $t+1$
- To encourage the model not to hold negative reward (-1) is applied to all hold positions.

4.5.1. Worked example

Example: $\{t+1, t+2, t+3\}$ and $t+1$ represents 1 minute of data.

Here an example is provided for reward calculation of reward 3 time steps into the future. Using the same process the RL model utilises data from 47 time steps into the future.

For a sequence of prices $\{100, 110, 90, 95\}$ that maps to prices $\{t, t+1, t+2, t+3\}$ the actions $\{Buy, Sell, Hold\}$ at each time step $\{t+1, t+2, t+3\}$ are calculated as follows:

Example: Reward calculation

Price at time t: 100

Time	Price	Buy Reward	Sell Reward	Hold Reward
t+1	110	110-100=10	100-110=-10	-1
t+2	90	90-100=-10	100-90=10	-1
t+3	95	95-100=-5	100-95=5	-1

5. Chapter 5

For this chapter an overivew of the implemented models and web application is described. The RL models Q learning and DQN have been applied to the following : RL agent utilises Q learning to play the game tic tac toe. RL agent utilises DQN to generate trading actions. In both SARSA and Q Learning the state value function is updated via a lookup table.

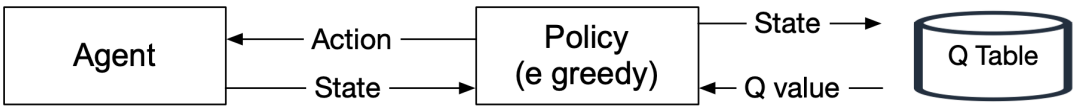

Figure 7. Q table lookup

DQN utilises a neural network instead of a 'Q table lookup' to map states to action values.

5.1. Tic Tac Toe

SARSA and Q learning are each evaluated in learning how to play the tic tac toe game. The RL agent learns where to place moves from self-play against randomly generated moves.

5.1.1. SARSA

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'; A \leftarrow A';$
  until S is terminal

Figure 8. SARSA algorithm (Sutton and Barto, 2018, p141)

SARSA is an acronym for State, Action, Reward, Next State, Next Action. An RL agent in state S takes action A which results in a reward R. Subsequently, the agent is then in the next state S' and takes the following action A' in an iterative process.

SARSA, in performing the temporal difference update subtracts the discounted Q value of the next state and action, S', A' from the Q value of the current state and action S, A. The current action is assigned to the next action at the end of each episode step, whereas Q-learning does not assign the current action to the next action at the end of each episode step.

The SARSA update function :

$$Q(S, A) = Q(S, A) + \alpha(R_{t+1} + \gamma Q(S', A') - Q(S, A))$$
$$S \leftarrow S'; A \leftarrow A'$$

Figure 9. SARSA update

implemented as Python code:

```
q_table[state][action] = q_table[state][action] + (step_size * (reward + (discount *
q_table[next_state][next_action]) - q_table[state][action]))
```

5.1.2. Q-Learning

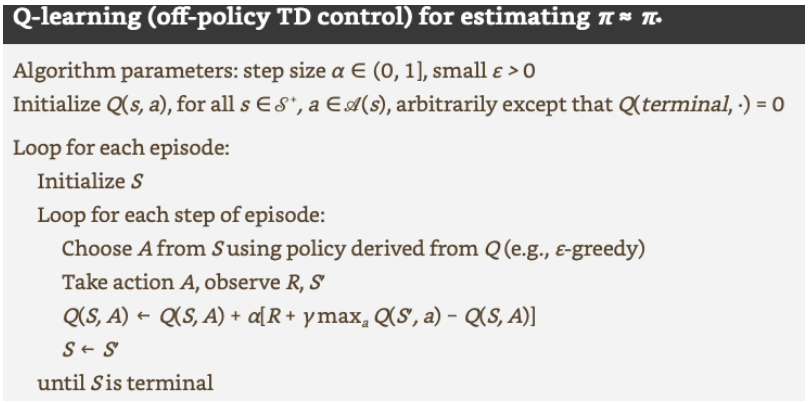

Figure 10. Q learning algorithm (Sutton and Barto, 2018, p143)

Q-learning is a model-free RL algorithm, therefore does not utilise a transition probability distribution when moving from state s at time t to state s' at time t+1. Q-Learning is very similar to SARSA, but SARSA, unlike Q-learning, does not include the max Q value as part of the temporal difference learning update. Q-learning takes the discounted difference between the max action value for the Q value of the next state S' and current action A.

The goal of Q-learning is to "learn a policy telling an agent what action to take under what circumstances" (Q-learning doi: https://en.wikipedia.org/wiki/Q-learning).

Q Learning can overestimate the value associated with actions, more specifically, values for certain actions may be incorrectly predicted to have a higher value which can result in an incorrect action for a given state being selected. The overestimation is irrelevant if the error in estimation is constant for all action value estimates. Hasselt et al (2015) provide a method for addressing this issue. Atari games trained with DQN has shown to overestimate Q values.

The Q learning update function :

$$Q(S,A) = Q(S,A) + \alpha(R_{t+1} + \gamma \max_a Q(S',A) - Q(S,A))$$
$$S \leftarrow S'$$

Figure 11. Q learning update

implemented as Python code:

```
q_table[state][action] = q_table[state][action] + (step_size * (reward + (discount *
np.max(q_table[next_state]) - q_table[state][action])))
```

5.2. Model Implementation

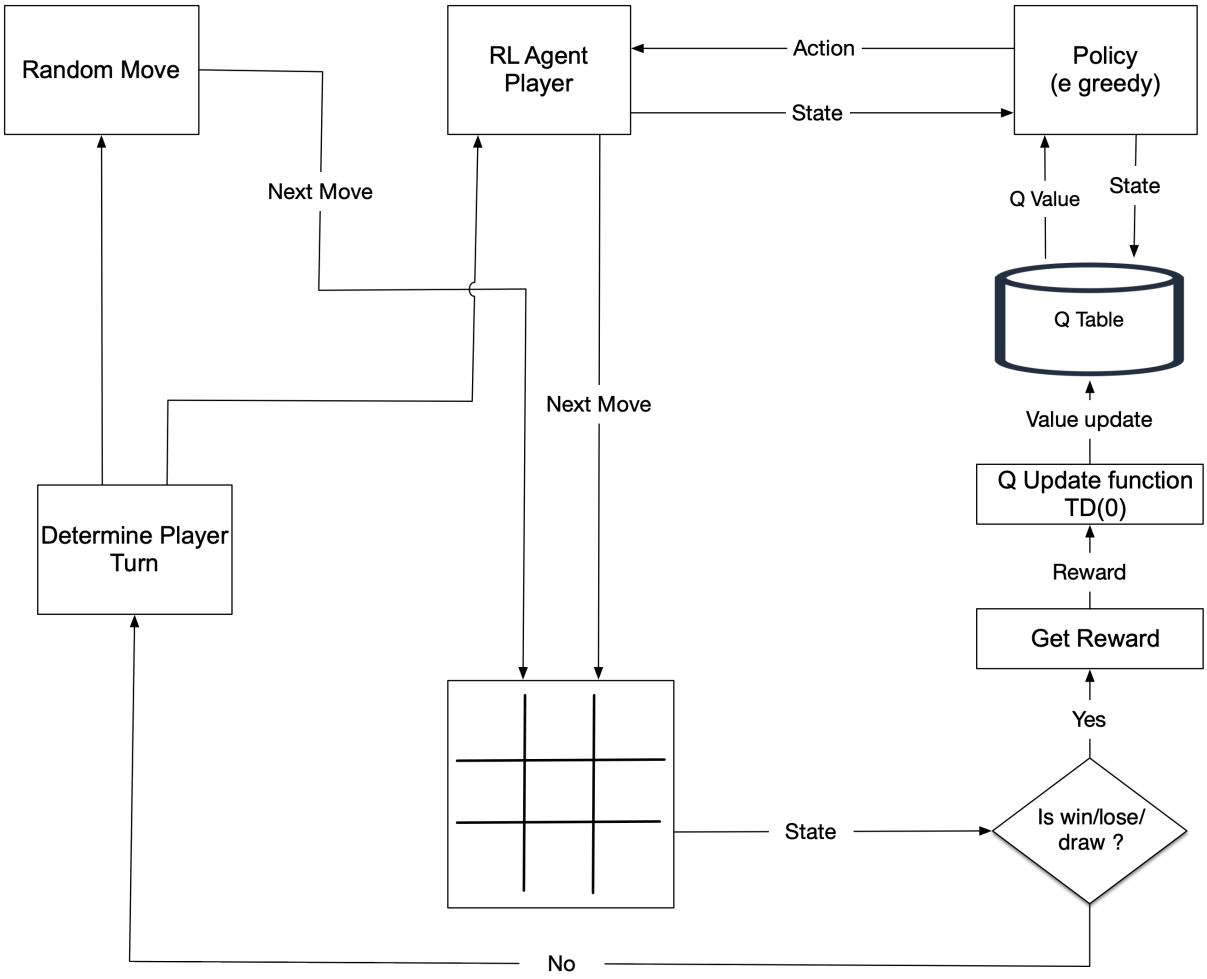

Figure 12. Q-learning and SARSA implementation

5.3. Trading actions

5.3.1. Deep Q Network (DQN)

A neural network is trained to output the Q value for a given state-action pair. The output of the neural network is a vector where each element of the vector is the learned value of a prediction for a given state. The action of the agent is chosen based on the output of the Q function mapping. The Q function follows the epsilon-greedy policy.

DQN approximates the state action mapping and utilises the backpropagation algorithm. The pricing data received from the exchange is normalised. The backpropagation algorithm works better on normalised data in the calculation of gradients for performing gradient descent.

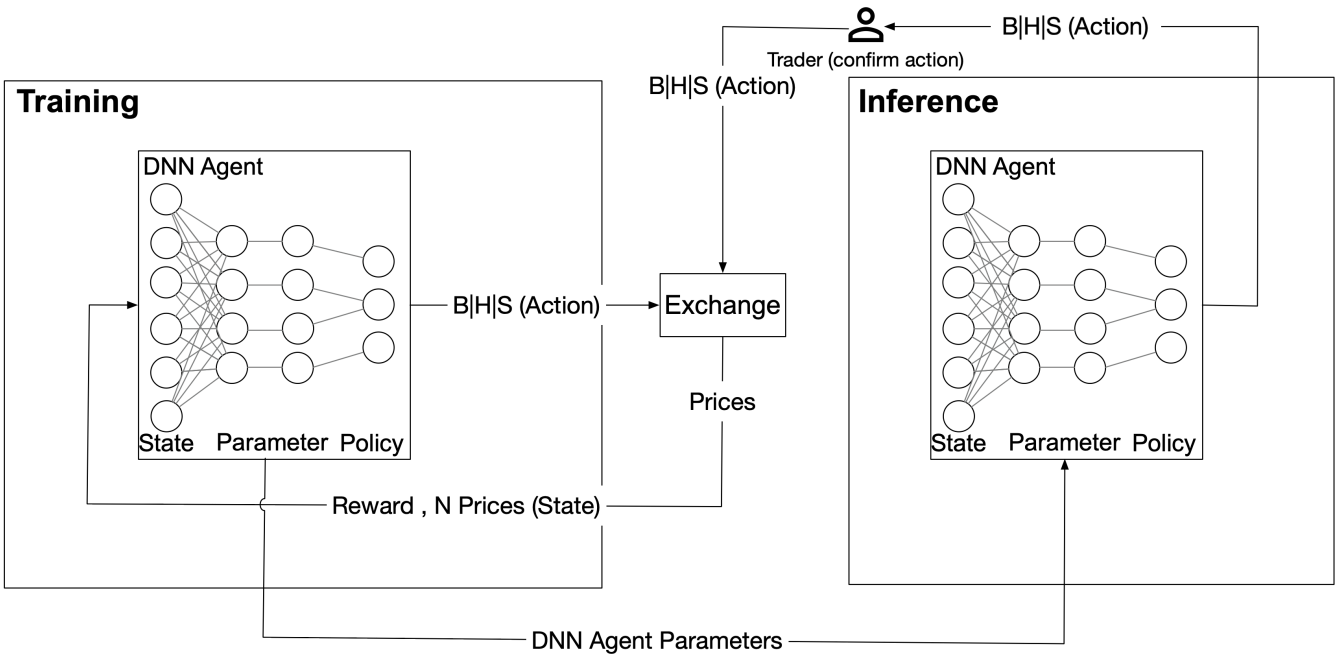

Figure 13. Deep Q Network (DQN) implementation

5.4. Application

A web application showcasing the models described has been implemented. The source code for this application is available at [1]

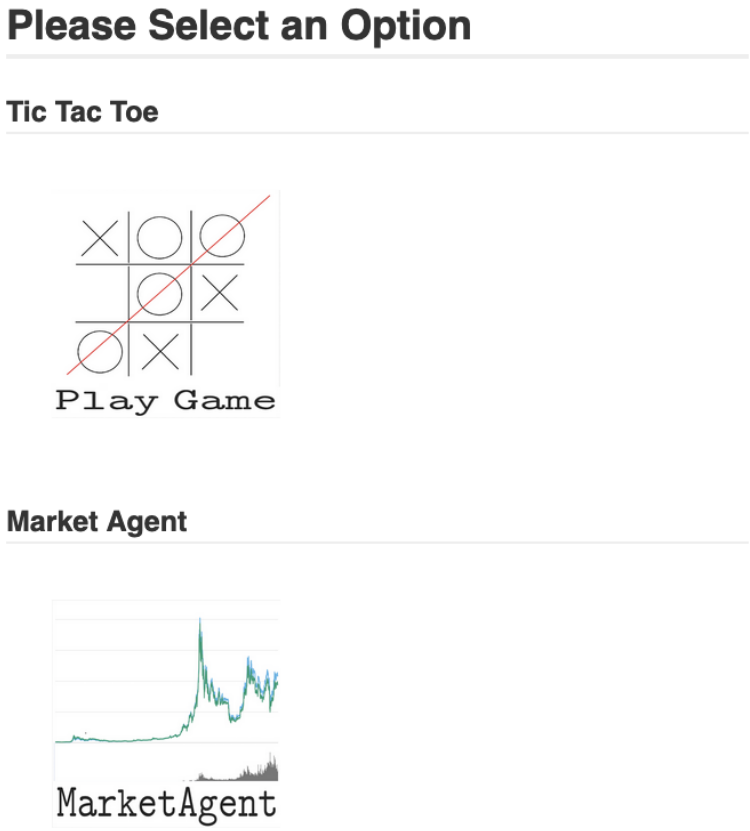

Figure 14. Application options screen

5.4.1. Tic tac toe

The client is a browser. Python on Jupyter notebooks is used to develop the models.

For the tic tac toe agent, a REST service exposes RL agent moves, read from the Q table.

Tic Tac Toe Player Screen

The user can interact and play with the learned RL agent Q table via playing a game.

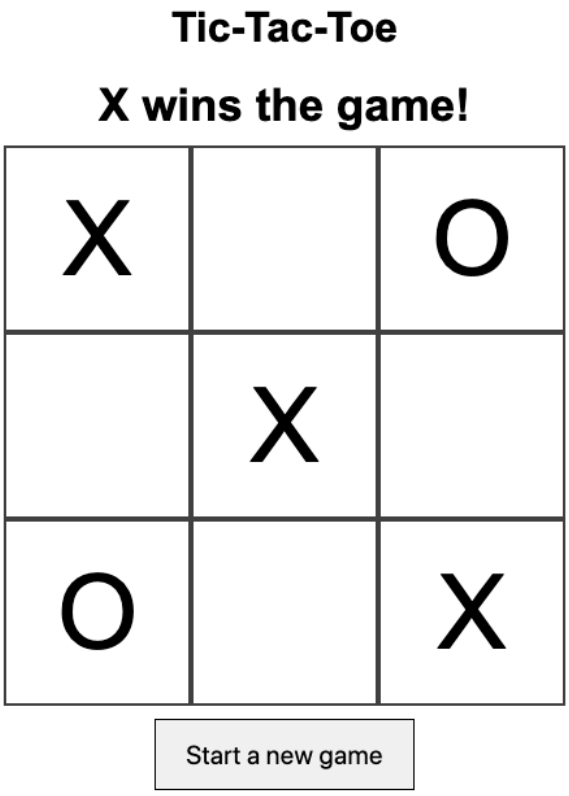

Figure 15. Tic Tac Toe player screen

5.4.2. Market agent

An Akka action reads from the exported predictions file and sends a message once per second to a downstream WebSocket. The browser (client) listens on this socket connection and updates the screen as new prediction messages are received.

Market Agent Screen

The best action for each time interval is displayed on the screen.

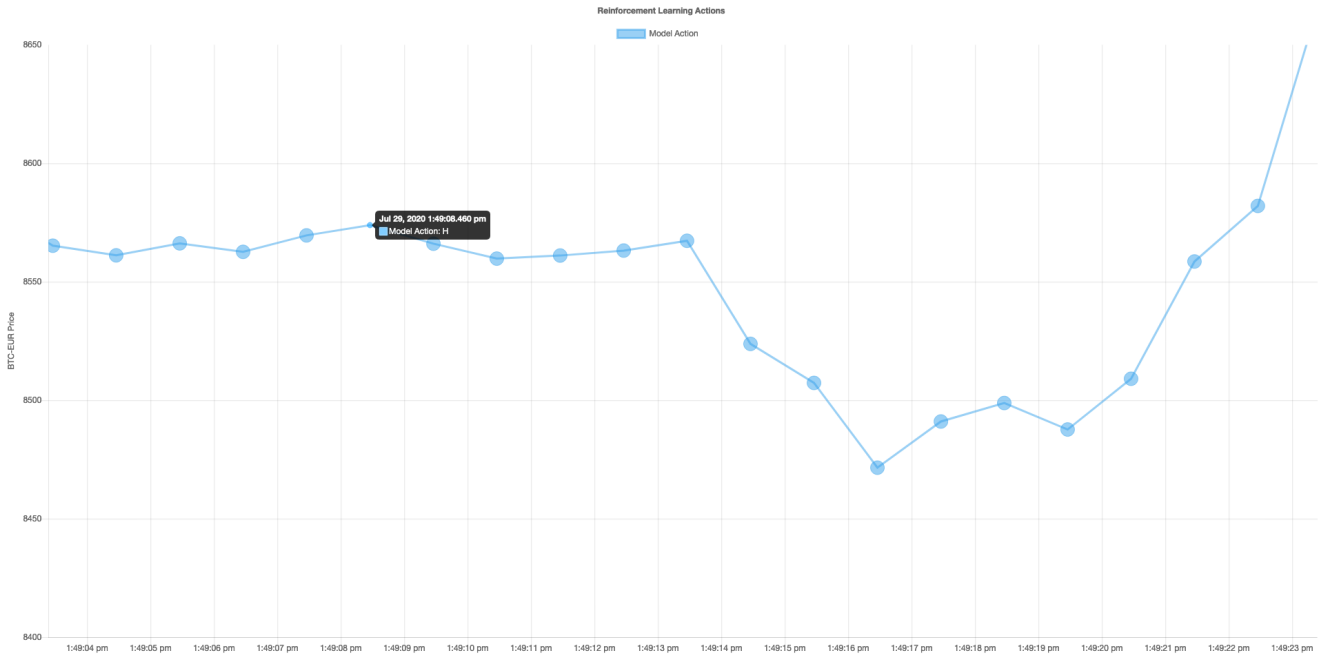

Figure 16. Market agent screen

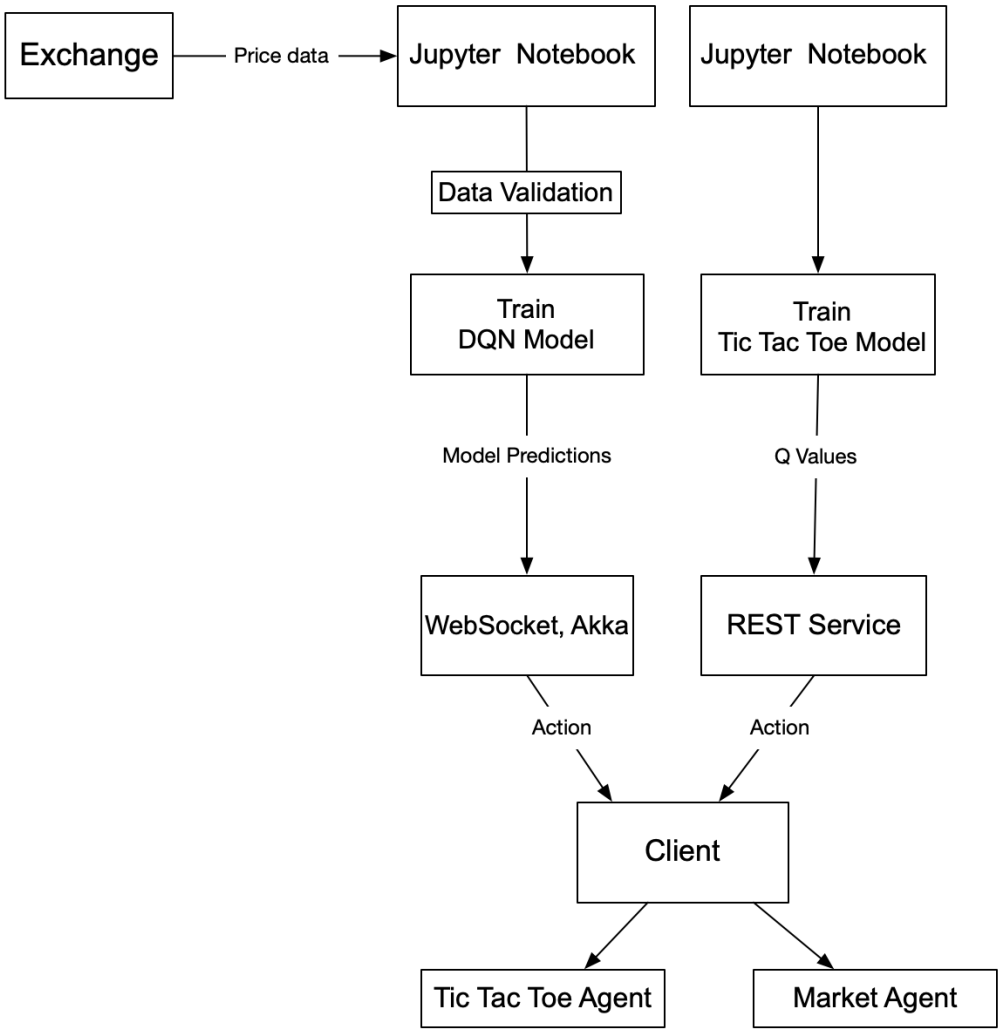

Figure 17. Application data flows high level architecture

1. https://github.com/aronayne/marketagent

6. Chapter 6

Source code for algorithms is provided within this chapter. Q Learning setup and implementation code contains detailed code comments to explain the implementation.

6.1. Q Learning Source Code

The algorithm player is assigned to 'X' while the other player is 'O'. The model learns by playing against an opponent making random moves.

6.1.1. Q Learning Setup Code

```
today = datetime.today()
model_execution_start_time = str(today.year)+"-"+str(today.month)+"-"+str(today.day)+" "+str(today.hour)+":"+str(today.minute)+":"+str(today.second)
epsilon = .1 ①
discount = .1  ②
step_size= .1  ③
number_episodes = 30000 ④

def epsilon_greedy(epsilon, state, q_table) :

    def get_valid_index(state):
        i = 0
        valid_index = []
        for a in state :
            if a == '-' :
                valid_index.append(i)
            i = i + 1
        return valid_index

    def get_arg_max_sub(values , indices) :
        return max(list(zip(np.array(values)[indices],indices)),key=lambda item :item[0])[1]

    if np.random.rand() < epsilon:
        return random.choice(get_valid_index(state))
    else :
        if state not in q_table :
            q_table[state] = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0])
        q_row = q_table[state]
        return get_arg_max_sub(q_row , get_valid_index(state))

def make_move(current_player, current_state , action):
    if current_player == 'X':
        return current_state[:action] + 'X' + current_state[action+1:]
    else :
        return current_state[:action] + 'O' + current_state[action+1:]

q_table = {} ⑤
```

```
max_steps = 9 ⑥
reward_per_episode = [] ⑦
reward = []

def get_other_player(p): ⑧
    if p == 'X':
        return 'O'
    else :
        return 'X'

def win_by_diagonal(mark , board): ⑨
    return (board[0] == mark and board[4] == mark and board[8] == mark) or (board[
2] == mark and board[4] == mark and board[6] == mark)

def win_by_vertical(mark , board): ⑩
    return (board[0] == mark and board[3] == mark and board[6] == mark) or (board[
1] == mark and board[4] == mark and board[7] == mark) or (board[2] == mark and board[
5] == mark and board[8]== mark)

def win_by_horizontal(mark , board): ⑪
    return (board[0] == mark and board[1] == mark and board[2] == mark) or (board[
3] == mark and board[4] == mark and board[5] == mark) or (board[6] == mark and board[
7] == mark and board[8] == mark)

def win(mark , board): ⑫
    return win_by_diagonal(mark, board) or win_by_vertical(mark, board) or
win_by_horizontal(mark, board)

def draw(board): ⑬
    return win('X' , list(board)) == False and win('O' , list(board)) == False and
(list(board).count('-') == 0)

s = []
rewards = []
def get_reward(state): ⑭
    reward = 0
    if win('X' ,list(state)):
        reward = 1 ⑮
        rewards.append(reward)
    elif draw(state) :
        reward = -1 ⑯
        rewards.append(reward)
    else :
        reward = 0 ⑰
        rewards.append(reward)

    return reward

def get_done(state):
    return win('X' ,list(state)) or win('O' , list(state)) or draw(list(state)) or
(state.count('-') == 0)a
```

① Controls the exploration/exploitation tradeoff of ϵ greedy
② Discount value, varies between 0 and 1
③ The learning rate of temporal difference update function
④ The number of tic tac toe games to play
⑤ The q table used to store the value for each state
⑥ the maximum nomber of moves per game is 9
⑦ Maintain a list of reward attained for each episode
⑧ Just return the other player name for the current player
⑨ Determine if a win has occured by a diaganol match
⑩ Determine if a win has occured by a vertical match
⑪ Determine if a win has occured by a horizontal match
⑫ Determine if a win has occured
⑬ Determine if a drawn match has occured
⑭ Return the reward for a given state
⑮ +1 reward if RL agent wins a game
⑯ -1 reward if RL agent loses a game
⑰ Do not assign positive or negative reward if RL agent draws with random player

6.1.2. Q Learning Implementation Code

```
q_table = {}
max_steps = 9
reward_per_episode = []
reward = []
cummulative_reward = []

today = datetime.today()
model_execution_start_time = str(today.year)+"-"+str(today.month)+"-"+str(today
.day)+" "+str(today.hour)+":"+str(today.minute)+":"+str(today.second)
epsilon = .1 # <> Controls the exploration/exploitation tradeoff of
latexmath:[$\epsilon$] greedy
discount = .1  # <> Discount value, varies between 0 and 1
step_size= .1  # <> The learning rate of
number_episodes = 30000 # <> The number of tic tac toe games to play

def epsilon_greedy(epsilon, state, q_table) :

    def get_valid_index(state):
        i = 0
        valid_index = []
        for a in state :
            if a == '-' : # <> If a move
                valid_index.append(i)
            i = i + 1
        return valid_index

    def get_arg_max_sub(values , indices) :
        return max(list(zip(np.array(values)[indices],indices)),key=lambda item
```

```
:item[0])[1]
 27
 28     if np.random.rand() < epsilon: ⑤
 29         return random.choice(get_valid_index(state))
 30     else :
 31         if state not in q_table :
 32             q_table[state] = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0])
 33         q_row = q_table[state]
 34         return get_arg_max_sub(q_row , get_valid_index(state))
 35
 36 def make_move(current_player, current_state , action):
 37     if current_player == 'X':
 38         return current_state[:action] + 'X' + current_state[action+1:]
 39     else :
 40         return current_state[:action] + 'O' + current_state[action+1:]
 41
 42 q_table = {} # <> The q table used to store the value for each state
 43 max_steps = 9 # <> the maximum nomber of moves per game is 9
 44 reward_per_episode = [] # <> Maintain a list of reward attained for each episode
 45 reward = []
 46
 47 def get_other_player(p): # <> Just return the other player name for the current
player
 48     if p == 'X':
 49         return 'O'
 50     else :
 51         return 'X'
 52
 53 def win_by_diagonal(mark , board): # <> Determine if a win has occured by a
diaganol match
 54     return (board[0] == mark and board[4] == mark and board[8] == mark) or (board
[2] == mark and board[4] == mark and board[6] == mark)
 55
 56 def win_by_vertical(mark , board): # <> Determine if a win has occured by a
vertical match
 57     return (board[0] == mark and board[3] == mark and board[6] == mark) or (board
[1] == mark and board[4] == mark and board[7] == mark) or (board[2] == mark and board
[5] == mark and board[8]== mark)
 58
 59 def win_by_horizontal(mark , board): # <> Determine if a win has occured by a
horizontal match
 60     return (board[0] == mark and board[1] == mark and board[2] == mark) or (board
[3] == mark and board[4] == mark and board[5] == mark) or (board[6] == mark and board
[7] == mark and board[8] == mark)
 61
 62 def win(mark , board): # <> Determine if a win has occured
 63     return win_by_diagonal(mark, board) or win_by_vertical(mark, board) or
win_by_horizontal(mark, board)
 64
 65 def draw(board): # <> Determine if a drawn match has occured
 66     return win('X' , list(board)) == False and win('O' , list(board)) == False and
```

```
(list(board).count('-') == 0)
 67
 68 s = []
 69 rewards = []
 70 def get_reward(state): # <> Return the reward for a given state
 71     reward = 0
 72     if win('X' ,list(state)):
 73         reward = 1 # <> +1 reward if RL agent wins a game
 74         rewards.append(reward)
 75     elif draw(state) :
 76         reward = -1 # <> -1 reward if RL agent loses a game
 77         rewards.append(reward)
 78     else :
 79         reward = 0 # <> Do not assign positive or negative reward if RL agent
draws with random player
 80         rewards.append(reward)
 81
 82     return reward
 83
 84 def get_done(state):
 85     return win('X' ,list(state)) or win('O' , list(state)) or draw(list(state)) or
(state.count('-') == 0)
 86 def q_learning():
 87
 88     total_reward = 0
 89
 90     for episode in range(0 , number_episodes) :
 91         latexmath:[$t$] = 0
 92         state = '---------' # <> Set the initial empty state to '---------', each
'-' represents a move has not beed made for the given position
 93
 94         player = 'X'
 95         random_player = 'O'
 96
 97
 98         if episode % 10000 == 0:
 99             print('in episode:',episode)
100
101         done = False
102         episode_reward = 0
103
104         while latexmath:[$t$] < max_steps:
105
106             latexmath:[$t$] = latexmath:[$t$] + 1
107
108             action = epsilon_greedy(epsilon , state , q_table)
109
110             done = get_done(state)
111
112             if done == True :
113                 break
```

```
114
115             if state not in q_table :
116                 q_table[state] = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]) #
<> Set the Q values for the next state if the state does not yet exist in Q lookup
table
117
118             next_state = make_move(player , state , action)
119             reward = get_reward(next_state)
120             episode_reward = episode_reward + reward
121             total_reward = total_reward + reward
122             done = get_done(next_state)
123
124             if done == True : # <> if the game is over
125                 q_table[state][action] = q_table[state][action] + (step_size *
(reward - q_table[state][action]))
126                 break
127
128             next_action = epsilon_greedy(epsilon , next_state , q_table)
129             if next_state not in q_table :
130                 q_table[next_state] = np.array([0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0
,0.0]) # <> Set the Q values for the next state if the state does not yet exist in Q
lookup table
131
132             q_table[state][action] = q_table[state][action] + (step_size * (reward
+ (discount * np.max(q_table[next_state]) - q_table[state][action]))) # <> Q learning
temporal difference update function
133
134             state = next_state
135
136             player = get_other_player(player)
137
138         reward_per_episode.append(episode_reward)
139         cummulative_reward.append(total_reward)
140
141 q_learning()
```

6.1.3. SARSA Implementation Code

Source code of the Q Learning and SARSA algorithm's used for the game tic tac toe is located at [1].

Primary difference is the method of updating the q value:

```
q_table[state][action] = q_table[state][action] + (step_size * (reward + (discount *
q_table[next_state][next_action]) - q_table[state][action]))
```

6.1.4. DQN Implementation Code

A static data file with price data in date range 2020-06-09 to 2020-07-24 is processed with DQN. The data file is available at [2]

DQN Source code location: [3]

DQN results analysis source code location: [4]

1. https://github.com/aronayne/public/blob/master/RL-tic-tac-toe-V2.ipynb
2. https://raw.githubusercontent.com/aronayne/public/master/btc-df.csv
3. https://github.com/aronayne/public/blob/master/dqn/model.py
4. https://github.com/aronayne/public/blob/master/dqn/results-generation.py

7. Chapter 7

Q Learning, SARSA and DQN results are now explored. Q Learning and SARSA are applied to the tic tac toe game while DQN is applied to the trading model.

7.1. Model Results - Tic Tac Toe

30'000 episodes are played against a player taking random moves for Q learning and SARSA. On a trained Q Learning and SARSA model the RL agent plays games against a player that takes random moves.

7.1.1. Q Learning results

The number of games played: 50'000

The number of X (RL agent) game wins: 22'004

The number of O (player taking random moves) game wins: 14'992

The number of drawn games: 13'004

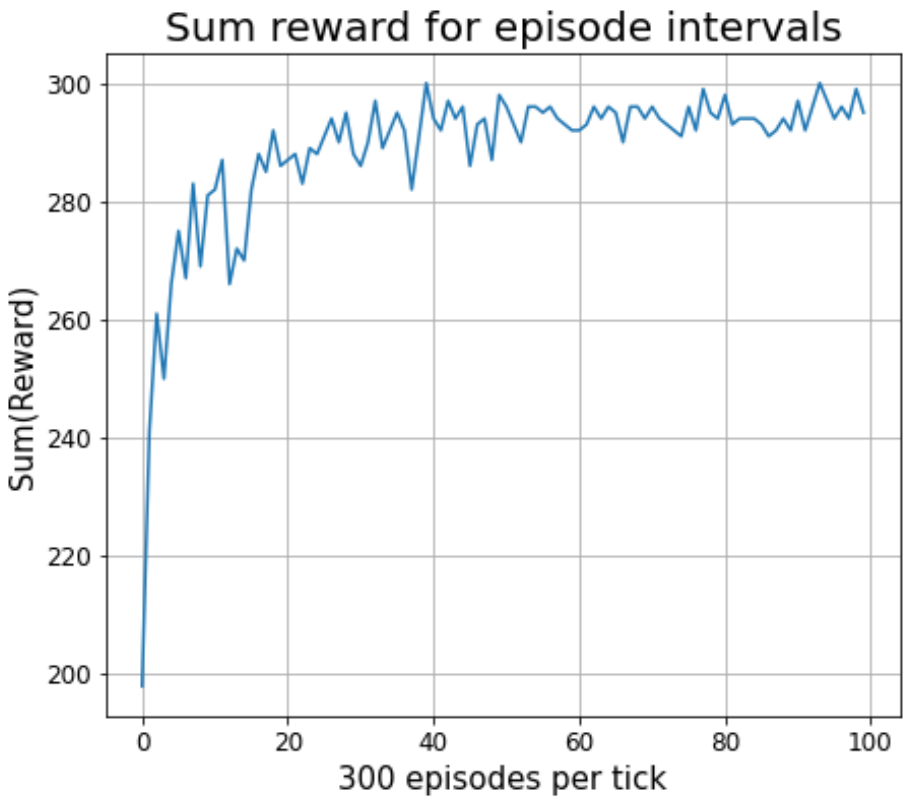

Figure 18. q learning reward sum

7.1.2. SARSA results

The number of games played: 50'000

The number of X (RL agent) game wins: 21'933

The number of O (player taking random moves) game wins: 15'118

The number of drawn games: 12'949

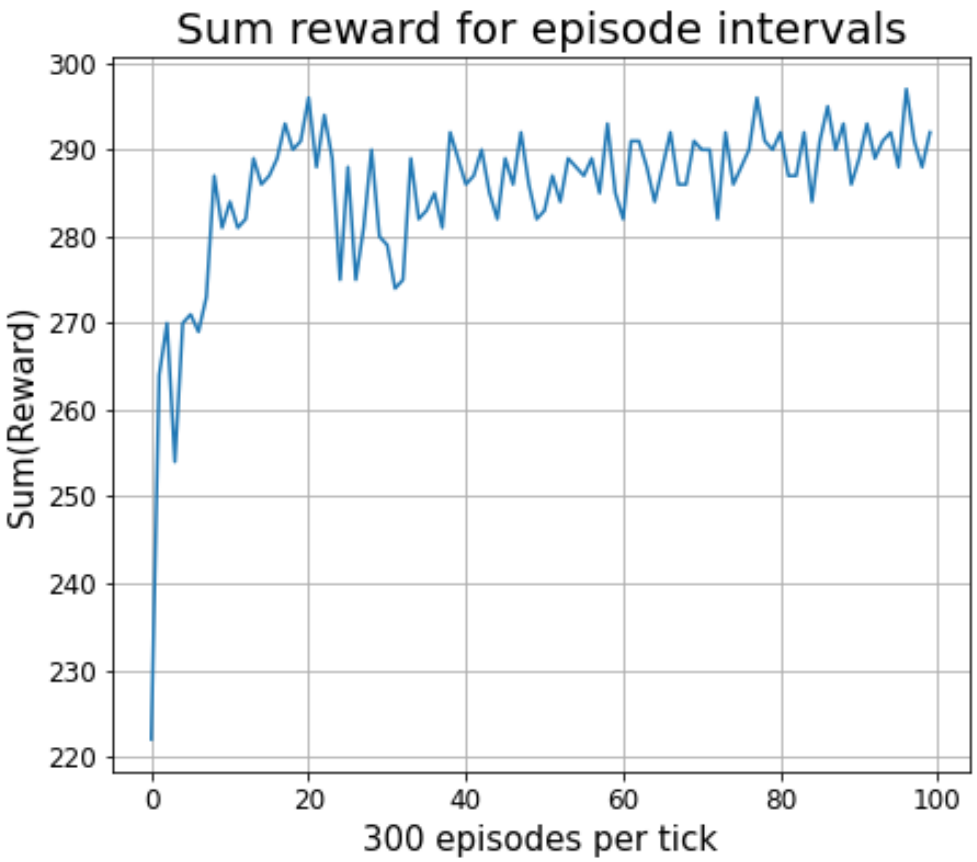

Figure 19. Sarsa reward sum

The results suggest Q Learning produces higher reward values than SARSA. For each plot of the sum of reward each tick tick represents the sum of rewards for 300 episodes games).

7.2. Model Results - Trading

Each of the following experiments use the following hyperparamter values:

Policy Network learning rate: .0001

Policy Network optimizer: 'adam'

e = .1

γ = .9

e decay = .01

Number of episodes: 5

7.2.1. Variability between experiments

Due to the random initialisation of the policy network weights some experiments perform better than others. For example, experiment 2 returns a total reward of -368.996 while experiment 12 returns a total reward of 194.436. More detailed results of experiments 2 and 12 are below:

Experiment 2 - Total reward -368.996

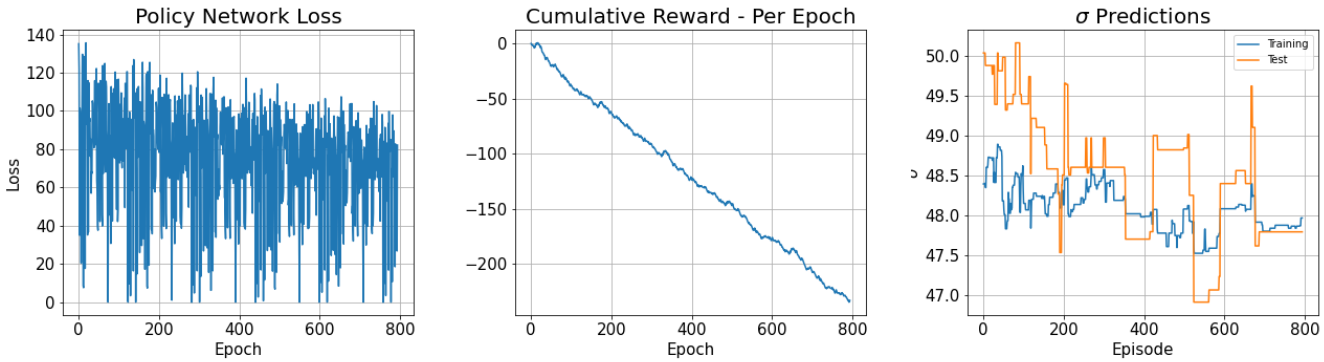

Figure 20. Experiment 2 - Policy network, cumulative reward...

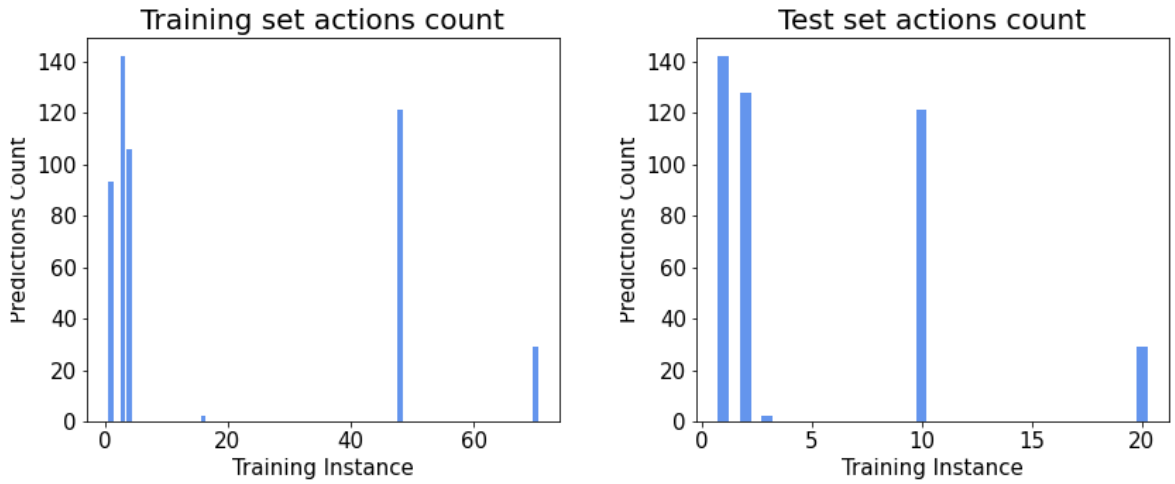

Figure 21. Experiment 2 - training set actions count, test set actions count

Experiment 12 - Total reward 194.436

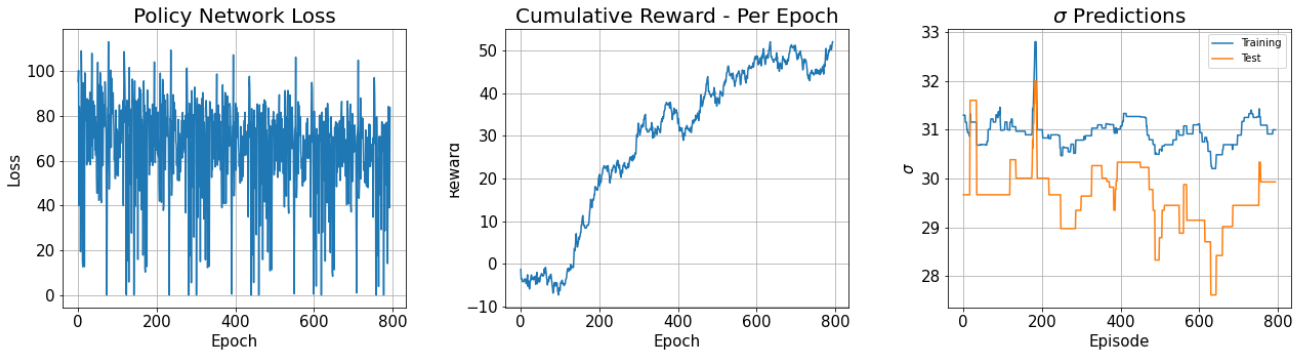

Figure 22. Experiment 12 - Policy network loss, cumulative reward...

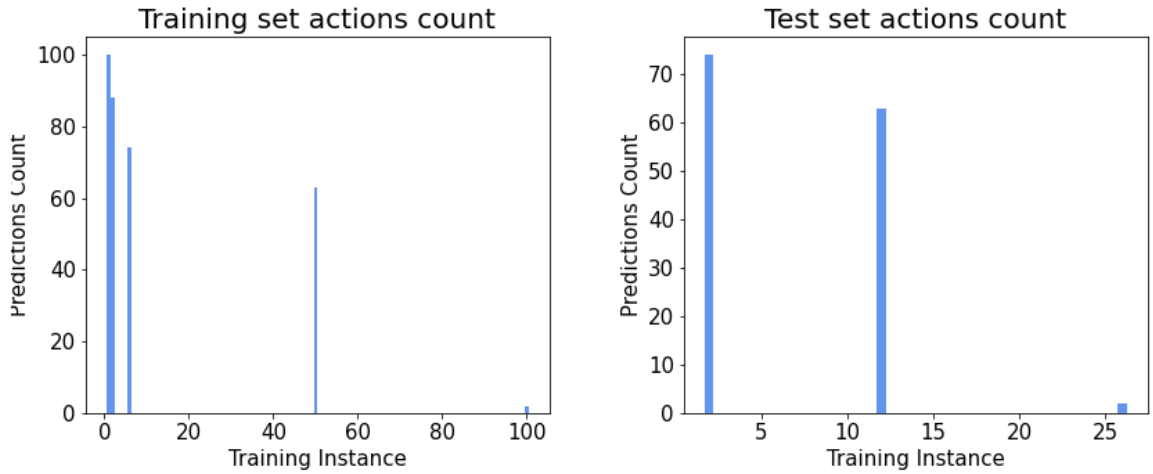

Figure 23. Experiment 12 - training set actions count, test set actions count

The results of more experiments are detailed below:

Experiment results, Optimizer: adam

Price at time t: 100

Experiment Number	All episodes reward sum
1	-1209.858
2	-368.996
3	-1881.011
4	-531.992
5	-864.310
6	-1793.888
7	-560.318
8	-443.884
9	-632.075
10	-1765.696
11	-1132.926
12	194.436
13	-1415.693
14	-163.495
15	-717.192

Each of the following experiments use the following hyperparameter settings, difference to previous experiments is SGD is used as the 'Policy Network optimizer' instead of adam:

Policy Network learning rate: .0001

Policy Network optimizer: 'SGD'

e = .1

γ = .9

e decay = .01

Number of episodes: 5

Experiment results, Optimizer: SGD

Price at time t: 100

Experiment Number	All episodes reward sum
1	-1155.874
2	-.448.163
3	-863.908
4	-2129.131
5	-1868.761
6	-331.426
7	-1903.327
8	-571.252
9	-1405.962
10	-247.345
11	-777.818
12	-1580.937
13	-1254.289
14	-621.546
15	-928.763

7.2.2. Optimiser selection

The above results suggest SGD produces higher negative reward values, this is further emphasized by the following graph:

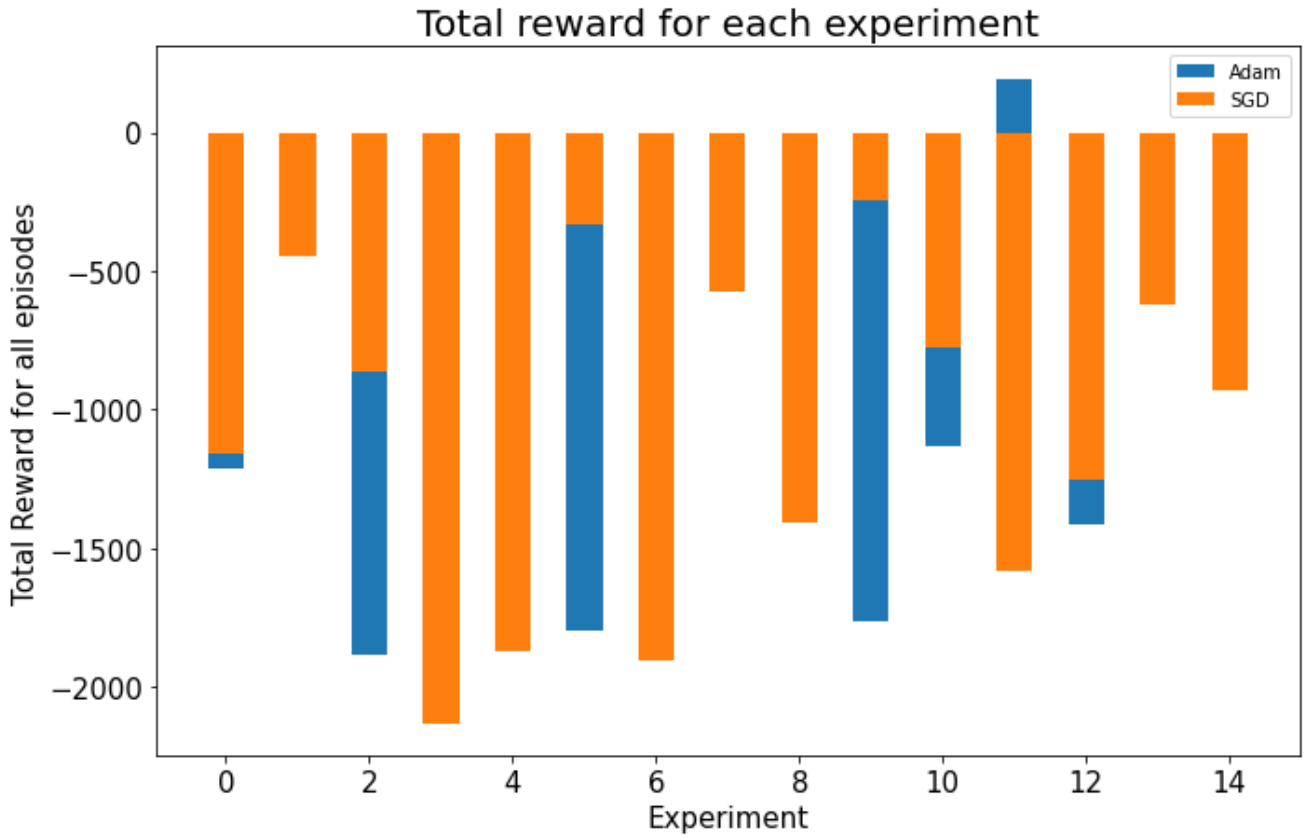

Figure 24. Adam versus SGD Optimiser

7.3. Results Analysis

The data is split into training, and test sets proportions of 80% and 20%, which results in 160 training set instances and 40 test set instances. Each instance contains 50 attributes where each attribute is 1 minute of price data. Each attribute is the mean price for a given minute. Therefore each instance consists of an interval of 50 minutes of price data. Due to its high volatility, the data for cryptocurrency 'ZRX-EUR' is used to train and test the RL model.

7.3.1. Q learning & SARSA

Q learning performs marginally better than SARSA. Form a total of 50'000 games the number of won games using Q learning is 22'004, while with SARSA the number of won games is 21'933. The area under the curve reward sum values for Q learning and SARSA marginally differ. It is clear that both Q learning and SARSA both improve their playing strategy over time.

7.3.2. DQN

Reflected by "Figure 24. Adam versus SGD Optimiser", optimizer selection directly impacts on how the policy network approximates states to actions. Applying the adam optimizer to optimize the policy network neural network weights rather than the SGD optimizer has a significant impact on model results. The adam optimizer results in a higher overall reward being attained. Other than SGD and adam, many other optimizers exist, this includes Momentum, Nesterov, Adagrad, Adadelta, RMSProm and Nadam. Trying other optimizers could lead to improved results. For all experiments, e is decayed by 0.01 for each episode and does not have a significant impact on results.

During training the standard deviation of actions the model selects are initially high but tend towards a small number of actions for all states on the training and test sets.

The experiments detailed previously "Experiment 2" & "Experiment 12" indicate a correlation between the σ of predictions and reward attained. As the σ of predictions decreases the total reward attained by the model increases.

8. Chapter 8

8.1. Model Improvements

Various improvements to the model can be applied. An example is employing experience replay.

8.1.1. Experience Replay

An experience is defined as the tuple (state, action, reward, next state). For each episode, if the experience replay memory is populated, the DQN policy is updated with a random sample from the experience memory list. The experience memory list structure is first in, first out, also known as a FIFO queue. This type of list structure ensures precedence is applied to recent experiences during policy updates. During the training of the model, the experience memory list is populated per training epoch with n experience's where n is a hyperparameter. The value of n can depend on the episode length. For example, if the number of states in an episode is 1000, and we wish to learn from the most recent 10% of experiences, set the memory length to 900. Experience replay can prevent the agent from getting stuck in a local optimum.

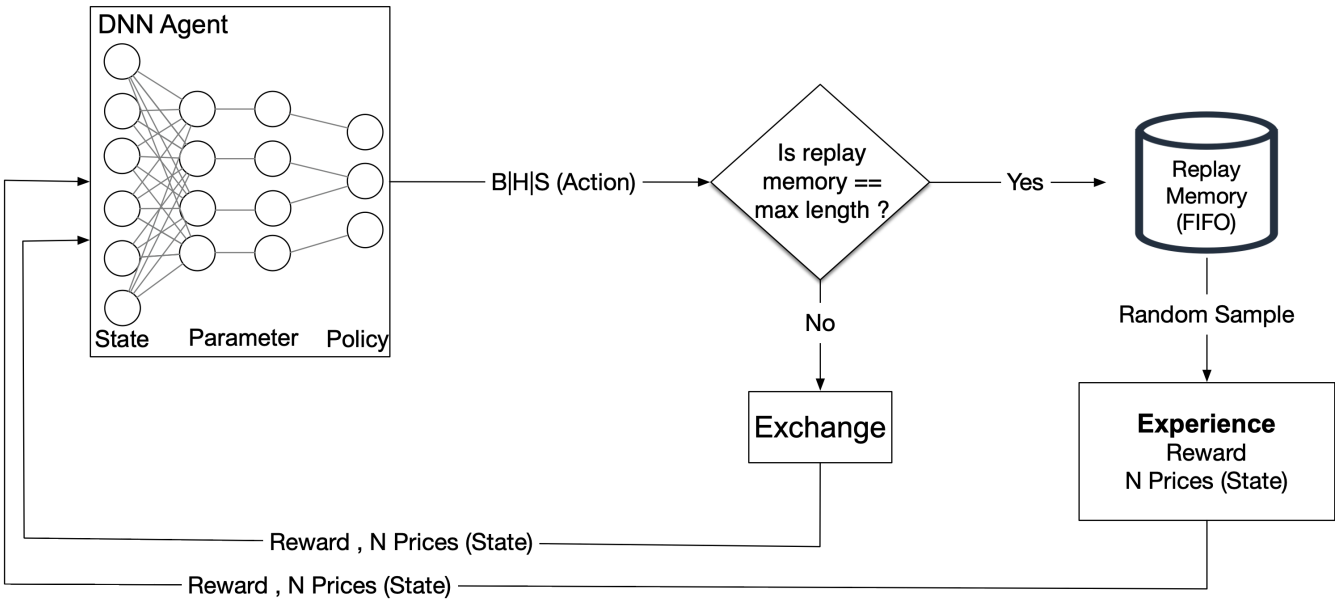

Figure 25. Experience Replay implementation

8.2. Conclusion

Domain knowledge should not be discounted when implementing an RL solution to any given problem. For example, domain knowledge is required in order to select cryptocurrencies that represent the maximum trading opportunity.

For a given interval if pricing data is unavailable, then the mean of the previous price is imputed for the inaccessible data. Imputation of data using this method may be adding noise to the training data. A deeper and better understanding of the data may lead to more relevant data imputation.

The RL model uses a simple solution to calculate the trading cost for each action. A 1% trading cost is applied to all reward values. In a live trading environment, the trading cost can significantly vary from 1%. As trading costs vary depending on market volatility, a significant portion of any positive return may be deducted from the reward attained.

For this research, each environment has been custom-built and is decoupled from the RL

algorithm implementation. Crucial to the implementation of a successful RL algorithm is deciding how the environment generates observations and reward for a given state while the RL agent operates in the environment. Discerning problems with reinforcement learning models is difficult due to the difficulty of determining if the problem is due to the model, the model implementation code or the model data. Utilising pre-built model environments such as the OpenAI gym [1] may result in reduced time designing and developing models.

The literature that describes RL for trading appears to overlook how an RL agent influences the environment as a result of its trading actions, which can be described as influence learning versus observation learning. Influence learning refers to the agent learning to take actions that result in being in desirable states where the actions influence the environment. Observation learning refers to the agent learning using its observations to take actions that result in the agent being in desirable states. The key difference is for observation learning the agent is not learning how the actions influence the environment, similar to a bandit problem. For example, a trading RL agent may perform a buy action at time t. At time $t+1$ the currency price has changed, and the agent has moved into a new state. The change in currency price may not have been influenced by the action taken, particularly if the amount of currency bought is insignificant. It is not clear how this scenario affects model learning.

Given the scope of RL, it is challenging to ascertain how an RL agent will perform in a trading environment. Explored in this research is a small facet of RL with an application to trading. Given the RL agent results, it does appear an agent can learn insightful trading actions. The trading agent has not been tested in a live trading environment. Therefore it is inconclusive as to whether the RL agent will influence trading actions on other trading participants.

For this research, the aim was to learn how an RL model can be applied to a problem that meets the following criteria:

- Solving the problem relies on human skill and intuition.
- Data is abundant and easily accessible.

The RL models research criteria:

- Use self-play.
- Use episodic learning.

A stretch goal is to discover how an RL model fits into an overall pipeline to form a software engineering solution.

To deliver on the objectives, I implemented an RL model which improves through self-play to play tic tac toe. Although the RL player does not discover how to play an optimum game, bootstrapping the RL player to play against improved versions of itself may lead to imporved results.

I implemented a version of DQN and applied it to a cryptocurrency dataset. The primary discoveries from the DQN implementation are the model has better convergence properties if the policy network uses the adam optimiser and higher reward is attained when the model reduces the σ of actions taken. I designed and developed a simple engineering architecture that describes incorporating the model into an overall software engineering solution.

RL is an extensive and detailed topic which incorporates many overlapping disciplines that include mathematics, probability, statistics, and engineering. Given the scope of RL and despite the difficult task, it is an intention of this thesis to show that focusing on first principles and a small facet of RL yields exciting and valuable results.

1. https://gym.openai.com

References

Zihao Zhang, Stefan Zohren, and Stephen Roberts "Deep Reinforcement Learning for Trading". arXiv preprint: 1911.10107 (2019) Available at https://arxiv.org/pdf/1911.10107.pdf

Yuqin Dai, Chris Wang, Iris Wang, Yilun Xu (2019), Reinforcement Learning for FX trading, Available at: http://stanford.edu/class/msande448/2019/Final_reports/gr2.pdf

Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Thirtieth AAAI conference on artificial intelligence. 2016. Available at: https://arxiv.org/pdf/1509.06461.pdf

Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581 (2015). Available at: https://arxiv.org/pdf/1511.06581.pdf

Hausknecht, Matthew, and Peter Stone. "Deep recurrent q-learning for partially observable mdps." 2015 AAAI Fall Symposium Series. 2015. Available at: https://arxiv.org/pdf/1507.06527.pdf

Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013). Available at: https://arxiv.org/pdf/1312.5602.pdf

Schaul, Tom, et al. "prioritised experience replay." arXiv preprint arXiv:1511.05952 (2015). Available at: https://arxiv.org/pdf/1511.05952.pdf

Deng, Yue, et al. "Deep direct reinforcement learning for financial signal representation and trading." IEEE transactions on neural networks and learning systems 28.3 (2016): 653-664. Available at: http://cslt.riit.tsinghua.edu.cn/mediawiki/images/a/aa/07407387.pdf

Jiang, Zhengyao, Dixing Xu, and Jinjun Liang. "A deep reinforcement learning framework for the financial portfolio management problem." arXiv preprint arXiv:1706.10059 (2017). Available at: https://arxiv.org/pdf/1706.10059.pdf

Xiong, Zhuoran, et al. "Practical deep reinforcement learning approach for stock trading." arXiv preprint arXiv:1811.07522 (2018). Available at: https://arxiv.org/pdf/1811.07522.pdf

Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. MIT press, (2018).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016. Available at: https://www.deeplearningbook.org/

Jiang, Zhengyao, and Jinjun Liang. "Cryptocurrency portfolio management with deep reinforcement learning." 2017 Intelligent Systems Conference (IntelliSys). IEEE, 2017. Available at: https://arxiv.org/pdf/1612.01277.pdf

Suganuma, Masanori, Shinichi Shirakawa, and Tomoharu Nagao. "A genetic programming approach to designing convolutional neural network architectures." Proceedings of the Genetic and Evolutionary Computation Conference. 2017. Available at https://arxiv.org/pdf/1704.00764.pdf

Hsu, Daniel. "Time series compression based on adaptive piecewise recurrent autoencoder." arXiv preprint arXiv:1707.07961 (2017). Available at https://arxiv.org/pdf/1707.07961.pdf

Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533. Available at https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf

Glavin, Frank G. Towards Inherently Adaptive First Person Shooter Agents using Reinforcement Learning. Diss. Doctoral Dissertation, 2015.

Coinist. 2020. Understanding Cryptocurrency Trading Volume. (ONLINE) Available at: https://www.coinist.io/cryptocurrency-trading-volume/. [Accessed 29 May 2020].

Jerdack, Natalia, et al. "Understanding what drives bitcoin trading activities." arXiv preprint arXiv:1901.00720 (2018).

Investopedia. 2020. Portfolio Weight. (ONLINE) Available at: https://www.investopedia.com/terms/p/portfolio-weight.asp. [Accessed 29 May 2020].

Jain, Prateek, et al. "Parallelizing stochastic gradient descent for least squares regression: mini-batching, averaging, and model misspecification." The Journal of Machine Learning Research 18.1 (2017): 8258-8299.

Nakamoto, Satoshi. Bitcoin: A peer-to-peer electronic cash system. Manubot, 2019.

Mastering the game of Go without human knowledge. 2020. (ONLINE) https://www.nature.com/articles/nature24270. [Accessed 7 August 2020].

CoinMarketCap. 2020. Top 100 Cryptocurrencies by Market Capitalization . (ONLINE) Available at: https://coinmarketcap.com/. [Accessed 29 May 2020].

SMB Capital. 2020. (ONLINE) Available at: https://smbcap.com/. [Accessed 29 May 2020].

The Bitter Lesson. 2020. Rich Sutton (ONLINE) Available at: http://www.incompleteideas.net/IncIdeas/BitterLesson.html. [Accessed 29 May 2020].

History of Bitcoin. 2020. (ONLINE) Available at: https://en.wikipedia.org/wiki/History_of_bitcoin#Prices_and_value_history. [Accessed 29 May 2020].

Data compression ratio. 2020. (ONLINE) https://en.wikipedia.org/wiki/Data_compression_ratio. [Accessed 29 May 2020].

Renaissance. 2020. (ONLINE) https://www.rentec.com/. [Accessed 8 August 2020]

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." Proceedings of the fourteenth international conference on artificial intelligence and statistics. 2011.

David Silver, Student Markov Chain. 2020. (ONLINE) https://www.davidsilver.uk/wp-content/uploads/2020/03/DP.pdf. [Accessed 29 July 2020].

Coinbase Pro. (ONLINE) https://pro.coinbase.com/markets. [Accessed 27 August 2020].

Geoffrey Hinton, Scaling the Inputs. (ONLINE) https://www.youtube.com/watch?v=Xjtu1L7RwVM. [Accessed 27 August 2020].

Geoffrey Hinton, Scaling the Inputs. (ONLINE) https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. [Accessed 28 August 2020].