A Machine Learning Approach to Improve the Pegging Algorithm

by Justin Li

A Machine Learning Approach to Improve the Pegging Algorithm

by

Justin Li (CID: 00938472)

Department of Mathematics
Imperial College London
London SW7 2AZ
United Kingdom

Thesis submitted as part of the requirements for the award of the
MSc in Mathematics and Finance, Imperial College London, 2018-2019

IMPERIAL COLLEGE LONDON

THESIS DECLARATION FORM

(for candidates whose degree will be awarded by **Imperial College**)

Declaration: I hereby confirm that the work contained in this thesis is my own work unless other wises stated.

Name Chun Yin Justin Li **CID** 00938472

Title of Thesis A Machine Learning Approach to Improve the Pegging Algorithm

Month and year of submission for examination September 2019

Date 10th September 2019

Signature of Candidate Justin

Acknowledgements

I would like to thank Quod Financial for providing data, highspeed computation facilities and guidance for my project. In particular I would like to thank my line manager Leor Shoker who has been a great help through his insightful comments and discussions.

Also, I would like to express my gratitude to my project supervisor Dr. Siorpaes for providing guidance and support whenever needed.

Finally, I would like to thank my family, my friends Will Li and Shijie Zhou, my girlfriend Sophia Hung for their continuous support and engagements.

Contents

1 Introduction

1.1 Background

The evolution and nature of financial trading have always been closely related to technological advancements. For example, transactions in the financial markets have started to migrate to electronic trading platforms since the 1970s as the telecommunication and computer technologies improve and mature. As a result, the trading efficiencies have been vastly improved and it has eventually set the stage for algorithmic trading. As technology continue to advance, we have reached the current age of having an ever-increasing amount of data and unprecedented computing power. This leads to an increased popularity of applying machine learning to algorithm trading, since the machine learning algorithms that deemed to be infeasible in the past due to slow training times, are now fast enough and have rich enough data to be utilized to provide better performances.

In this thesis we will explore ways to improve the performances of a particular algorithmic trading strategy - the pegging algorithm, via machine learning methods such as artificial neural networks for predictions and Q-learning. The thesis will be structured as follows. We will first introduce the traditional pegging algorithm in Section 1.3. After that we will state the approach, the general framework and assumptions made to developing the new pegging algorithm in Section 2. In Section 3 and Section 4 we will explore and analyse the data used in building the new pegging algorithm and providing justification on some statements made in Section 2. The theoretical background and details of the machine learning models used, as well as their training methods, will be discussed in Section 5 and Section 6. We will then analyze and compare the results of the new pegging algorithm to the traditional one in Section 7, and Section 8 concludes the thesis.

1.2 Related Work

Supervised learning and reinforcement learning are two main branches of machine learning that are commonly applied in finance and algorithmic trading. Supervised learning can be used in any algorithms that involve predicting the future. For example, for price predictions, Shen, Jiang and Zhang (2012)[11] presented a one day ahead forecasting model for several stock market indexes based on Support Vector Machines, and tested the trading system based on the produced predictions. For volatility predictions, Chuong Luong and Nikolai Dokuchaev (2018)[3] presented a method to forecast realized volatilities by combining the heterogeneous autoregressive model with the random forest algorithm.

Reinforcement learning is useful when the situation involves maximising some cumulative rewards obtainable from some kind of environment, which is usually the case for trading in general. In Ning, Ling and Jaimungal (2018)[2], a deep reinforcement learning approach is proposed for optimal trade executions, which outperformed the standard TWAP algorithm in 7 out of 9 stocks.

There are also frameworks that are similar to that of this thesis, where supervised learning is combined with reinforcement learning to build an overall algorithmic trading system. For instance, Azhikodan, Bhat and Jadhav (2019)[1] presented a stock trading method that uses a recurrent convolutional neural network to predict stock trends from the financial news, and applies the predictions as a state to a deep deterministic policy gradient-based reinforcement learning agent.

1.3 The Pegging Algorithm

The pegging algorithm is a trading execution strategy that follows the market trends by sending orders at prices which are pegged to some kind of benchmark price plus some kind of offset. The algorithm's goal is to execute trades as passively as possible while being able to execute most if not all of the orders. What we mean by passive is that to have an as low (resp. high) as possible price for a buy (resp. sell) order.

For a buy (resp. sell) order, a typical best effort pegging algorithm pegs the purchase (resp. sell) price to the national best bid/ask price minus (resp. plus) a fixed non-negative offset. The buy (resp. sell) orders are then matched and successfully executed when the pegged prices intersect the ask (resp. bid) price. More formally, assume that we are in a Cadlag (right continuous with left limits) time setting with discrete tick prices, where each price step is one-tenth of a tick (a tick is defined as 0.0001), and suppose we are trying to execute a buy pegged order. Let A_t and B_t be the best ask and bid prices respectively. Then the pegged to best ask and bid prices with non-negative integer tick offsets x_t, denoted as $\widetilde{A}_t(x_t)$ and $\widetilde{B}_t(x_t)$, are defined as

$$\widetilde{A}_t(x_t) := A_t + 0.0001x_t$$
$$\widetilde{B}_t(x_t) := B_t - 0.0001x_t$$

Assuming that the underlying asset is liquid enough such that the volume of the orders are negligible compared to the volume available at the best prices. Then the buy orders are executed at time

$$t^B := \min_{\substack{t \geq 0 \\ \widetilde{B}_{t^-}(x_{t^-}) \geq A_t}} t$$

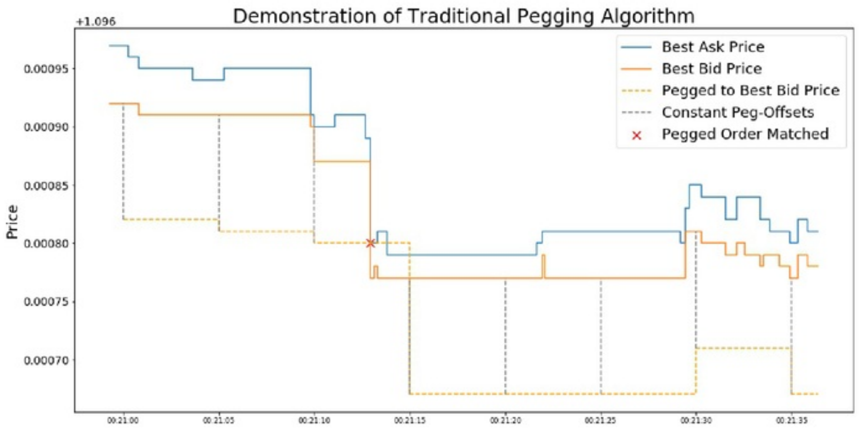
where $t^- := \lim_{s \uparrow t} s$ is the left limit of t.
Similarly, the sell orders are executed at time

$$t^A := \min_{\substack{t \geq 0 \\ \widetilde{A}_{t^-}(x_{t^-}) \leq B_t}} t$$

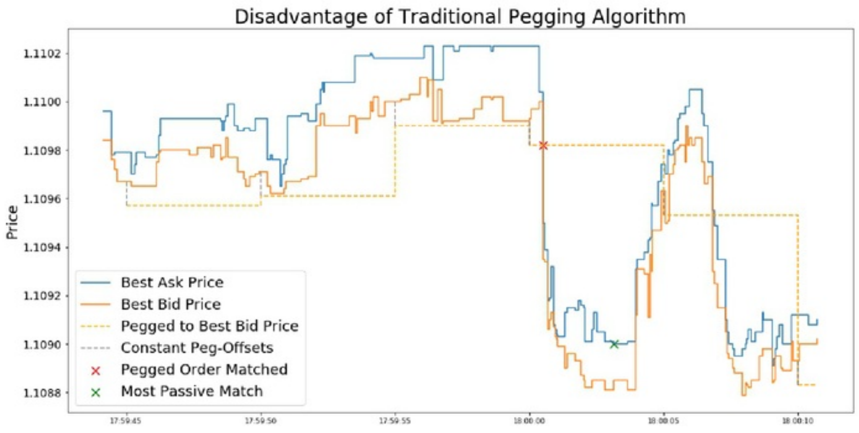
In reality though, it is impractical to peg in the way as described above, as this means that we need to amend our orders continuously. A more practical approach is to use a constant frequency peg, where we peg the prices every w seconds and have the pegged prices to stay constant in the w-seconds windows. Thus our definition of the pegged prices (We will be using this definition from now on, unless stated otherwise) are altered as follows. Let $\widetilde{t} := w\left\lfloor \frac{t}{w} \right\rfloor$, then

$$\widetilde{A}_t(x_t) := A_{\widetilde{t}} + 0.0001x_{\widetilde{t}}$$
$$\widetilde{B}_t(x_t) := B_{\widetilde{t}} - 0.0001x_{\widetilde{t}}$$

The diagram below demonstrates how the constant frequency peg with 5-seconds windows and constant peg-offset of 1 for buy orders work.



Traditional pegging algorithms have a fixed offset, i.e. $x_t = x \in \mathbb{N}_0\ \forall t \geq 0$. A high offset indicates that the trader is passive and believes that the orders can be matched due to periods of high volatility, whereas a low offset indicates aggressiveness, where the trader foresees periods of low volatility. By fixing the offset as constant, we are effectively assuming that the market conditions are not going to change. As a result, the performance of the constant pegging algorithm may suffer if that is not the case. Below is an example demonstrating the disadvantages of having a constant peg-offset.



The pegging algorithm in the diagram above is the same as the one in the first diagram. The pegged order is matched at around 18:00:00, where there is an increase in volatility, which results in a rapid drop in prices. Since the pegging algorithm has a constant peg-offset, the order is executed at a price near 1.1098. However, this order can actually be executed at a much lower price if the peg-offset is higher, with the most optimal execution price located at the green cross. While it is very difficult to predict the exact location of the green cross beforehand, if we can predict the increase in volatility and appropriately alter the peg-offsets, then the orders can be executed more passively.

From above, we have seen how a pegging algorithm with relatively low constant offset can execute an order too quickly in periods of high volatility. The converse is true as well; if the constant peg-offset is set to be too high, then the orders can never (or rarely) be executed in periods of low volatility. All of the above motivate the need for a dynamical pegging algorithm which alters the peg-offsets based on market conditions.

Nevertheless, it is not that straightforward to find the optimal way to alter the peg-offsets. The main obstacle here is that in order to do this, we are required to predict the market conditions throughout the whole trading period, which involves multiple steps ahead predictions. In most cases we are only able to make one-step ahead predictions with decent accuracies. Therefore, we do not expect the algorithms to find the optimal way to alter the peg-offsets. Instead, we aim to make use of the one-step ahead predictions of market conditions in the dynamical pegging algorithm to alter the peg-offsets, until the performance is good enough to be a decent alternative of the traditional pegging algorithm with constant peg-offsets.

2 Methodology

The problem is split into two parts. The first part is to build the main algorithm to dynamically optimize the peg-offsets, which we deem that a reinforcement learning approach would be the most suitable, since the dynamical pegging algorithm can be easily formulated as a Markov Decision Process (See Section 5). Future volatility trends will be used as the main feature in the main algorithm. Since the future volatility trends are unknown in real time, volatility forecasting models are required for real time implementation of the main algorithm. This forms the second part of the problem.

Note that for simplicity, the algorithms are built and tested on buy orders only throughout the thesis. For a buy order, the volatility trends of the best ask prices are used, since matching of buy orders rely on the fluctuations of the best ask prices. The algorithm for sell orders is analogous and uses the best bid volatility trends.

2.1 General Assumptions and Settings

The following assumptions are made in an attempt to simplify the problems.

2.1.1 Length of Trading Time

The pegging algorithm is usually used within a higher level execution algorithm like the TWAP (Time Weighted Average Price), which sends child orders of equal volumes out at a fixed time interval. Typically these child orders should be filled within 30 minutes to an hour. To satisfy this requirement and to maximize the number of trading episodes used for training, we set our pegging algorithm to run no longer than 30 minutes.

2.1.2 Re-peg Frequency

Ideally, we want to re-peg our orders as frequently as possible to follow the market. However, as explained in Section 1.3, this is impractical. Instead, we use a 5-seconds frequency, which is the highest frequency possible for re-pegging to work. This means that the orders are re-pegged every 5 seconds and the peg-offsets are valid for the next 5 seconds.

2.1.3 Viable Peg-Offsets

We limit our peg-offset range to 0 to 5 ticks, since it is very unlikely for orders to be matched at offsets higher than 5 ticks. While negative offsets are valid, orders can easily be matched at offset 0, so there is no point in being any more aggressive than that.

The peg-offsets can be altered to any offsets as orders are being re-pegged.

2.1.4 Other Assumptions

- Only one order can be executed in each 5-seconds window, and is fully matched and executed instantly once the execution criteria is met
- The assumption above means that we have 360 orders at maximum in each run of the algorithm

2.2 Data

The algorithm will be built and tested on FX, since it is the most liquid out of all asset classes, which means that more data is available and the assumption of orders being fully matched is more realistic. In this thesis we are going to build and test the algorithms on EUR/USD, GBP/USD and NZD/USD. We will be using level I broker's data that contains ultra high frequency (milliseconds) best bid and ask rates for all currency pairs. Nevertheless, the algorithm can easily be extended to other asset classes in the future.

2.3 Volatility

Since we are working in a short term environment, we are going to use realized volatility as our risk measure for better indication of short term market conditions [14]. Suppose we want to calculate the realized volatility of a currency with exchange rate A_t. Furthermore assume that the rates are observed at time $0 = t_0 < t_1 < t_2 < \dots$. Then the realized volatility of the asset in the time range $[s, t]$ is defined as

$$\mathrm{RV}_{s,t} := \sqrt{\sum_{t_i \in [s,t]} \left(\log \left(\frac{A_{t_i}}{A_{t_{i-1}}} \right) \right)^2}$$

In our setting, the realized volatilities are typically very small numbers. Therefore, we are going to multiply our realized volatilities by 1000 to minimize numerical errors in the algorithms.

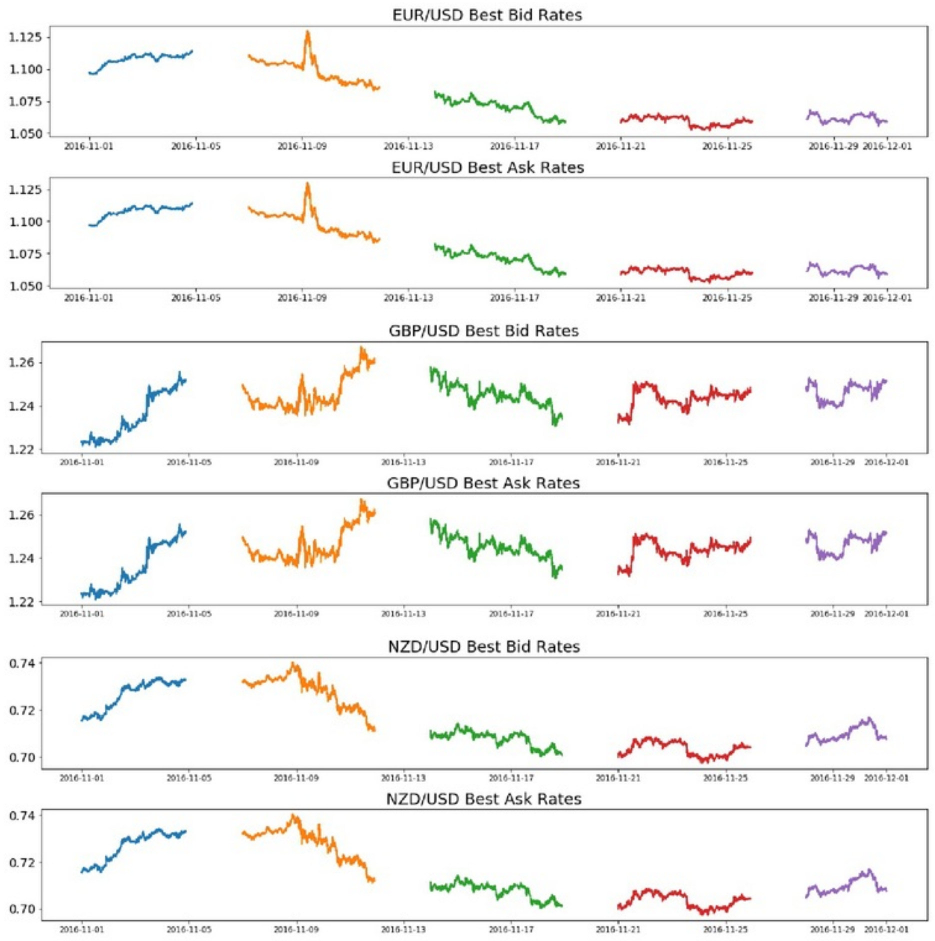
2.4 Performance Metrics

The top priority of a pegging algorithm is to successfully execute an order. Therefore, we will be using the proportion of successful executions among trade orders to determine whether an algorithm is viable. For example, we can set a minimum successful execution rate of 99% to be the benchmark of a viable algorithm.

Furthermore, we also want the orders to be executed as passively as possible (that is, at an as low as possible peg offset). Therefore, among the viable algorithms, we calculate the average peg offsets of successful executions, and also investigate the distributions of the peg-offsets in order to determine the best algorithm.

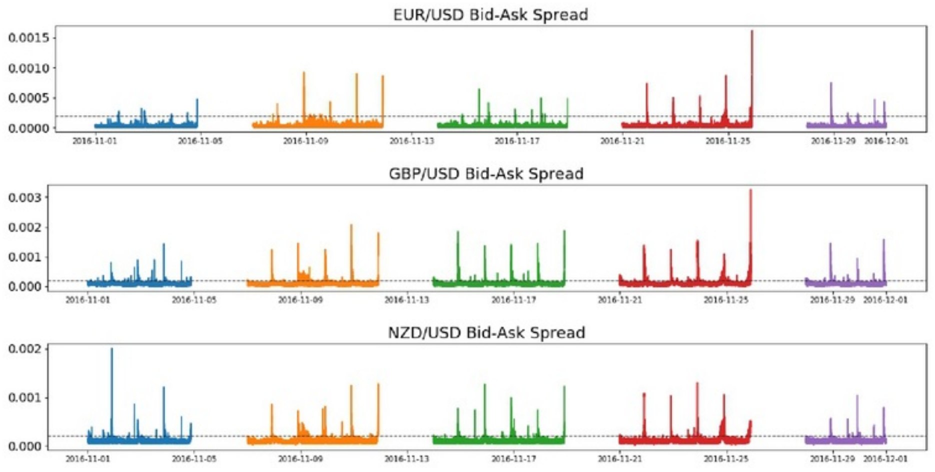
3 Exploratory Data Analysis

Recall that we are using EUR/USD, GBP/USD and NZD/USD best bid and ask rates as our raw data. We are going to build and tune the algorithms based on the rates in November 2016, which consists of approximately 6.2 million pairs of best bid-ask rates. Let us first look at the raw data.



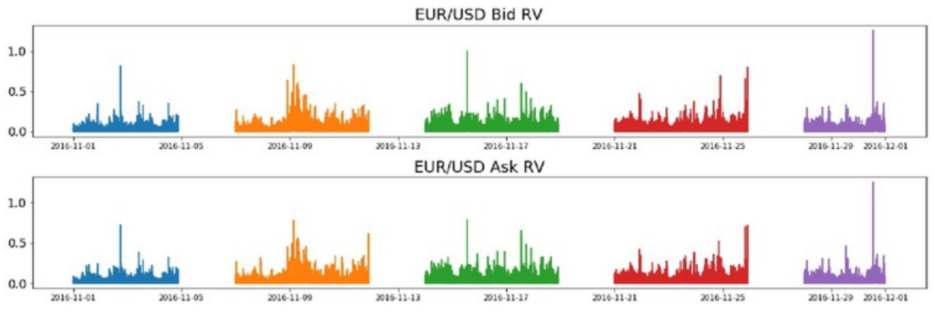
As we can see, the data consists of gaps in about every 4 days. This is due to the fact that the foreign exchange market operating from 22:00 GMT Sunday until 22:00 GMT Friday for each week, so there will always be gaps in between weeks. The data seems fine apart from the sudden peak on the 9th of November for the EUR/USD rates. This is unlikely to be an error though since it corresponds to the Trump election date.

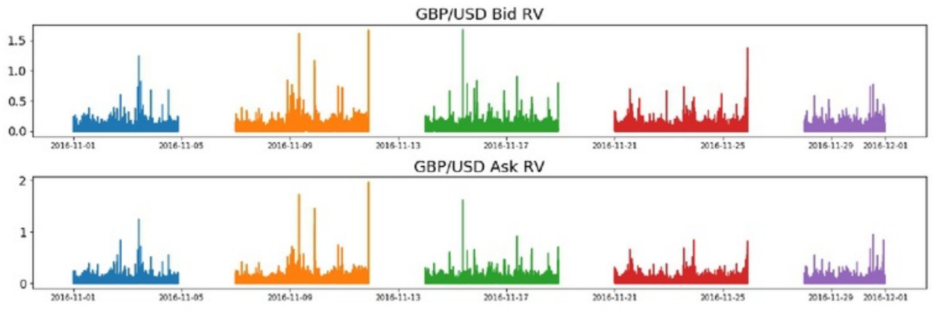
Notice that it is hard to distinguish the bid and ask rates in the plots above because they are so close to each other. We should plot the bid-ask spreads for this purpose.

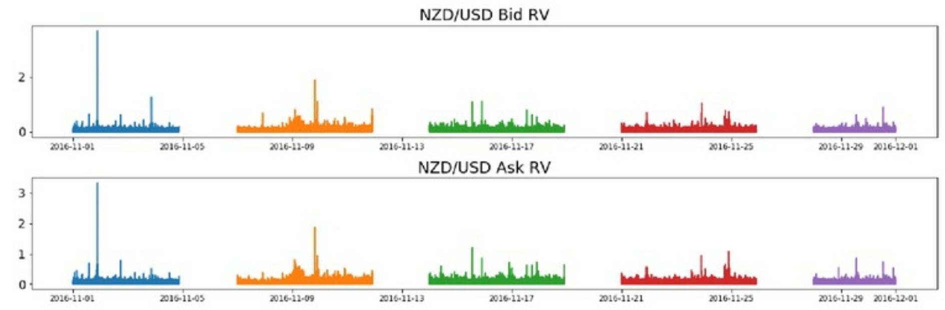


We can see that the spreads are quite small since most of the spreads are within 2 ticks (the dashed lines). This indicates the high liquidity nature of the currency pairs and that the data is suitable for our problem as stated in 2.2. Additionally, there are occasional spread peaks in all currency pairs and they appear at similar places. This combined with similar general trends across raw exchange rates suggests that there may be some correlations in between these currency pairs.

We now investigate the correlations in between the currency pairs. To do this, we will investigate the 5-seconds realized volatilities of the exchange rates their corresponding cross-correlogram.



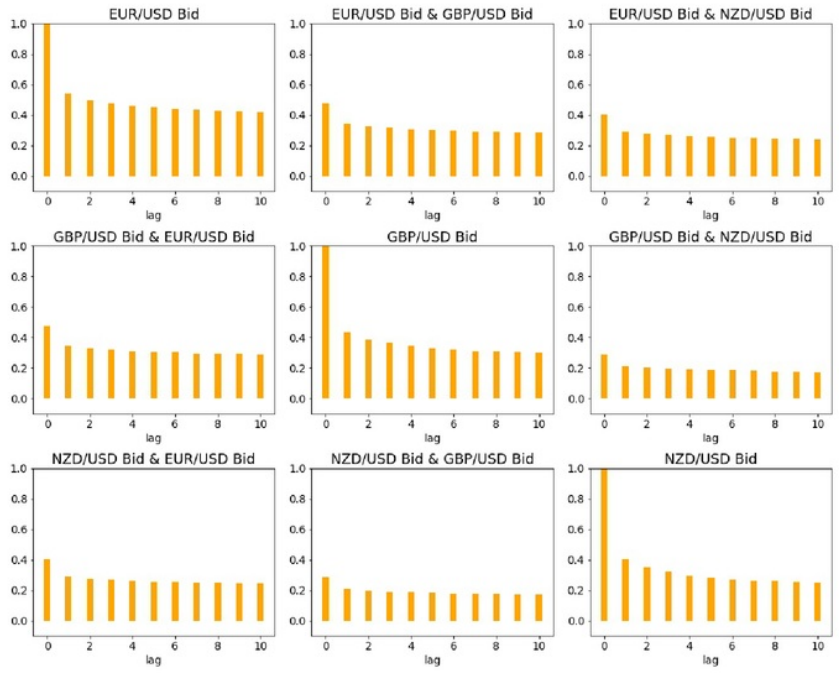


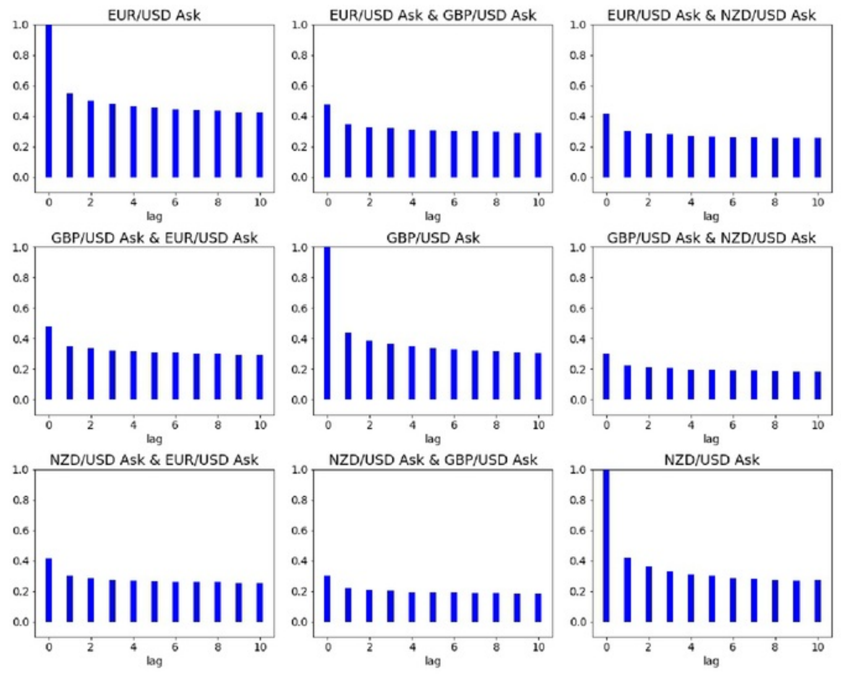


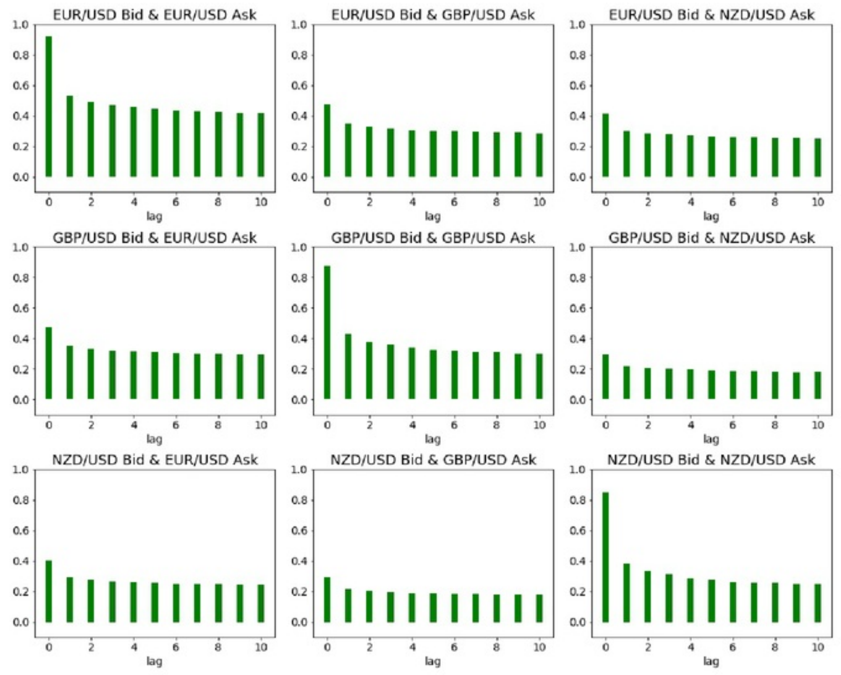
Judging from the volatility graphs from above, we can see that the raw volatilities are very noisy, which means that it is very difficult to make accurate forecasts. To deal with this, one may try and denoise the data with standard methods such as wavelet transformations. However, due to the short term nature of our problem, this is not suitable for our case. This is because the algorithm works in an ultra high frequency setting such that noises form a large part of the data. Removing the noises means that the data is altered completely.

Instead, we will try and predict the volatility trends in between 5-seconds windows, forecasting whether the volatility is going to decrease, remain unchanged or increase. Notice that we have included the unchanged category. This is actually possible and happen frequently when there are no price changes in those windows, leading to zero realized volatilities.

The corresponding Bid, Ask, and Bid-Ask realized volatility cross-correlograms are as follows.




EUR/USD Ask
EUR/USD Ask & GBP/USD Ask
EUR/USD Ask & NZD/USD Ask
GBP/USD Ask & EUR/USD Ask
GBP/USD Ask
GBP/USD Ask & NZD/USD Ask
NZD/USD Ask & EUR/USD Ask
NZD/USD Ask & GBP/USD Ask
NZD/USD Ask
lag


EUR/USD Bid & EUR/USD Ask
EUR/USD Bid & GBP/USD Ask
EUR/USD Bid & NZD/USD Ask
GBP/USD Bid & EUR/USD Ask
GBP/USD Bid & GBP/USD Ask
GBP/USD Bid & NZD/USD Ask
NZD/USD Bid & EUR/USD Ask
NZD/USD Bid & GBP/USD Ask
NZD/USD Bid & NZD/USD Ask
lag

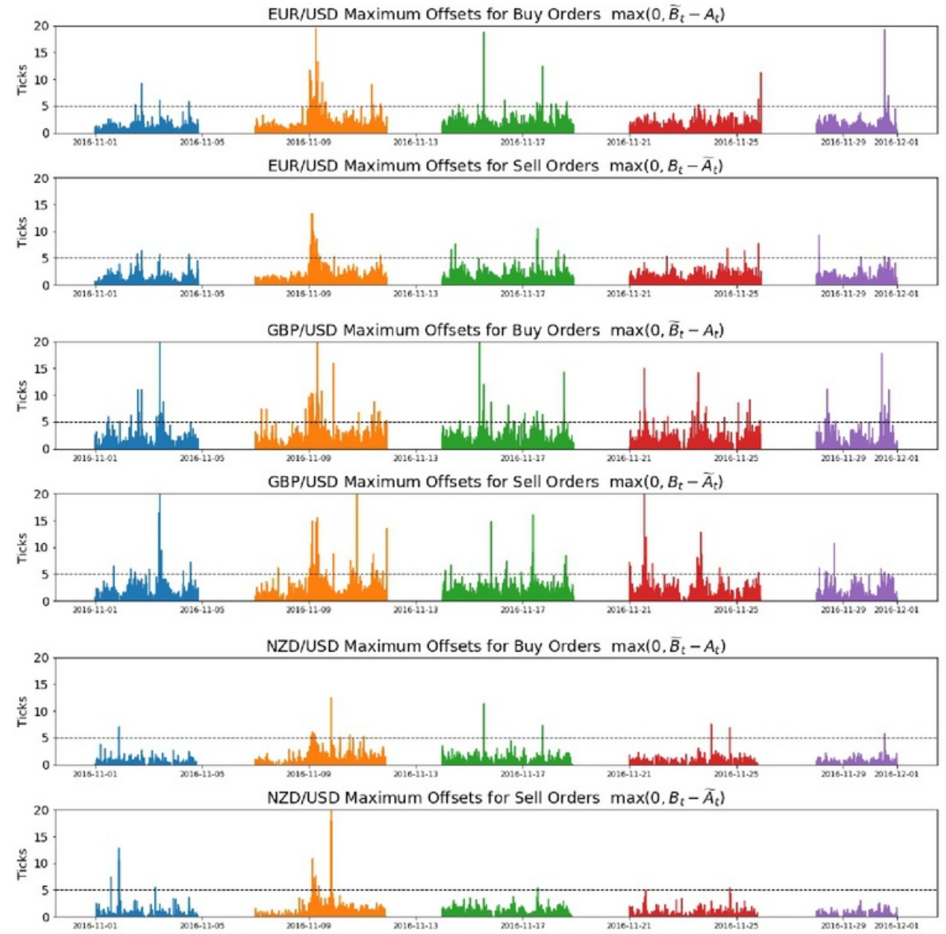
Note that we define the cross-correlation in between two univariate sequences $(x_t)_{t=1}^T$ and $(y_t)_{t=1}^T$ with lag τ as

$$r_{xy}(\tau) := \frac{\sum_{t=\tau+1}^{T}(x_t - \bar{x})(y_{t-\tau} - \bar{y})}{\sum_{t=1}^{T}(x_t - \bar{x})(y_t - \bar{y})}$$

where $\bar{x}$ and $\bar{y}$ are the means of the sequences $(x_t)_{t=1}^T$ and $(y_t)_{t=1}^T$ respectively.

Judging from above, we observe that the auto and cross-correlations of realized volatilities are persistent, which suggests the existence of volatility clusters. In particular, the relatively high correlations in between same currency pairs supports the idea of using historical volatilities of the same currency pairs for forecasting.

We now investigate the upper bound of peg-offsets to be used in our algorithm. Below are the plots showing the maximum offsets which the pegged orders can be executed at.



Observe that apart from occasional peaks, most orders can only be executed within offset 5. This justifies the choice of a maximum peg-offset of 5.

4 Feature Selections for Volatility Trend Predictions

From the previous section we saw that the 5-seconds realized volatilities are serially correlated. Since our problem involves predicting the ask volatility trends, it makes sense to use the past ask volatilities as features. It remains to decide on how far back we should look at to include as features. The process of choosing which features to be included in any supervised learning models is known as feature selection.

There are three main principals in feature selection, namely, the filter method, the wrapper method and the embedded method. The filter method selects features directly before training any models by considering the relationship in between the features and the target variables. The most related features are then chosen and used in model training. The wrapper method requires us to first decide on the models to be trained on and then chooses the best combination of the features in terms of model performances. This is done during model training and hyperparameters tuning. The embedded method essentially means that we choose models that incorporate their own feature selection process. Examples include the LASSO regression and decision trees methods.

In this thesis, we will use the filter method approach as it is less computationally intensive. In particular, we will be selecting features based on their mutual information.

4.1 Mutual Information

Suppose we treat the features and the targets of each observation as samples of random variables. Then the mutual information in between targets and each feature is a measure of the mutual dependence in between them. Intuitively, the mutual information in between two random variables quantifies the amount of information obtainable for one of the random variables through observing the other random variable.

More formally, let X and Y denote the random variables of a feature and the target respectively, with corresponding marginal probability density function P_X and P_Y respectively. Furthermore, let the joint probability density function of (X, Y) be $P_{X,Y}$. Then the mutual information I in between X and Y is defined as

$$\begin{aligned}
I(X;Y) &:= \mathbb{E}_{P_{X,Y}}\left[\log\left(\frac{P_{X,Y}(X,Y)}{P_X(X)P_Y(Y)}\right)\right] \\
&= \mathbb{E}_{P_{X,Y}}\left[-\log P_X(X) - \log P_Y(Y) + \log P_{X,Y}(X,Y)\right] \\
&= -\mathbb{E}_{P_{X,Y}}\left[\log P_X(X)\right] - \mathbb{E}_{P_{X,Y}}\left[\log P_Y(Y)\right] + \mathbb{E}_{P_{X,Y}}\left[\log P_{X,Y}(X,Y)\right] \\
&= -\mathbb{E}_{P_X}\left[\log P_X(X)\right] - \mathbb{E}_{P_Y}\left[\log P_Y(Y)\right] + \mathbb{E}_{P_{X,Y}}\left[\log P_{X,Y}(X,Y)\right] \\
&=: H(X) + H(Y) - H(X,Y)
\end{aligned}$$

where $H(X), H(Y)$ are the marginal entropies of X and Y respectively, and that $H(X,Y)$ is the joint entropy of X and Y.

For our problem, we use the non-parametric, $k-$nearest neighbour approach to estimate the entropies. Let Z be some p-dimensional random variables with samples Z_i, $i = 1 \to n$. Then the entropy estimator $\hat{H}$ is

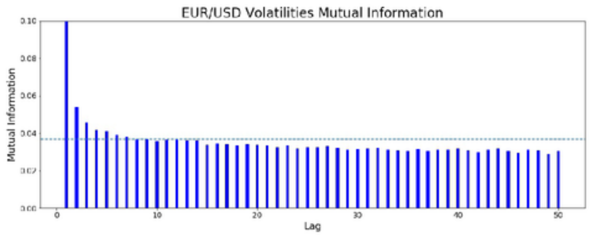
$$\hat{H}(Z) = -\frac{1}{n}\sum_{i=1}^{n}\log \hat{f}(Z_i)$$

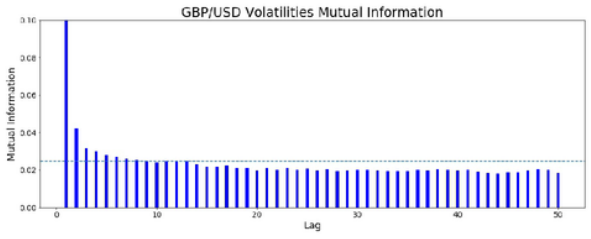
with

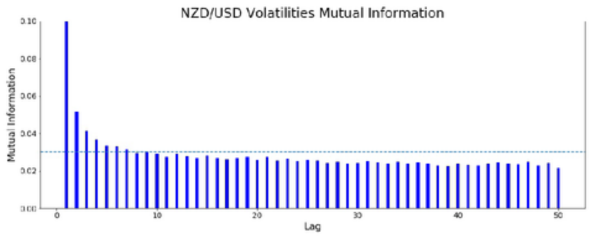
$$\hat{f}(Z_i) = \frac{k\Gamma(1+\frac{p}{2})}{n\pi^{\frac{p}{2}}R^p_{i,k,n}}$$

where k is some arbitrary number such that $0 < k \leq n$, Γ is the gamma function and $R_{i,k}$ is the euclidean distance in between Z_i and its kth nearest neighbour within the samples $\{Z_j, 1 \leq j \leq n, j \neq i\}$. The derivation of the above is omitted. For more details see [12].

Below are the graphs showing the mutual information in between one-step ahead volatility trends and each of the past volatilities, estimated using $k = 5$.







We can see that for every currencies, the mutual information in between the one-step ahead volatility trends and past volatilities decreases as we look further back. Therefore there should be a diminishing increase in performance when using more past volatilities. To limit the number of features, we have decided to only include up to the past 9 windows. The corresponding mutual information in between the one-step ahead volatility trends and the volatility 9 windows back for each currency pair is shown as the light blue dashed line in each graph.

5 Machine Learning Algorithms

5.1 Main Reinforcement-Learning Algorithm

5.1.1 Markov Decision Process and Reinforcement Learning

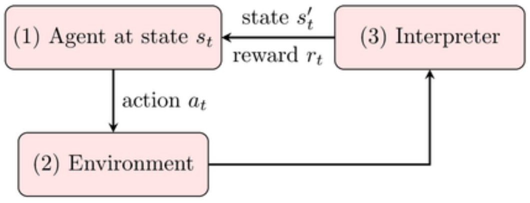
A standard reinforcement learning system can be modelled as a finite Markov Decision Process [13], which consists of

- The set of states S
- The set of actions A
- The probability of transition $P_a(s, s')$ from state $s \in S$ to $s' \in S$ under action $a \in A$
- The immediate reward function $R_a(s, s')$ after transition from $s \in S$ to $s' \in S$ under action $a \in A$

In such a system, there is an agent which interacts with an environment within some time period known as an episode. An episode consists of time-steps. At each time-step t, the agent is responsible for making actions that alters the environment. An interpreter then observes the environment and determine which state the agent is in and sends an immediate reward

$$r_t = R_{a_t}(s_t, s'_t)$$

Below is a diagram demonstrating the standard reinforcement learning paradigm at time-step t. Starting from the state s_t, the agent performs an action a_t, receives reward r_t just before arriving at time-step $t+1$ and transits to state s'_t. The agent then arrives at time-step $t+1$ starting from state $s_{t+1} = s'_t$ and continue until the episode ends at a termination state.



The policy in reinforcement learning is a function $\pi : A \times S \mapsto [0, 1]$ defined as

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

The purpose of reinforcement learning is for the agent to learn the optimal policy π^* which defines the optimal actions from any state $s \in S$. By optimal we mean that at each time-step t, the policy maximizes the expected cumulative reward obtainable in the future, given the present state $s_t = s$. that is, to find π^* that maximizes the state-value function $V_\pi : S \mapsto \mathbb{R}$, defined as

$$V_\pi(s) := \mathbb{E}_\pi \left[R \mid s_t = s\right] := \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s\right]$$

where $\gamma \in [0,1]$ is the discount rate, which determines the trade-off in between immediate and future rewards.

For our case, the agent is the dynamical pegging algorithm; the state variables include the time elapsed, the current peg-offset, the volatility trend prediction, and an execution indicator; the actions are the change in peg-offsets; the rewards can be any non-decreasing functions of the peg-offsets for successful executions and some penalization for non-execution states.

5.1.2 Q-Learning

There are many different types of reinforcement learning algorithm. Among those, we have decided to use the standard Q-learning as our reinforcement learning algorithm, since it is easy to implement and has a low time cost, which is essential under a high frequency setting in real time. Another advantage of using Q-learning is that it is model-free, meaning that it does not require us to make any assumptions in the transition probabilities of states in the algorithm.

The Q-learning algorithm attempts to maximize the expected cumulative reward by building the function $Q_{\pi*} : S \times A \mapsto \mathbb{R}$, defined as

$$\begin{aligned} Q_{\pi*}(s,a) &:= \max_\pi \left(Q_\pi(s,a)\right) \\ &:= \max_\pi \left(\mathbb{E}_\pi \left[R \mid s_t = s, a_t = a\right]\right) \\ &:= \max_\pi \left(\mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a\right]\right) \end{aligned}$$

where $Q_\pi : S \times A \mapsto \mathbb{R}$ is regarded as the action-value function. Note that Q_π is similar to V above but with actions as additional input. The output of $Q_{\pi*}$ determines the quality of an action $a \in A$ from any state $s \in S$, the higher the quality the better the action is. In typical settings like ours, the action and state sets are finite, which means that the action value functions can be stored as multidimensional arrays, where we denote as the Q-Table.

The optimized action-value function is learned using value iteration. Starting from an initial action value function Q, we update Q at each time-step t at each episode until it converges to Q_{π^*} (or until achieving satisfactory performances). The update rule where s_t is a non termination state is as follows.

$$Q(s_t, a_t) \leftarrow (1-\alpha_t)Q(s_t,a_t) + \alpha_t \left[r_t + \gamma \max_a Q(s_{t+1}, a)\right]$$

where α_t is the learning rate of the algorithm at each time-step t. In fully deterministic environment, a constant learning rate of $\alpha_t = 1$ is optimal, yet this is not the case for us. For a stochastic

environment, a decreasing learning rate under some technical conditions are required for convergence of the algorithm [13]. We will be using a learning rate of $\alpha_t = \frac{1}{t+1}$ for a convergence rate that has an exponential dependence on $\frac{1}{1-\gamma}$ [6].
If s_t is a termination state then for well defined immediate reward r_t, we set $s'_t = s_t$ and that

$$Q(s_t, a_t) \leftarrow r_t$$

Throughout the learning episodes, the actions a_t is chosen based on the output of Q with input s_t and all possible actions in the action space A. There are many different ways in literature to choose actions. Naturally, one may want to choose the action with the highest corresponding Q-value from state s_t. An obvious disadvantage of such approach is that we would not have explored other possible states that may give better future cumulative rewards. Ideally, we want to have a good balance in exploration and exploitation in our action choosing algorithm. One way to do this is to use the ϵ-greedy algorithm, which chooses the action with the highest Q-value from the given state s_t with probability $1 - \epsilon \in [0, 1]$, and with probability $\epsilon \in [0, 1]$ choose the remaining actions with equal chance. A small value of ϵ encourages exploitation, whereas a large value of ϵ encourages exploration. The pseudo-code of the ϵ-greedy algorithm is shown below in details.

Algorithm 1: ϵ-greedy

```
1  Generate $U \sim \text{Uniform}(0,1)$
2  Set $a^* := \underset{a}{\text{argmax}}\, Q(s_t, a)$
3  Set $A^* := A \setminus \{a^*\}$
4  Set $(a^{(j)})_{j=1}^{|A^*|}$ to be an arbitrary sequence containing all elements of $A^*$
5  if $U \leq 1 - \epsilon$ then
6      Set $a_t := a^*$
7  else
8      Initialize $i \leftarrow 1$
9      while $U \geq 1 - \epsilon\left(1 - \frac{i}{|A^*|}\right)$ do
10         Update $i \leftarrow i + 1$
11     end
12     Set $a_t := a^{(i)}$
13 end
```

In a reinforcement learning paradigm, there is no train-test split of data. Instead, the agent will learn the action value function as we provide data to it in real time. However, this also means that the algorithm may not be working well at the start. Therefore, we are going to pre-train the agent using historical data before implementing and testing it in real time. In the pre-training stage, we will be setting $\epsilon = 0.6$ to encourage exploration, whereas in real time testing, we will be setting $\epsilon = 0.05$ to encourage exploitation to maximize algorithm performance.

Below is the pseudo-code of the standard Q-Learning algorithm in details, where E denotes the total number of episodes the agent is trained, and T is the maximum number of time-steps within an episode.

Algorithm 2: Q-Learning for Finite State and Action Spaces

```
1  Initialize Q-Array and state $s_0$
2  for $episode = 1 \to E$ do
3      for $t = 0 \to T$ do
4          if $s_t$ is a termination state then
5              Set $Q(s_t, a_t) := r_t$
6              Go to next episode
7          else
8              Choose action $a_t$ from state $s_t$ using $\epsilon$-greedy algorithm
9              Observe the transitioned state $s'_t$ from $s_t$ due to action $a_t$
10             Collect reward $r_t = R_{a_t}(s_t, s'_t)$
11             Set $s_{t+1} := s'_t$
12             Set $Q(s_t, a_t) := (1 - \alpha_t)Q(s_t, a_t) + \alpha_t \left[r_t + \gamma \max_a Q(s_{t+1}, a)\right]$
13         end
14     end
15 end
```

5.1.3 States, Environments and Actions

For our case, all information related to our problem can be obtained from the elapsed time, the current peg-offset and the best ask and bid prices, hence they form the environment of our Q-learning algorithm. It is obvious that the current peg-offset must be included in the state as it is the core of the problem. To determine other essential states that need to be included, note that an episode ends when an order is matched, or when 30 minutes has elapsed, whichever comes first. To determine whether a state is terminal, we must store the time elapsed and to determine whether an order is matched. Therefore, we include a time variable and an execution indicator in our states. Furthermore, we need to incorporate the market conditions in our states for dynamical pegging. This means that we will be adding a volatility trend classifier (increases, remains constant and decreases) variable to our states. We will discuss more about the volatility variable in section 5.2.

Recall from Section 2.1 that we have stated that the orders are repegged every 5 seconds. This means that each time-step represents 5 seconds, and we have 360 time-steps in every episode. Additionally, the peg-offsets ranges from 0 to 5 ticks. This suggests that our state s has dimension $360 \times 6 \times 2 \times 3$. Combined with 6 possible actions of altering to 0 to 5 ticks, our Q-Table has dimension $360 \times 6 \times 2 \times 3 \times 6$.

5.1.4 Reward Function and Discount Factor

The most essential part of a reinforcement learning algorithm is the reward function as it directly determines how the algorithm behaves. It is tricky to choose a function that behaves and converges to what one would like, because the number of choices are potentially unlimited, and there are no definite best choices. This becomes even more difficult when the complexity of the problem increases and the metric to assess the algorithm performance becomes non trivial. Therefore, it may be impossible to optimize the reward function in our Q-learning system. Instead, we limit our choices to several standard functions and find combinations of coefficients that result in good performances. Note that even this simpler reward function tuning task requires a combination of having a good understanding of the problem, knowledge in the related domains, and lots of trial and errors.
The objectives of our algorithm is prioritized as follows.

1. Execute the orders successfully
2. Execute the orders as passively as possible, that is, the higher the peg-offset the better
3. In the event that the same order can be executed successfully at the same offset but different times, favour the earlier one

The first objective is achievable by penalizing the agent if the episode ends with the order not executed. This means that we provide the same least reward to all termination states with time variable corresponding to 30 minutes and execution indicator variable corresponding to non-executions.

The second objective is achievable by having a reward function that is increasing in the peg-offset.

The third objective is achievable in two ways, either by having a discount rate of $\gamma < 1$ or giving a small immediate penalization for all non-termination states. The former approach is less preferred for the following reasons. Firstly, it may not work as intended if the output of the reward function can potentially be negative. Secondly, since the effect of discount rates are multiplicative, it may dampen the future rewards by too much if we have a lot of time-steps, which is the case of ours.

Combining the reasoning above, we propose a reward function $R_a : S \times S$ of the following form

$$R_a(s, s') = \begin{cases} p & \text{If } s' \text{ is terminal due to time running out in an episode} \\ f(x(s')) & \text{If } s' \text{ is terminal due to execution} \\ c & \text{Otherwise} \end{cases}$$

where $p < c \leq 0$ are the penalization for non-executions, $f : \{0, 1, 2, 3, 4, 5\} \mapsto \mathbb{R}^+$ is a non-negative, increasing function in peg-offsets, $x : S \mapsto \{0, 1, 2, 3, 4, 5\}$ is a function to output peg-offsets from states s'.

5.2 Volatility Trend Forecasting Algorithms

Recall from Section 2.3 that we are using realized volatility as our volatility measure. Since we cannot de-noise our data, the results of the volatility predictions are quite poor, with an average R^2 of below 0.5 for any of our models, thus is not practical to use as a proxy for the future true volatility. Consequently, we have decided to alter our approach to predict whether the volatility is going to increase, remains constant or decrease in the next 5-seconds windows. This changes the nature of our problem from a regression problem to a classification one.

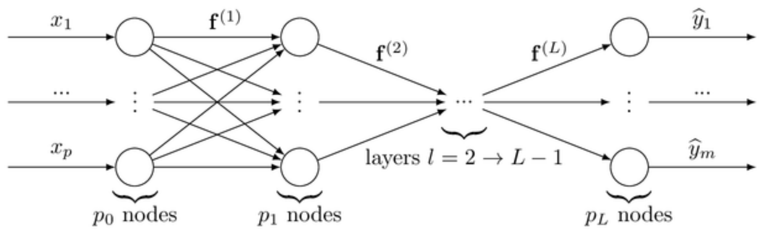
For a binary classification problem, one may consider using a logistic regression approach. However, we have 3 classes here instead of 2, so we cannot use the logistic regression. Instead, we will be using the softmax regression and its extensions, which are generalizations of the logistic regression and are essentially artificial neural networks with different architectures.

5.2.1 Single and Multilayer Perceptron

The multilayer perceptron is a feed-forward artificial neural network which is fully connected and contains an input layer, an output layer, and an arbitrary number of hidden layers. More formally, let $\mathbf{x} \in \mathbb{R}^p$ be an observation of a data matrix with p features (covariates), $\mathbf{y} \in \mathbb{R}^m$ be the corresponding target of $\mathbf{x}$, and $\widehat{\mathbf{y}} \in \mathbb{R}^m$ be the multilayer perceptron predicted target. Then the multilayer perceptron with $L-1$ hidden layers is a function that maps $\mathbf{x}$ to $\widehat{\mathbf{y}}$ in the following way.

$$\begin{aligned} \mathbf{o}^{(0)} &= \mathbf{x} \\ \left.\begin{aligned} \mathbf{a}^{(l)} &= W^{(l)}\mathbf{o}^{(l-1)} + \mathbf{b}^{(l)} \\ \mathbf{o}^{(l)} &= \mathbf{f}^{(l)}(\mathbf{a}^{(l)}) \end{aligned}\right\} &\text{ for } l = 1 \to L \\ \widehat{\mathbf{y}} &= \mathbf{o}^{(L)} \end{aligned}$$

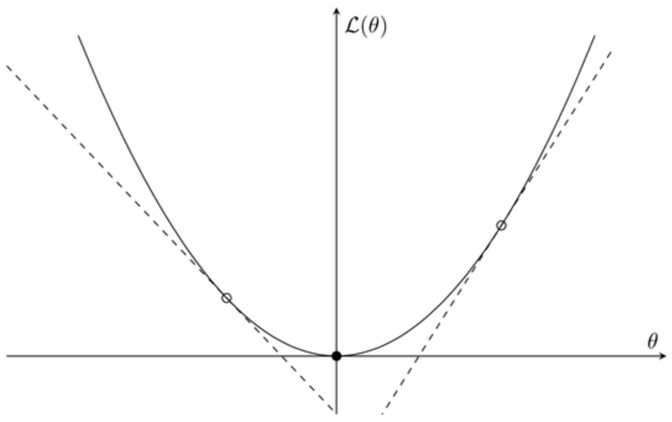
where $\mathbf{a}^{(l)}, \mathbf{o}^{(l)} \in \mathbb{R}^{p_l}$ are the activation and output, $\mathbf{f}^{(l)} : \mathbb{R}^{p_l} \mapsto \mathbb{R}^{p_l}$ is the activation function, $W^{(l)} \in \mathbb{R}^{p_l \times p_{l-1}}, \mathbf{b}^{(l)} \in \mathbb{R}^{p_l}$ are the weight and bias of layer l respectively. Furthermore, we have $p_0 := p$, $p_L := m$, and for $l = 1 \to L-1$, p_l are some arbitrary positive numbers set by the user, denoting the number of nodes in layer l. The process above is known as the foward propagation.



The single layer perceptron is just a special case of the multilayer one with $L = 1$. The parameters of a perceptron are the weights and biases at each layer. The training process of the perceptron involves finding the optimal (or good enough) weights and biases such that the loss in between $\widehat{\mathbf{y}}$ and $\mathbf{y}$ is minimized. To do this, we require a loss function $\mathcal{L}$, as well as an optimization algorithm. Usually, the optimization algorithms are variations of the gradient descent. They involve calculating the gradients of a differentiable loss function with respect to the parameters (weights and biases at each layer) and apply the following update rule until convergence.

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L}$$

where $\boldsymbol{\theta}$ is the parameter to be optimized and $\alpha > 0$ is the learning rate. The idea is demonstrated in the diagram below in the case where $\boldsymbol{\theta}$ is one dimensional.



In the diagram above, the black dot represents the optimum, whereas the unfilled points are parameters that are still undergoing gradient descent. We can see that the points on the left and right are negative and positive, while having negative and positive slopes respectively. As a reseult, they will both approach the optimum if they subtract themselves from positive factors of its gradient. Furthermore, if the loss function is of the form in the diagram above, then θ is going to stay near the optimum when being closed to it, since the magnitude of the gradient decreases as θ approaches the optimum. However, in practice the loss function may not be in this form, so there is no guarantee that the parameters stays near the optimum. For this reason, the learning rate α should be set carefully, since if the learning rate is too high, the parameter may just oscillate around the optimum, whereas if the learning rate is too low, the parameter is going to take a very long time to reach anywhere near the optimum. A better approach is to have a vary learning rate over time.

Another problem with the original gradient descent is that it may not work well in cases which the loss functions have large range of gradient magnitudes. To deal with this, one can set stepsizes that are defined not based on the magnitude of the gradients and update the parameters in the direction based on the sign of the gradients. We now show a popular optimization algorithm named Adam, which combines the ideas above, as well as using the exponential moving averages of gradients in the attempt to smooth out the noises from minibatch training (which will be introduced later on).

Algorithm 3: Adam

```
1  Set learning rate $\alpha > 0$, exponential decay rates $\beta_1, \beta_2 \in [0,1)$, $\epsilon > 0$
2  Set $\boldsymbol{\Theta}$ to be the (finite) parameter space, whose elements are to be optimized
3  Set loss function $\mathcal{L}_{\boldsymbol{\Theta}}$
4  Initialize parameter vectors $\boldsymbol{\theta}_0^{(0)}, \boldsymbol{\theta}_0^{(1)}, \boldsymbol{\theta}_0^{(2)}, ...$
5  Initialize first and second moment vectors $(\mathbf{m}_0^{(0)}, \mathbf{v}_0^{(0)}), (\mathbf{m}_0^{(1)}, \mathbf{v}_0^{(1)}), ...$ to be zero vectors.
6  for $j = 0, 1, 2, ...$ do
7      Initialize timestep $t = 0$
8      while stopping criteria not met do
9          Set $\mathbf{g}_{t+1}^{(j)} := [\nabla_{\boldsymbol{\theta}^{(j)}} \mathcal{L}_{\boldsymbol{\Theta}}]_{\boldsymbol{\theta}^{(j)} = \boldsymbol{\theta}_t^{(j)}}$
10         Set $\mathbf{m}_{t+1}^{(j)} := \beta_1 \mathbf{m}_t^{(j)} + (1 - \beta_1)\mathbf{g}_{t+1}^{(j)}$
11         Set $\widehat{\mathbf{m}}_{t+1}^{(j)} := \frac{1}{1-\beta_1^t} \mathbf{m}_{t+1}^{(j)}$
12         Set $\mathbf{v}_{t+1}^{(j)} := \beta_2 \mathbf{v}_t^{(j)} + (1 - \beta_2)\left(\mathbf{g}_{t+1}^{(j)} \odot \mathbf{g}_{t+1}^{(j)}\right)$
13         Set $\widehat{\mathbf{v}}_{t+1}^{(j)} := \frac{1}{1-\beta_2^t} \mathbf{v}_{t+1}^{(j)}$
14         Set $\boldsymbol{\theta}_{t+1}^{(j)} := \boldsymbol{\theta}_t^{(j)} - \alpha \left(\widehat{\mathbf{m}}_{t+1}^{(j)} \oslash \left({\mathbf{v}_{t+1}^{(j)}}^{\circ\frac{1}{2}} + \epsilon\right)\right)$
15         $t \leftarrow t + 1$
16     end
17 end
18 return $\boldsymbol{\theta}_t$
```

The symbols $\odot$ and $\oslash$ in the algorithm above are the Hadamard product and division respectively (Entrywise product and divisions of two arrays with the same dimensions. The output array also has the same dimensions). Similarly, the superscript symbol $\circ\frac{1}{2}$ denotes the entrywise square root of an array.

An effective way to calculate the gradients is through backpropagation, which iteratively computes the gradients backwards, via chain rule, starting from L then $l = L - 1 \to 1$. For any fully connected MLP, the gradients can be computed as follows

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \left(\nabla_{\mathbf{a}^{(L)}} \mathbf{o}^{(L)}\right)^T \nabla_{\mathbf{a}^{(L)}} \mathcal{L} \\
\boldsymbol{\delta}^{(l)} &= \left(\nabla_{\mathbf{a}^{(l)}} \mathbf{o}^{(l)}\right)^T {W^{(l+1)}}^T \boldsymbol{\delta}^{(l+1)} \\
\nabla_{\mathbf{b}^{(l)}} \mathcal{L} &= \boldsymbol{\delta}^{(l)} \\
\nabla_{W^{(l)}} \mathcal{L} &= {\boldsymbol{\delta}^{(l)}}^T \mathbf{o}^{(l-1)}
\end{aligned}$$

The above relation can proved by letting $\boldsymbol{\delta}^{(l)} = \nabla_{\mathbf{a}^{(l)}}\mathcal{L}$ and applying the multivariate chain rule.

Notice that we have introduced the architecture of the multilayer perceptron via the perspective of one observation per time. For training to work though, we must incorporate the whole data set in the backpropagation and Adam algorithm. This is usually done via the concept of minibatch training. Let $b \in \mathbb{N}$ be the batch size of minibatch training, and let $\widetilde{\mathcal{L}} : \mathbb{R}^m \times \mathbb{R}^m \mapsto [0, \infty)$ be the loss function for a single observation. Furthermore, let $X \in \mathbb{R}^{n\times p}$ be the data matrix that contains $n \geq b$ observations. Assume for simplicity that n is a multiple of b, then the loss function $\mathcal{L} : \mathbb{R}^{b\times m} \times \mathbb{R}^{b\times m} \mapsto [0, \infty)$ in the backpropagation and Adam algorithm above, is defined as

$$\mathcal{L}\left(Y^{(b)}, \widehat{Y}^{(b)}\right) = \frac{1}{b}\sum_{i=1}^{b} \widetilde{\mathcal{L}}\left(\mathbf{y}_i, \widehat{\mathbf{y}}_i(\mathbf{x}_i)\right)$$

where $\mathbf{x}_i$ are some unique rows (observations) of X, with corresponding targets $\mathbf{y}_i$. The whole data matrix X is split into sub data matrices $X^{(b)} \in \mathbb{R}^{b\times p}$ with corresponding target matrices $Y^{(b)} \in \mathbb{R}^{b\times m}$ and predicted target matrices $\widehat{Y}^{(b)}$, which are all fed to the algorithm recursively. A complete session of feeding all sub matrices once is defined as an epoch. A standard way to define the stopping criteria of the Adam algorithm is to set the number of epochs.

We have just briefly introduced the general framework of perceptron training. To apply it on our problem, we need to set the hyperparameters. Some of the hyperparameters, such as the activation function of the output layer and the corresponding loss function, are more important as they are problem specific. On the other hand, those hyperparameters that are not problem specific should be tuned during model training. See Section 6.

In a m-class classification problem, we use a one-hot encoded variable $\mathbf{y} \in \{0, 1\}^m$ as the target variable. By one-hot encoding, we mean that $\mathbf{y}$ has exactly one entry being 1 and 0 elsewhere. The position of 1 in $\mathbf{y}$ corresponds to the class $\mathbf{y}$ is in. We can also interpret the entries of $\mathbf{y}$ as the probability of $\mathbf{y}$ being in the corresponding states. This interpretation motivates us to the softmax activation function. (Note that the softmax regression is basically a single layer perceptron with softmax output activation function).

At any layer l with p_l nodes, the softmax function $\mathbf{s} : \mathbb{R}^{p_l} \mapsto [0, 1]^{p_l}$ is defined as

$$\mathbf{s}(\mathbf{a}^{(l)}) = \frac{\exp(\mathbf{a}^{(l)})}{\mathbf{1}^T \exp(\mathbf{a}^{(l)})}$$

where the exponential function is defined entrywise.

The softmax function is one of the most suitable activation functions for a classification problem. The reason for that is that the output of softmax can be interpreted as probabilities, as the sum of all entries of softmax's output is $\mathbf{1}^T\mathbf{s}(\mathbf{a}^{(l)}) = 1$, and that each entry is in between 0 and 1. By having the softmax function as the final activation function, each entry of the output $\widehat{\mathbf{y}}$ can be interpreted as the probability of being in the corresponding class.

To train any perceptrons with softmax output activation function, it is standard to use the categorical cross-entropy loss function. Let $\widetilde{\mathcal{L}}_{ce} : \{0,1\}^m \times [0,1]^m \mapsto [0,\infty)$ be the categorical cross-entropy loss function for one observation. Then we have

$$\widetilde{\mathcal{L}}_{ce}\left(\mathbf{y}, \widehat{\mathbf{y}}\right) = -\mathbf{y}^T \log \widehat{\mathbf{y}}$$

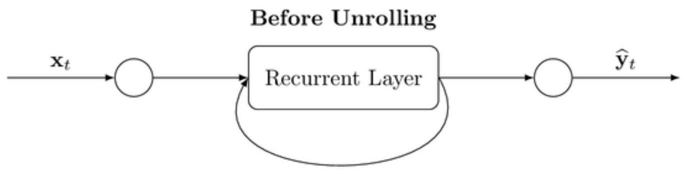
where log is the natural logarithmic function defined entrywise.

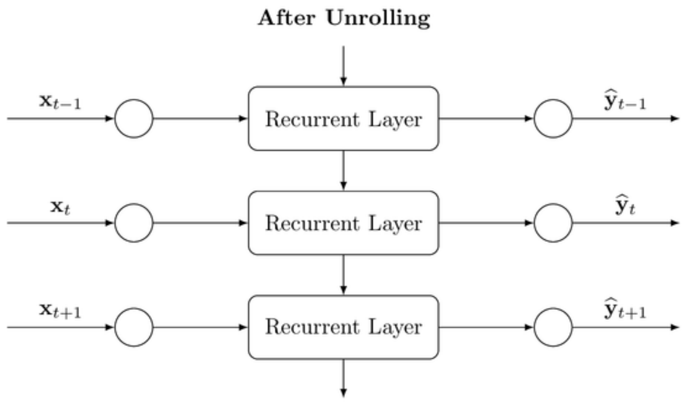
5.2.2 Recurrent Neural Networks

While the multilayer perceptron is a powerful and flexible function approximator, it considers the observations independently and neglects any possible interactions in between observations. In our problem, the data is a time series, which is sequential in nature. Therefore, it may be useful to also consider algorithms that are able to incorporate that.

Recurrent neural network is a class of artificial neural network where some nodes or layers of the network is connected to those of the next timestep, thus forming a directed graph along a temporal sequence. This means that the inputs to the network are processed as sequences and is suitable for our problem.

While there are many types of recurrent neural networks, many of them are similar and follow the same general architectures. One of the most basic recurrent neural network architecture near timestep t is shown below.





The two diagrams above show the exact same architecture, with the second one being more clear about how the connection in between time steps. Note that while it is possible to unroll the architecture introduced above, this is not always the case in general. Nevertheless, it is sufficient for us as we will be focusing on recurrent neural networks of the form above, where there is only one recurrent layer in the networks.

In contrast to the hidden layers of a multilayer perceptron which contains nodes which are fully connected, the recurrent layers of a recurrent neural networks consist of architectures of their own, which uniquely determine the types of recurrent neural networks they belong to. To train any of the recurrent neural networks of the form above, a typical approach is to use the backpropagation through time method, which essentially unrolls the networks and perform backpropagation on the unrolled feed forward network.

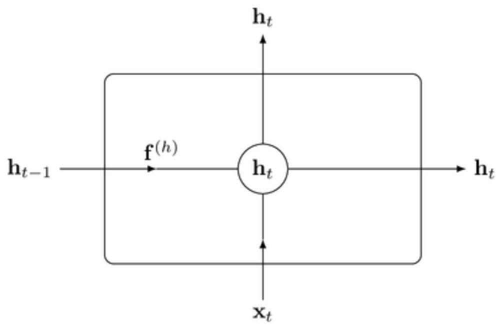
We now introduce 3 types of recurrent neural networks which we will be using in our problem, namely, Elman Network, Long Short-Term Memory and Gated Recurrent Units. They all consists of an input vector $\mathbf{x}_t \in \mathbb{R}^p$, an output vector $\widehat{\mathbf{y}}_t \in \mathbb{R}^m$, and a hidden state vector $\mathbf{h}_t \in \mathbb{R}^h$. The rest of the features of each network is introduced separately in their corresponding sections below.

Elman Network[5]

The Elman Network is one of the most basic recurrent neural networks. It extends the 1-hidden layer perceptron by combining the hidden state vector $\mathbf{h}_{t-1} \in \mathbb{R}^m$ with the input vector $\mathbf{x}_t \in \mathbb{R}^p$ at the next time step, and fed together into the new hidden state vector $\mathbf{h}_t \in \mathbb{R}^h$ for some $h \in \mathbb{N}$. More formally, we have the forward propagation of the network at timestep t as

$$\begin{aligned}
\mathbf{h}_t &= \mathbf{f}^{(h)}\left(W^{(h)}\mathbf{x}_t + U^{(h)}\mathbf{h}_{t-1} + \mathbf{b}^{(h)}\right) \\
\widehat{\mathbf{y}}_t &= \mathbf{f}^{(y)}\left(W^{(y)}\mathbf{h}_t + \mathbf{b}^{(y)}\right)
\end{aligned}$$

where $\mathbf{f}^{(h)}$, $\mathbf{f}^{(y)}$ are the activation functions corresponding to the hidden state and output respectively. $W^{(h)} \in \mathbb{R}^{h\times p}$, $U^{(h)} \in \mathbb{R}^{h\times h}$ and $\mathbf{b}^{(h)} \in \mathbb{R}^h$ are the hidden state parameters. $W^{(y)} \in \mathbb{R}^{m\times h}$, $\mathbf{b}^{(y)} \in \mathbb{R}^m$ are the output layer parameters. Note that we need to initialize $\mathbf{h}_0$ with some vectors.



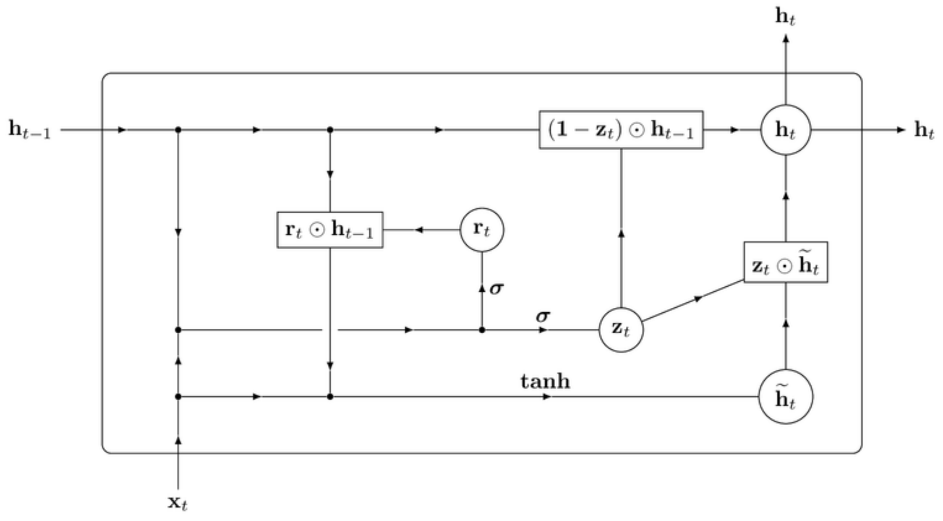
Gated Recurrent Units[8]

The architecture inside a gated recurrent unit is more complex than that of the Elman Network's, involving more nodes inside the recurrent layer. Apart from the hidden state node, a fully gated unit consists of two more nodes known as the update gate $\mathbf{z}_t \in \mathbb{R}^h$ and the reset gate $\mathbf{r}_t \in \mathbb{R}^h$. The purpose of the update gate is to control how much information to be remembered from the hidden state of previous timestep, and the reset gate determines the way to combine the current information with the past. More formally, we have the forward propagation of the network at timestep t as

$$\begin{aligned}
\mathbf{z}_t &= \boldsymbol{\sigma}\left(W^{(z)}\mathbf{x}_t + U^{(z)}\mathbf{h}_{t-1} + \mathbf{b}^{(z)}\right)\\
\mathbf{r}_t &= \boldsymbol{\sigma}\left(W^{(r)}\mathbf{x}_t + U^{(r)}\mathbf{h}_{t-1} + \mathbf{b}^{(r)}\right)\\
\widetilde{\mathbf{h}}_t &= \mathbf{tanh}\left(W^{(h)}\mathbf{x}_t + U^{(h)}\left(\mathbf{r}_t \odot \mathbf{h}_{t-1}\right) + \mathbf{b}^{(h)}\right)\\
\mathbf{h}_t &= (1-\mathbf{z}_t)\odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \widetilde{\mathbf{h}}_t\\
\widehat{\mathbf{y}}_t &= \mathbf{f}^{(y)}\left(W^{(y)}\mathbf{h}_t + \mathbf{b}^{(y)}\right)
\end{aligned}$$

where $W^{(z)}, W^{(r)}, W^{(h)} \in \mathbb{R}^{h\times m}$, $U^{(z)}, U^{(r)}, U^{(h)} \in \mathbb{R}^{h\times h}$ and $\mathbf{b}^{(z)}, \mathbf{b}^{(r)}, \mathbf{b}^{(h)} \in \mathbb{R}^h$ are the update gate, reset gate and hidden state parameters respectively. $\boldsymbol{\sigma} : \mathbb{R}^h \mapsto [0,1]^h$ and $\mathbf{tanh} : \mathbb{R}^h \mapsto [-1,1]^h$ are the multidimensional sigmoid and hyperbolic tangent functions respectively, which are defined as follows. For $i = 1 \to h$, let $\sigma_i : \mathbb{R}^h \mapsto [0,1]$ and $\tanh_i : \mathbb{R}^h \mapsto [-1,1]$ be the functions which output the i^{th} entry of $\boldsymbol{\sigma}$ and $\mathbf{tanh}$ respectively. Then we have

$$\begin{aligned}
\sigma_i(\mathbf{x}) &= \frac{1}{1+e^{-x_i}}\\
\tanh_i(\mathbf{x}) &= \frac{e^{2x_i}-1}{e^{2x_i}+1}
\end{aligned}$$

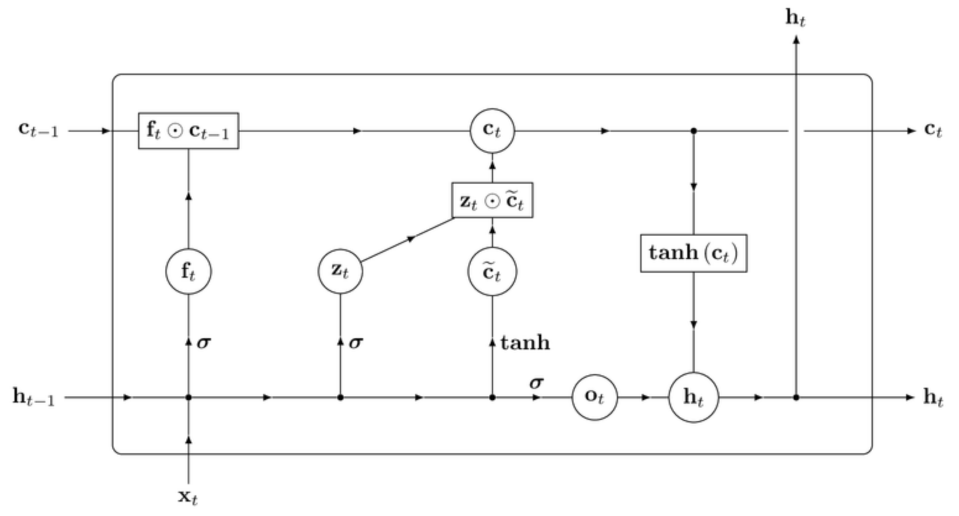

$\mathbf{h}_t$
$\mathbf{h}_{t-1}$
$(\mathbf{1}-\mathbf{z}_t)\odot\mathbf{h}_{t-1}$
$\mathbf{h}_t$
$\mathbf{h}_t$
$\mathbf{r}_t\odot\mathbf{h}_{t-1}$
$\mathbf{r}_t$
$\boldsymbol{\sigma}$
$\mathbf{z}_t\odot\widetilde{\mathbf{h}}_t$
$\boldsymbol{\sigma}$
$\mathbf{z}_t$
tanh
$\widetilde{\mathbf{h}}_t$
$\mathbf{x}_t$

Long Short-Term Memory[10][7]

Similar to the Gated Recurrent Unit, the Long Short Term Memory also consists of gating mechanisms. The main differences are as follows. Firstly, the Long Short Term Memory has an additional cell state $\mathbf{c}_t \in \mathbb{R}^h$. Secondly, the Long Short Term Memory has an output gate $\mathbf{o}_t \in \mathbb{R}^h$, while not having the reset gate $\mathbf{r}_t$. Thirdly, the update gate in Gated Recurrent Unit is now split into an update gate $\mathbf{z}_t$ and a forget gate $\mathbf{f}_t \in \mathbb{R}^h$ in Long Short-Term Memory.

$$
\begin{aligned}
\mathbf{f}_t &= \boldsymbol{\sigma}\left(W^{(f)}\mathbf{x}_t + U^{(f)}\mathbf{h}_{t-1} + \mathbf{b}^{(f)}\right)\\
\mathbf{z}_t &= \boldsymbol{\sigma}\left(W^{(z)}\mathbf{x}_t + U^{(z)}\mathbf{h}_{t-1} + \mathbf{b}^{(z)}\right)\\
\mathbf{o}_t &= \boldsymbol{\sigma}\left(W^{(o)}\mathbf{x}_t + U^{(o)}\mathbf{h}_{t-1} + \mathbf{b}^{(o)}\right)\\
\widetilde{\mathbf{c}}_t &= \mathbf{tanh}\left(W^{(c)}\mathbf{x}_t + U^{(c)}\mathbf{h}_{t-1} + \mathbf{b}^{(c)}\right)\\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{z}_t \odot \widetilde{\mathbf{c}}_t\\
\mathbf{h}_t &= \mathbf{o}_t \odot \mathbf{tanh}\left(\mathbf{c}_t\right)\\
\widehat{\mathbf{y}}_t &= \mathbf{f}^{(y)}\left(W^{(y)}\mathbf{h}_t + \mathbf{b}^{(y)}\right)
\end{aligned}
$$

where $W^{(f)}, W^{(o)}, W^{(c)} \in \mathbb{R}^{h\times m}$, $U^{(f)}, U^{(o)}, U^{(c)} \in \mathbb{R}^{h\times h}$ and $\mathbf{b}^{(f)}, \mathbf{b}^{(o)}, \mathbf{b}^{(c)} \in \mathbb{R}^h$ are the forget gate, output gate and cell state parameters respectively.



5.2.3 Ensemble Learning

Recall that we have proposed several different neural networks for forecasting the volatility trends. A typical approach is to compare their (out of sample) performances and choose the best model to be our volatility prediction algorithm. For our problem though, since the data is very noisy, there is no guarantee that any of the individual models would be performing adequately well as a forecast. This brings us to the idea of ensemble learning, where we combine the output of individual models into a meta-model in order to obtain better predictive performance. This mimics the idea of seeking for multiple opinions from different people to solve a problem.

There are many different approach to ensemble learning. We will be testing the following methods and choose the best to be used in our problem, if it exceeds the predictive power of the best individual model.

Max Voting

This approach works for any classification problem. Suppose we have C individual sub-models for a m-class classification problem. For $i = 1 \to C$, let the target prediction to be $\widehat{\mathbf{y}}^{(i)} \in [0, 1]^m$. Then we define k^{th} element of the target prediction of the Max Voting meta-model as

$$\widehat{y}_k := \begin{cases} 1 & \text{if } k = \min\left(\underset{j\in\{1,2,\dots,m\}}{\operatorname{argmax}} \left(\sum_{i=1}^{C} \left\lfloor \widehat{y}_j^{(i)} + 0.5 \right\rfloor \right) \right), \\ 0 & \text{otherwise.} \end{cases}$$

Essentially, the max voting approach outputs the class which most sub-models predict to be. The reason for us to add 0.5 to the elements of predictions and then take the round down is to convert the predictions to one-hot encoded predictions. Note that the min in the above expression is only needed in cases of ties.

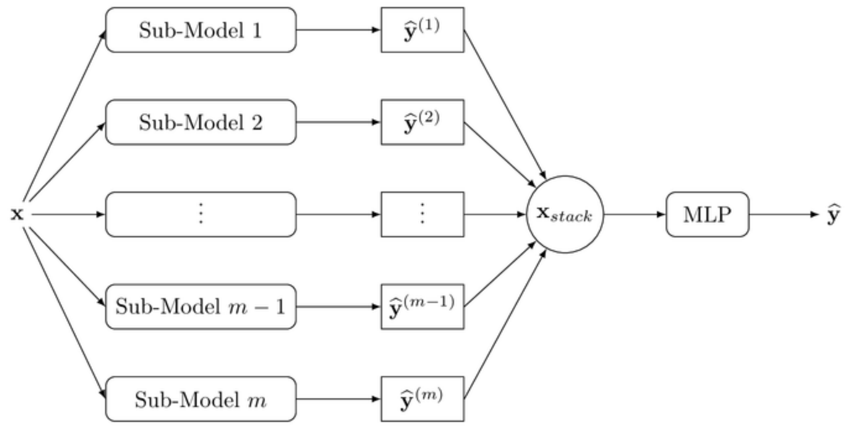
Averaging

This approach works best if the target predictions are continuous. Although we have a classification problem, our prediction vectors are generated by the softmax activation function from any of the models, which means that the prediction vectors are continuous. We can then define target prediction of the Averaging meta-model as

$$\widehat{\mathbf{y}} := \frac{1}{C} \sum_{i=1}^{C} \widehat{\mathbf{y}}^{(i)}$$

Stacking

While the Max Voting and Averaging methods are relatively trivial and easy to implement, they treat all individual sub-models as equals. This is a huge disadvantage as more often than not there will be some sub-models that perform better than the others, thus the weaker sub-models may hinder the performance of meta-models that treat all sub-models equally. One way to deal with this is to assign weights to each sub-model based on their predictive performances. The stacking meta-model is a generalization of such idea by training a multilayer perceptron that maps the target predictions of each sub-model to the meta target predictions. The training process is almost the same as the multilayer perceptron sub-model, but we concatenate the target predictions from sub-models to form a vector $\mathbf{x}_{stack} \in [0,1]^{mC}$ as features instead.


Sub-Model 1
Sub-Model 2
Sub-Model $m-1$
Sub-Model m
$\widehat{\mathbf{y}}^{(1)}$
$\widehat{\mathbf{y}}^{(2)}$
$\widehat{\mathbf{y}}^{(m-1)}$
$\widehat{\mathbf{y}}^{(m)}$
$\mathbf{x}$
$\mathbf{x}_{stack}$
MLP
$\widehat{\mathbf{y}}$

There are no restrictions in the architecture of the stacking meta-model apart from having a softmax output activation function. However, we can choose to either use the continuous softmax sub-model predictions $\widehat{\mathbf{y}}^{(i)}$ or their corresponding one-hot encoded vectors to be the features fed into the stacking meta-model. We will be testing both, as well as a range of architectures, in Section 6.

6 Training and Hyperparameters Tuning

Previously, we have introduced the algorithms to be used in our problem. In this section, we move on to train the models. The training process of the main Q-Learning algorithm is a bit different from that of the volatility algorithms since the latter belongs to the class of supervised learning, which is a different genre to reinforcement learning. For this reason, we are going to split this section into two subsections as before.

6.1 Volatility Algorithms

We first begin with the volatility algorithms because the Q-Learning states depend on the volatility forecasts, which means that we need to train the volatility models first.

Recall that our aim is to build meta-models which consists of various sub-models. We will start with training the sub-models, and then we train the meta-models using the best sub-models.

6.1.1 Hyperparameters Tuning

All of our models are artificial neural networks, which consists of many hyperparameters, and it is important to tune the them to optimize the predictive power of the neural networks. The two most standard ways to do this are grid search and random search. The grid search is a brute force method, which requires us to specify all possible hyperparameter combination and test their corresponding performances one by one. This method does not work well when there are many hyperparameters, due to the curse of dimensionality. In contrast, in a random search not all hyperparameter combinations are tried out. Instead, the user specifies the number of different hyperparameter combinations to be sampled from specified distributions and choose the best combination accordingly. The random search works better than the grid search when the hyperparameter space dimension is high.

Since there are many hyperparameters in an artificial neural network, we will be using the random search approach. The hyperparameters of an artificial neural network include but are not limited to the number of hidden layers, number of hidden nodes, number of epochs, mini-batch size, optimization methods and its corresponding parameters, activation functions etc. To simplify our problem, we are going to do the following.

- Use the ReLU function as our activation function for every layer apart from the output layer
- Fix the optimization method to be the Adam algorithm and tune the Adam parameters
- Use the same number of hidden nodes in each hidden layer and tune both the number of nodes and layers
- Tune the number of epochs and minibatch size

ReLU Function

Let p_l be the number of hidden nodes in hidden layer l. Then as we reach layer l, the ReLU function $\mathbf{ReLU} : \mathbb{R}^{p_l} \mapsto [0, \infty)^{p_l}$ is defined as

$$[\mathbf{ReLU}(\mathbf{x})]_i = \max(x_i, 0)$$

There are many advantages of using the ReLU function as the hidden layer activation functions. For instance, it is computationally efficient to train networks with ReLU activation functions, as only addition, comparison and multiplication is needed to compute the activation output and derivative. Also, due to the nature of the ReLU function, many of the activation outputs become zeros, which makes the model more sparse and apparent.

Adam Parameters

Recall that the Adam algorithm consists of the learning rate $\alpha > 0$, exponential decay rates $\beta_1, \beta_2 \in [0, 1)$ and $\epsilon > 0$. While the author of Adam's paper[4] suggests a default setting of $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$, we will be tuning all of these parameters, sampling them from appropriate distributions. The parameters α and ϵ are usually small and rarely exceed 1. Therefore, we want to sample them uniformly in some sort of logged scale. One way to do it is to use the log-uniform distribution, with its probability density function f_{logU} and cumulative distribution function F_{logU} defined as follows.

$$f_{logU}(x \mid a, b) = \frac{1}{x \log\left(\frac{b}{a}\right)}$$
$$F_{logU}(x \mid a, b) = \frac{\log x - \log a}{\log b - \log a}$$

where $0 < a < b$ and $x \in [a, b]$. The samples can be generated easily by inverting transform sampling.

We will sample α and ϵ from logUniform($a = 10^{-3}, b = 1$) and logUniform($a = 10^{-8}, b = 1$) respectively. For the exponential decay rates, values that are close to 1 are more favoured. As a result, we will be sampling them by simulating $1 - \beta_1$ and $1 - \beta_2$ from logUniform($a = 10^{-3}, b = 1$).

Minibatch Size and Number of Epochs

The minibatch size controls the amount of noise in the optimization process. The smaller the minibatch size the more noise there is. The advantage of having those noises is to avoid getting stuck in the local minima during training [9]. The purpose of minibatch size tuning is to find a good compromise in between injecting enough noise to each gradient update in Adams, while achieving a relative speedy convergence.

The number of epochs controls the distance in between the parameters at the end of the algorithm and the optima for the training set. Using a small number of epochs will lead to underfitting, whereas too many number of epochs may lead to overfitting. The purpose of tuning the number of epochs is to maximise the model's out of sample predictive performances.

We will be setting the grids for minibatch size and number of epochs as $1 \to 10000$ and $100 \to 1000$ respectively.

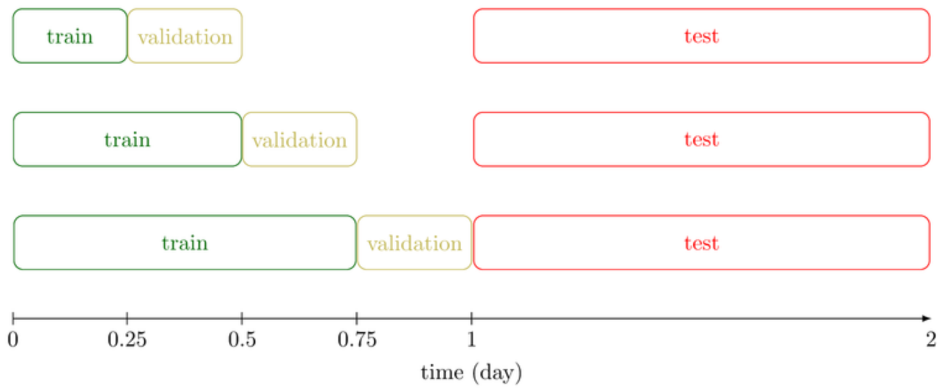
Network Architectures Tuning

The hidden nodes and layers are discrete positive variables. To sample from them, we need to specify the maximum numbers. For our problem, we believe that a maximum number of 5 hidden layers and 100 hidden nodes per layer in multilayer perceptrons is sufficient. Anything more than that may lead to overfitting. As for recurrent neural networks, we will only use one recurrent unit, so we will only be tuning the hidden state vector dimensions.

6.1.2 Train-Test-Validation Split

Recall that in Section 4 we have decided to retrain the volatility model every day, and then the model is used to predict next day's volatility. In order to find the best combination of hyperparameters for each day, we need to compare the corresponding out of sample performances. To do this, we split our data into training, testing and validation sets. The training and validation sets are drawn from current days' data, whereas the test sets are data from the next days. The training sets are responsible for model training for each hyperparameter combination, the validation sets are used to compute estimators of out of sample predictive power of corresponding hyperparameter combination. The test sets are then used to evaluate the performances of the best hyperparameter combinations, which are then compared to other models.

In order to reduce the variances of the out of sample predictive power estimations, we will be using cross-validation techniques. It is very common to use the K-fold cross validation, which means that after holding out the test sets, the remaining data is split into K subsets. Each subset is used once as a validation set, whereas the remaining data are used as training set. As a result, we obtain K out of sample prediction metrics from the validation sets. We then take the average of those and define it to be our cross-validation prediction metrics. However, we cannot use the standard K-fold cross validation in our problem since our data is a time series and is sequential in nature. It is therefore inappropriate to shuffle the data or use data in the future to predict the past. Instead, we use the time series variant of K-fold cross validation. The idea is very similar to that of the standard K-fold cross validation. However, instead of using all the remaining data as training set, we only use the data before the validation set as the training set. We also need to ensure that the data is not shuffled. The diagram below demonstrates how the time series K-fold cross validation works in our case for each day.



After obtaining the best hyperparameter combination, we retrain the model using that combination and all data from day 0 to 1, and the resulting model is evaluated using the test set.

6.1.3 Tuning Results

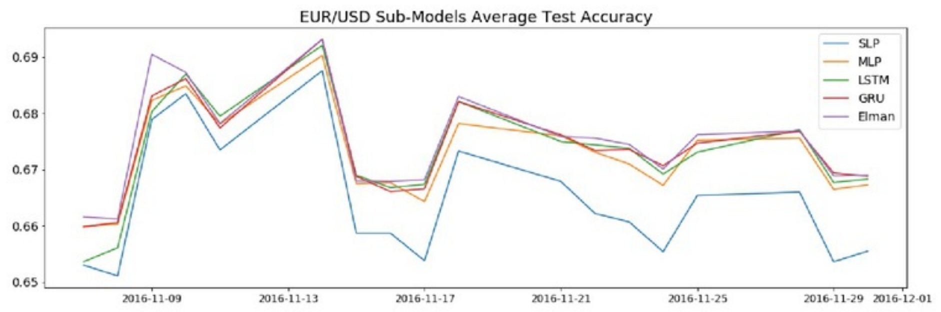
After tuning, we have obtained the test accuracies of the best hyperparameter combinations of each model for each day. To compare the overall performances of each model, we calculate the average and standard deviations of the daily test accuracies.

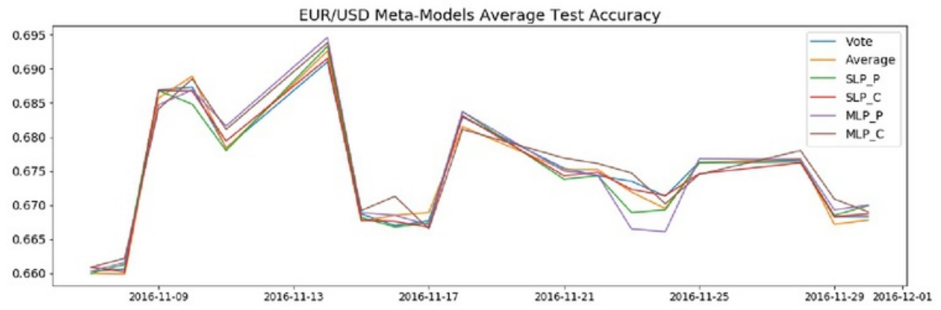
Individual Sub-Models for EUR/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
SLP	0.6644	0.0110
MLP	0.6725	0.0083
LSTM	0.6729	0.0096
GRU	0.6738	0.0087
Elman	0.6748	0.0091

Ensemble Meta-Models for EUR/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
Max Voting	0.6742	0.0077
Averaging	0.6740	0.0081
Stacking SLP (Probability)	0.6738	0.0079
Stacking SLP (One-Hot)	0.6740	0.0078
Stacking MLP (Probability)	0.6740	0.0083
Stacking MLP (One-Hot)	0.6749	0.0076





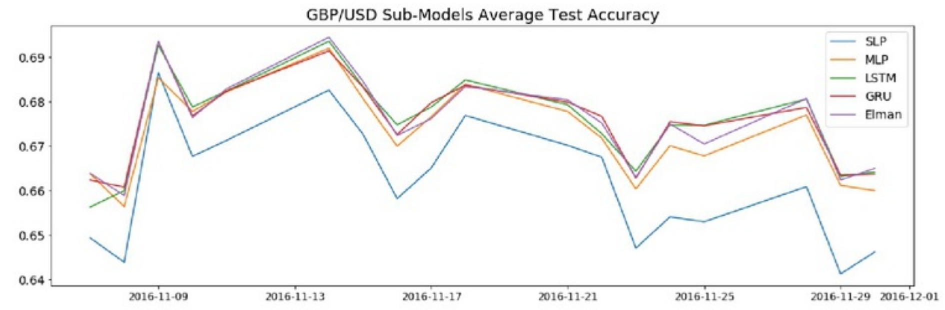
For EUR/USD, the Elman network has the highest average test accuracy, outperforming not only the rest of the sub-models but all models apart from stacking with multilayer perceptron using one-hot sub-model features. The one-hot stacking multilayer perceptron is the only meta-model here that does not hinder the predictive power of the Elman network.

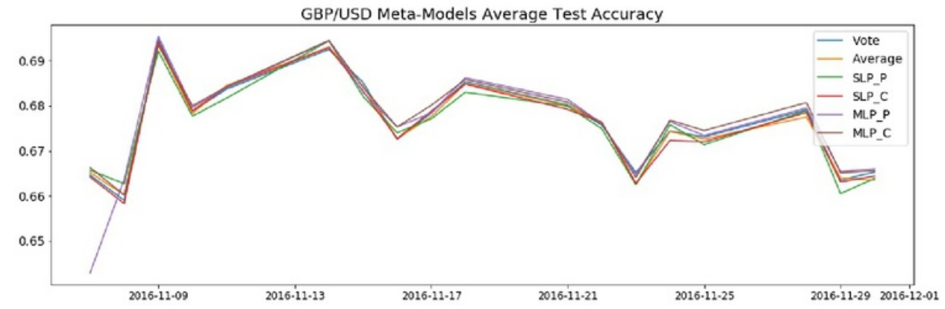
Individual Sub-Models for GBP/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
SLP	0.6619	0.0136
MLP	0.6731	0.0101
LSTM	0.6755	0.0106
GRU	0.6757	0.0098
Elman	0.6755	0.0104

Ensemble Meta-Models for GBP/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
Max Voting	0.6762	0.0101
Averaging	0.6758	0.0100
Stacking SLP (Probability)	0.6755	0.0098
Stacking SLP (One-Hot)	0.6756	0.0103
Stacking MLP (Probability)	0.6759	0.0123
Stacking MLP (One-Hot)	0.6771	0.0098





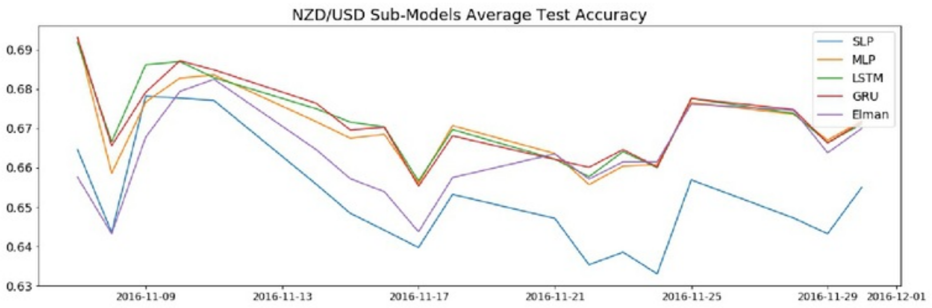
For GBP/USD, the gated recurrent network has the highest average test accuracy and the lowest corresponding standard deviation, thus is clearly the best sub-model. Nevertheless, it is being outperformed by many meta-models. The best overall model is stacking with multilayer perceptron using one-hot sub-model features.

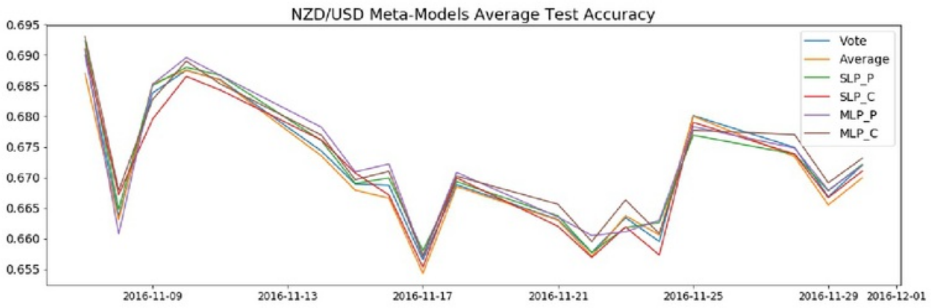
Individual Sub-Models for NZD/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
SLP	0.6521	0.0142
MLP	0.6699	0.0102
LSTM	0.6717	0.0103
GRU	0.6715	0.0102
Elman	0.6631	0.0109

Ensemble Meta-Models for NZD/USD

Models	Average Test Accuracy	S.D. of Test Accuracy
Max Voting	0.6714	0.0104
Averaging	0.6707	0.0105
Stacking SLP (Probability)	0.6720	0.0104
Stacking SLP (One-Hot)	0.6709	0.0104
Stacking MLP (Probability)	0.6724	0.0106
Stacking MLP (One-Hot)	0.6729	0.0100





For NZD/USD, the long short term memory network has the highest average test accuracy and therefore is the best sub-model. Nevertheless, it is being out-performed by some of the meta-models. The best overall model is stacking with multilayer perceptron using one-hot sub-model features.

It is interesting to note that although the best sub-model is different in each currency pair, the best overall model is always the stacking MLP with one-hot sub-model features. Also, on average the ensemble meta models outperform the individual sub-models. This shows the advantage of using ensemble learning that ensemble meta models are more consistent in out of sample predictions. In conclusion, the volatility trends in the main Q-Learning Algorithm should be predicted using the stacking MLP with one-hot sub-model features.

6.2 Main Q-Learning Algorithm

Recall that in reinforcement learning, training and predictions are done at the same time and cannot be separated. Instead, we will be using historical data (November 2016) to pre-train the model before live implementations to improve the initial model performances. Another reason for having the pre-training process is to tune the reward function to be used in the model. Our approach here is to use a grid-search type method. We first optimize the Q-tables with different type of reward functions based on the historical data, then the Q-table with the best performance will then be used as the pre-trained table and be further tested.

6.2.1 Reward Function Tuning

Recall that we will be using reward functions of the following form

$$R_a(s,s') = \begin{cases} p & \text{If } s' \text{ is terminal due to time running out in an episode} \\ f(x(s')) & \text{If } s' \text{ is terminal due to execution} \\ c & \text{Otherwise} \end{cases}$$

where $p < c \leq 0$ are the penalization for non-executions, $f : \{0,1,2,3,4,5\} \mapsto \mathbb{R}^+$ is a non-negative, increasing function in peg-offsets, $x : S \mapsto \{0,1,2,3,4,5\}$ is a function to output peg-offsets from states s'.

Note that that the dimension of the problem is infinite, thus is impossible to find the best combination of p, c and f above. Instead we should restrict the form of f and domain of p and c and then tune them. The key here is to find a good balance in between the penalties p and c for not executing and the reward $f(x(s'))$ we get at the end if executed. We first start with deciding on the forms of the execution reward function f. Since it is easy to get executed at low constant offsets, we want to see if providing a relatively larger reward for higher offsets would encourage the reinforcement agent to explore ways to execute the orders at higher offsets. Therefore, we will test and compare the performance of using the identity function $f(x) = x$, the quadratic function $f(x) = x^2$ and the exponential function $f(x) = \exp(x)$ as the execution reward.

For the non-execution penalty p, we want it to be of a magnitude comparable to that of the execution rewards, as we want the reinforcement agent to distinguish in between non-executions and executions at offset 0. In particular, we want to ensure the algo executes enough orders, thus the penalty cannot be too small. As a result, we test p with the grid $\{0, -2, -5, -10\}$.

As for the intermediate penalty c, we want it to be a relatively small number in magnitude. This is to ensure that the intermediate penalty does not overwhelm the final execution reward. The sole purpose of the intermediate penalty is to distinguish the outcomes of earlier executions at the exact same offset. For instance, the intermediate penalty should be small enough such that executing at offset 3 at the end of the episode is better than executing at offset 2 at the beginning. We set the grid for c as $\{0, -10^{-4}, -10^{-2}\}$.

6.2.2 Initial Peg-Offset

The initial peg-offset is fixed at 4 ticks, due to the following hypothesis - a higher initial peg-offset allows the reinforcement agent to explore more before dropping down to more passive offsets.

6.2.3 Tuning Results

The summary of the tuning results of running the Q-Table over all data once are shown below. The results are benchmarked against the performance of the constant offsets pegging algorithm. The best penalty combinations for each type of execution reward type are highlighted in red. When determining the best, only combinations that produce an execution rate of higher than that of the constant 1-offset pegging algorithm are considered. Combinations with execution rate lower than required are highlighted in blue.

EUR/USD Benchmark

Constant Peg-Offset	0	1	2	3	4	5
Exe-Rate	0.9999	0.9794	0.7439	0.3633	0.1721	0.0810

EUR/USD with Identity Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9966	0.5132	0.9940	0.8000	0.9943	0.7311
-2	0.9962	0.5196	0.9944	0.8000	0.9952	0.7243
-5	0.9942	0.7010	0.9933	0.7786	0.9952	0.7057
-10	0.9957	0.6015	0.9946	0.7805	0.9962	0.6731

EUR/USD with Quadratic Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9944	0.6873	0.9927	0.7950	0.9957	0.6894
-2	0.9938	0.8018	0.9950	0.7861	0.9952	0.7304
-5	0.9926	0.7687	0.9931	0.7456	0.9949	0.7270
-10	0.9942	0.7541	0.9948	0.7263	0.9958	0.5755

EUR/USD with Exponential Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9992	0.1587	0.9989	0.2145	0.9992	0.1593
-2	0.9989	0.1162	0.9988	0.1454	0.9991	0.1553
-5	0.9991	0.1817	0.9991	0.1972	0.9988	0.1282
-10	0.9991	0.1460	0.9992	0.1743	0.9989	0.1844

Notice that combinations with exponential execution rewards have higher execution-rates, at the cost of executing at lower peg-offsets. This is true for the other currencies below as well. Therefore, the choice of execution reward types are determined by the minimum execution rate the trader wants. For EUR/USD, the minimum execution rate required here is 0.9794, which is satisfied by all penalty combinations with all types of execution rewards. The overall best reward function for EUR/USD is the quadratic execution rewards with non-execution final penalty $p = -2$ and no intermediate penalties. Using this reward function means that the agent is indifferent to when the orders are executed, as soon as they are executed by the end of episodes.

GBP/USD Benchmark

Constant Peg-Offset	0	1	2	3	4	5
Exe-Rate	0.9982	0.9623	0.7520	0.4410	0.2385	0.1370

GBP/USD with Identity Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9557	0.8110	0.9505	0.8822	0.9529	0.8903
-2	0.9438	0.9307	0.9483	0.9112	0.9521	0.8481
-5	0.9580	0.7659	0.9551	0.8373	0.9602	0.6986
-10	0.9428	0.9208	0.9520	0.9052	0.9561	0.8327

GBP/USD with Quadratic Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9427	0.8485	0.9604	0.5382	0.9533	0.7473
-2	0.9543	0.7847	0.9523	0.8043	0.9530	0.7860
-5	0.9537	0.7427	0.9469	0.7969	0.9536	0.7906
-10	0.9577	0.7194	0.9487	0.7667	0.9539	0.7383

GBP/USD with Exponential Execution Rewards

p \ c	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9852	0.2760	0.9870	0.2458	0.9861	0.2175
-2	0.9867	0.2280	0.9833	0.3519	0.9874	0.2214
-5	0.9877	0.1987	0.9874	0.2707	0.9852	0.2725
-10	0.9855	0.2661	0.9861	0.2698	0.9870	0.2216

For GBP/USD, the minimum execution rate required is 0.9623. This can only be satisfied by penalty parameter combinations in the exponential execution rewards. In fact, both the identity execution rewards and quadratic execution rewards are on average inferior to the constant 1-offset pegging algorithm. Nevertheless, if the trader wishes to have a chance to gain exposure to higher peg-offsets, this can still be achieved by the identity and quadratic execution rewards, so they are not completely useless relative to the constant 1-offset pegging algorithm.

The overall best reward function for GBP/USD is the exponential execution rewards with non-execution final penalty $p = -2$ and intermediate penalties $c = -10^{-4}$.

NZD/USD Benchmark

Constant Peg-Offset	0	1	2	3	4	5
Exe-Rate	0.9889	0.6248	0.1952	0.0688	0.0323	0.0186

NZD/USD with Identity Execution Rewards

c \ p	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.7579	0.2725	0.7875	0.2537	0.8295	0.1923
-2	0.7410	0.2742	0.7293	0.3193	0.8369	0.1554
-5	0.8019	0.2066	0.8068	0.2271	0.8254	0.1751
-10	0.7772	0.2890	0.7896	0.2713	0.8388	0.1680

NZD/USD with Quadratic Execution Rewards

c \ p	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.7798	0.2630	0.7480	0.3303	0.8471	0.1608
-2	0.7388	0.3010	0.7398	0.3390	0.8249	0.2019
-5	0.8239	0.1805	0.8012	0.2406	0.8092	0.2011
-10	0.7145	0.3951	0.7934	0.2766	0.8272	0.1631

NZD/USD with Exponential Execution Rewards

c \ p	0		-10^{-4}		-10^{-2}	
	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset	Exe-Rate	Avg-Offset
0	0.9504	0.0535	0.9524	0.0496	0.9457	0.0598
-2	0.9564	0.0337	0.9529	0.0557	0.9513	0.0498
-5	0.9499	0.0670	0.9506	0.0554	0.9539	0.0394
-10	0.9550	0.0438	0.9527	0.0580	0.9481	0.0350

Similar to the results in EUR/GBP, the minimum execution rate required here is 0.6248, which is satisfied by all penalty combinations with all types of execution rewards. The overall best reward function for NZD/USD is the quadratic execution rewards with non-execution final penalty $p = -5$ and no intermediate penalties.

For each currency, we have obtained the corresponding pre-trained Q-Tables with the best reward functions. These Q-Tables will be used as initialization for live implementation of the dynamical pegging algorithm.

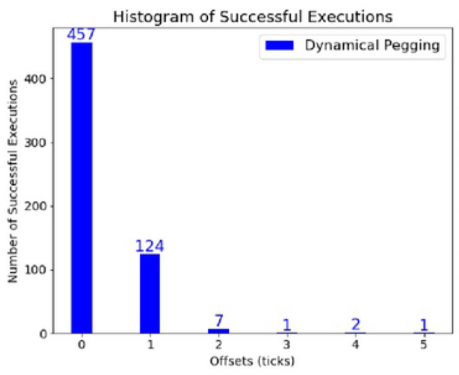
7 Algorithm Evaluation

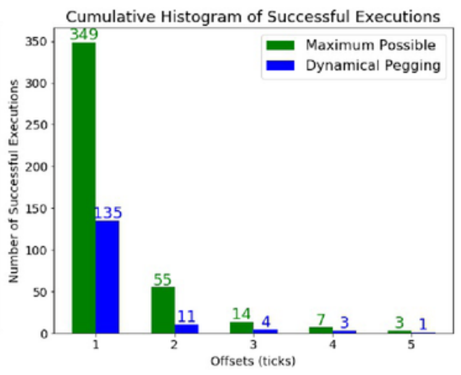
In this section, the dynamical pegging algorithm is implemented live from 19th to 23rd August 2019, using the pretrained Q-Tables in last section as initialization and setting $\epsilon = 0.05$. Volatility trends are predicted using tuned Stacking MLP with one-hot encoded features. Note that the next state's volatility trend predictions are unknown in real time. As a result, the Q-Table update is lagged by one time unit. The results, including the constant offset pegging algorithms (denoted as C0 to C5) as benchmarks, the distributions of the offsets from 0 to 5 in the dynamical pegging algorithm and the cumulative distribution of offsets from 5 to 1.

7.1 EUR/USD Results

Peg Algo	Dynamical	C0	C1	C2	C3	C4	C5
Exe-Rate	0.9136	0.9992	0.6511	0.1986	0.0579	0.0293	0.0127
Avg-Offset	0.2601	0	1	2	3	4	5

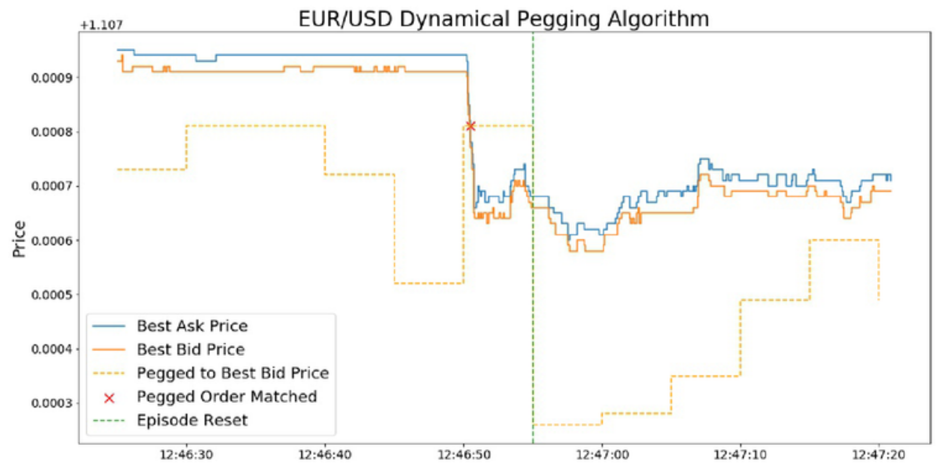
The above table shows that the EUR/USD dynamical pegging algorithm has a performance better than algorithm C0, provided that we use an execution rate benchmark of algorithm C1. In fact, as shown from the histogram of successful executions below, the dynamical pegging algorithm can be interpretted as mainly a hybrid in between algorithms C0 and C1, while occasionally matching the orders at higher offsets as a bonus.





To further assess the ability of the dynamical pegging algorithm to match orders at higher offsets, the cumulative histogram of successful executions on the right graph above is plotted. In there the bars in green represent the total numbers of successful executions using constant offset pegging algorithms. Those are also the maximum possible number of successful executions achievable at corresponding offsets. The bars in blue are the total numbers of successful executions using the dynamical pegging algorithm with offsets greater or equal to the corresponding offsets. Note that since the Q-learning episodes start straightaway after the previous ones ended, the total number of episodes are different for each algorithm. In particular, the total number of episodes are much higher for constant pegging algorithms with lower offsets, which makes the maximum possible number of successful executions at lower offsets much higher than that of the dynamical pegging algorithm. Therefore, it may be misleading to think that the dynamical pegging algorithm does not work well due to the difference in successful executions at lower offsets. Instead, we should mainly focus on the higher offsets when evaluating the cumulative histogram. We can see that although potential successful executions at very higher offsets are scarce as seen from low number of corresponding maximum possible successful executions, the EUR/USD dynamical pegging algorithm is still able to find some of them, while maintaining a high execution rate of 0.9136 compared to those of high-offsets constant pegging algorithms. This suggests that the EUR/USD dynamical pegging algorithm here is superior if the objective of the trader is to be able to execute successfully at a decent rate while gaining exposure to higher-offsets.

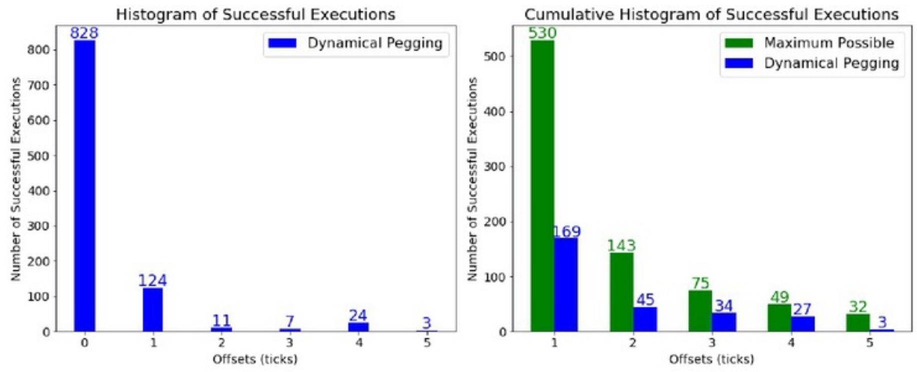
Below is a demonstration of the EUR/USD dynamical pegging algorithm in action, where it successfully executes an order and then resets to the next episode.



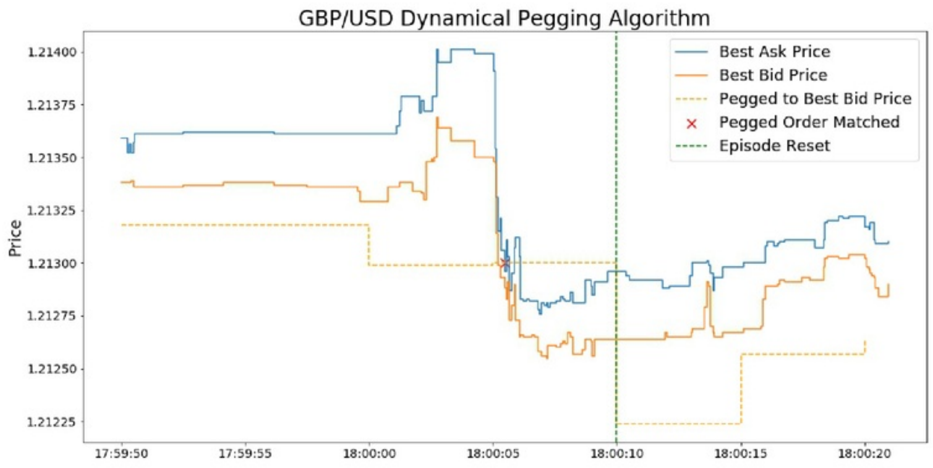
7.2 GBP/USD Results

Peg Algo	Dynamical	C0	C1	C2	C3	C4	C5
Exe-Rate	0.9147	0.9844	0.7864	0.4063	0.2534	0.1788	0.1231
Avg-Offset	0.2788	0	1	2	3	4	5

The results of GBP/USD is very similar to that of EUR/USD, with the dynamical pegging algorithm acting as a hybrid in between algorithms C0 and C1. The difference is that in here, there are higher numbers of potential successful high-offset executions. Nevertheless, there are also comparatively higher numbers of successful high-offset executions from the GBP/USD dynamical pegging algorithm. It is also interesting to note that the dynamical pegging algorithm is able to find more than half of the maximum possible number of successful executions at offset 4.



Below is a demonstration of the GBP/USD dynamical pegging algorithm in action, where it successfully executes an order and then resets to the next episode.

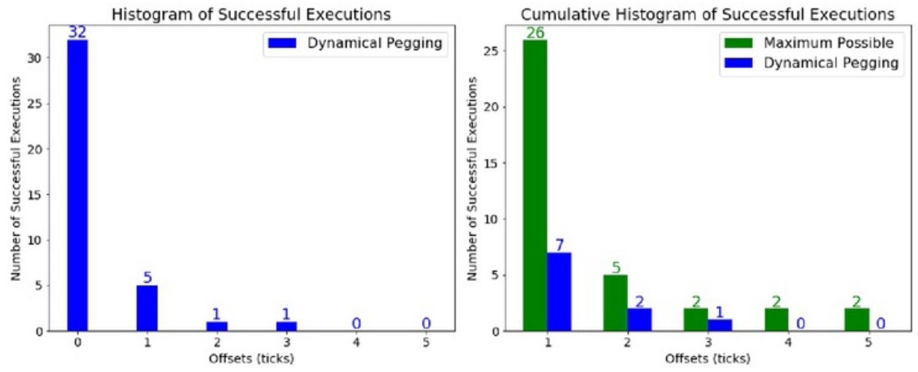


7.3 NZD/USD Results

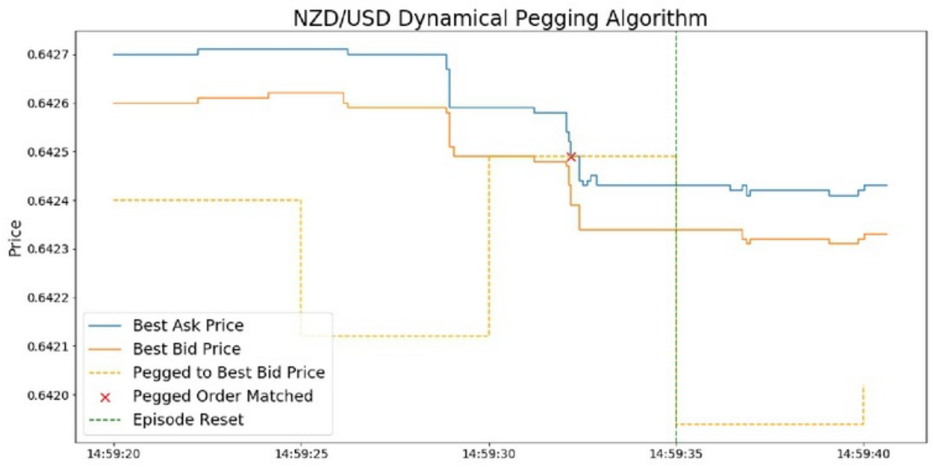
Peg Algo	Dynamical	C0	C1	C2	C3	C4	C5
Exe-Rate	0.1548	0.7381	0.1032	0.0211	0.0085	0.0085	0.0085
Avg-Offset	0.3677	0	1	2	3	4	5

In contrast to the cases of both EUR/USD and GBP/USD, the execution rates of all pegging algorithms for NZD/USD are much lower. This may be due to lower NZD/USD volatilities in this period. As a result, even though the NZD/USD dynamical pegging algorithm acts as a hybrid of algorithms C0 and C1 as the cases in EUR/USD and GBP/USD, the NZD/USD dynamical pegging algorithm here is sub par, as the execution rate of 0.1548 is too low to be useful in practise. In scenario like this, it may be better to use algorithm C0 instead, or alter the reward functions to the exponential execution reward types for higher successful execution rates.

The NZD/USD dynamical pegging algorithm also performs worse in terms of finding high-offsets execution opportunities, compared to the cases in EUR/USD and GBP/USD.



Below is a demonstration of the NZD/USD dynamical pegging algorithm in action, where it successfully executes an order and then resets to the next episode.



8 Conclusion and Future Work

In this thesis we have introduced an approach of building the dynamical pegging algorithm for foreign exchanges via a combination of supervised learning and reinforcement learning methods. We have also compared the performance of the dynamical pegging algorithm to that of the standard pegging algorithm with constant peg-offsets, on EUR/USD, GBP/USD and NZD/USD. While not perfect, the results are encouraging in the sense that the dynamical pegging algorithms are able to distinguish themselves from the standard ones. In particular, under certain benchmarks, the dynamical pegging algorithms are able to outperform the standard pegging algorithms with constant peg-offsets. Nevertheless, the dynamical pegging algorithm still have lots of rooms for explorations and improvements. A non-exhaustive list of such is shown below

- Incorporate more data (e.g. level 2 data in the order book) in the machine learning models
- Build the dynamical pegging algorithm for other currency pairs or asset classes
- Run the Q-learning system for longer periods of times for it to converge more and lead to a better performance
- Explore other forms of reward functions in the main Q-learning Algorithm
- Explore using extensions of the Q-learning algorithm, such as the deep Q network, to improve slow convergence rate of the trivial Q-learning method due to high state dimensions
- Explore incorporating other sub-models that are different from neural networks into the meta-stacking models for volatility trends predictions
- Explore different sampling distributions in random search
- Explore other methods for hyperparameter tunings, e.g. Bayesian optimization or Genetic Algorithms
- Explore pegging with a lower frequency such that the volatilities are less noisy
- Explore using different volatility measures

References

[1] A. Azhikodan, A. Bhat, M. Jadhav. (2019). Stock Trading Bot Using Deep Reinforcement Learning. *10.1007/978-981-10-8201-6_5.*

[2] B. Ning, F.H. Ling, S. Jaimungal. (2018). Double Deep Q-Learning for Optimal Execution. *ArXiv, abs/1812.06600.*

[3] Chuong Luong and Nikolai Dokuchaev. (2018). Forecasting of Realised Volatility with the Random Forests Algorithm. *Journal of Risk and Financial Management.*

[4] Diederik P. Kingma, Jimmy Lei Ba (2015) *Adam: A Method for Stochastic Optimization.* Preprint available at arXiv:1412.6980

[5] Elman, Jeffrey L. (1990). Finding Structure in Time. *Cognitive Science.* **14** (2): 179211.

[6] Eyal Even-Dar; Yishay Mansour. (2003). Learning Rates for Q-learning. *Journal of Machine Learning Research 5*, 2003 1-25.

[7] Felix A. Gers, Jürgen Schmidhuber, Fred Cummins (2000). Learning to Forget: Continual Prediction with LSTM. *Neural Computation.* **12** (10): 2451-2471.

[8] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* Preprint available at arXiv:1406.1078.

[9] Rong Ge, Furong Huang, Chi Jin, Yang Yuan (2015). Escaping From Saddle Points-Online Stochastic Gradient for Tensor Decomposition. *Proceedings of The 28th Conference on Learning Theory, PMLR* 40:797-842, 2015.

[10] Sepp Hochreiter, Jürgen Schmidhuber (1997). Long short-term memory. *Neural Computation.* **9** (8): 1735-1780.

[11] Shunrong Shen, Jiang Haomiao and Tongda Zhang. *Stock Market Forecasting Using Machine Learning Algorithms.* (2012).

[12] Singh, Harshinder & Hnizdo, V & Fedorowicz and E. Demchuk, Adam & Misra, Neeraj. (2003). Nearest Neighbor Estimates of Entropy. *American Journal of Mathematical and Management Sciences.* 23. 10.1080/01966324.2003.10737616.

[13] Sutton, Richard S.; Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction.* MIT Press. ISBN 978-0-262-19398-6.

[14] Z Zheng, Z Qiao, T Takaishi, HE Stanley, B Li (2014). *Realized Volatility and Absolute Return Volatility: A Comparison Indicating Market Risk*. PLoS ONE 9(7): e102940. https://doi.org/10.1371/journal.pone.0102940.

A Machine Learning Approach to Improve the Pegging Algorithm

GRADEMARK REPORT

FINAL GRADE	GENERAL COMMENTS
/0	**Instructor**

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21	
PAGE 22	
PAGE 23	
PAGE 24	
PAGE 25	
PAGE 26	
PAGE 27	
PAGE 28	
PAGE 29	
PAGE 30	
PAGE 31	
PAGE 32	
PAGE 33	
PAGE 34	
PAGE 35	
PAGE 36	
PAGE 37	
PAGE 38	
PAGE 39	
PAGE 40	
PAGE 41	
PAGE 42	
PAGE 43	
PAGE 44	
PAGE 45	

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51