Optimizing Market Making using Multi-Agent Reinforcement Learning

Yagna Patel
yagnapatel@berkeley.edu

***Abstract*— In this paper, the concept of deep reinforcement learning is applied to the problem of optimizing market making. A multi-agent reinforcement learning framework is employed to optimally place limit orders that lead to successful trades. Specifically, two agents are used: one agent, the macro-agent, optimizes on making the decision to buy, sell or hold an asset, while the other, the micro-agent, optimizes on placing limit orders within the orderbook. For the context of this paper, the proposed framework is applied and studied on the Bitcoin cryptocurrency market.**

I. Introduction

Algorithmic trading, and in particular high frequency algorithmic trading (HFT), has gained immense popularity in the recent decade. With advances in hardware and software, algorithmic trading has rapidly become the norm. The increasing popularity of machine learning has slowly made its way across to financial markets [1], where it is primarily used to predict price movements of assets. However, there are a number of challenges that these classic machine learning approaches bring:

1) *Prediction time:* In machine learning, model complexity can have an impact on prediction time. Many times, in the supervised learning setting, neural networks are used to make predictions. Due to the computational complexity that comes with these models, as the model complexity increases, the decision time also increases [2]. In the HFT setting, by the time your model makes a prediction, it may already be too late to take the predicted action. The problem then becomes, how can these added latency costs be incorporated into our prediction?
2) *Prediction accuracy:* The general rule of thumb in finance is that the historical performance of an asset does not predict the future performance of that asset, i.e. predicting the market is tough. In fact, markets are sometimes compared to random walk processes [10]. Therefore, approaches that solely rely on the historical performance of an asset are likely to not have high prediction accuracy.
3) *Policy optimization:* Suppose that our model predicted a 55% chance of increase and a 45% chance of decrease for some given stock. How can our model's prediction be converted into an action? An optimized policy and certain decision thresholds are needed to be able to turn a prediction into an action. However, coming up with such a policy is usually done by hand. The development, optimization, and back-testing of these policies are commonly done by mix of humans and computers. Therefore, if the market suddenly shifted, these policies would likely not be able to adapt.

II. Related Work

The concept of deep reinforcement learning is relatively new in finance. Therefore, there has been very little research done into whether or not is a viable approach in placing trades. Multi-agent approaches to stock trading have been taken previously [9], however, they do not account for placing limit orders. The general concept of the micro-agent (see section 3) has been shown by Nevmyvaka et al. [3] and further extended by Juchli [4]. These papers present the problem of optimally buying or selling an asset over a given time horizon. This is very similar to the purpose of the micro-agent. The work for the micro-agent will substantially be based, and build up on the ideas presented in these two papers.

III. Problem Formulation

In this paper a less common approach of reinforcement learning is utilized to optimize market making. More specifically, a multi-agent approach (with two agents) is used:

1) *Macro-agent:* The macro-agent will be given minute tick data, data at a macro-level, and will make the decision to buy, sell, or hold an asset.
2) *Micro-agent:* The micro-agent will be given orderbook data, data at a micro-level, and will make the decision of where to place a limit order.

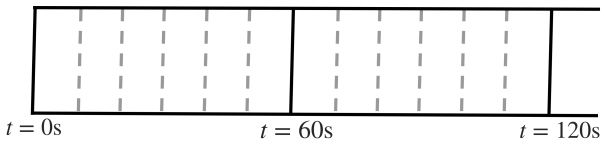

Fig. 1. Time-steps where agents will take actions. The Macro-agent takes actions at every bold black line, while the Micro-agent takes actions between the dashed lines.

At the start of every minute, the macro-agent will make its decision to buy, sell or hold an asset. Additionally, a running count of how many buys an agent chooses to make is kept until the first sell, at which point the count will reset. The total buys before the first sell will be fed into the micro-agent, i.e. all collected assets are exhausted on each decision to sell. Within the 60 seconds that follow, the micro-agent will attempt to exhaust it's held assets within this time horizon by placing limit orders. It may place multiple limit orders, however the the micro-agent is limited to one order placement every 10 seconds, as depicted in Fig. 1.

A. Data Collection

Historical orderbook and minute tick data for markets is not readily available. Therefore, the first step was to collect data. The Bitcoin market was used in this study for the reason that its data is readily available. All trade, bid, and ask data was collected by subscribing to Bittrex's websocket in the date range of November 2nd, 2018 to November 17th, 2018. With data provided by the websocket, a historical orderbook was constructed, i.e. a sequence of historical orderbook states over that time period. A minute tick dataset was able to be created using the trade data collected. In total, there were 41830629 trade, bid, and ask data points, and 10945 minute tick data points (Fig. 2). However, the entire dataset was not used. Instead, only the most recent few days were chosen for this study.

Fig. 2. Bitcoin Minute tick data collected over 2018-11-02 to 2018-11-17

B. Brief Overview of Reinforcement Learning

Reinforcement learning is a learning technique in machine learning, in particular sequential decision making, where an agent learns to take actions optimally in an environment. Unlike supervised learning, in the context of a reinforcement learning problem, this agent learns to take actions that maximize the reward it receives from the environment. At each time-step, the agent observes its environment, and takes an action based on the observation. The environment provides feedback on how well the action performed in the form of a reward. In the case of this problem, these rewards can be delayed. An instant reward feedback for an action to buy, sell, or hold, or placing a limit or market order at the current time-step may not be given. For example, the decision to hold an asset may not yield an instant reward.

This process of an agent interacting with the environment is formalized as a Markov Decision Process (MDP). MDPs are used to describe the environment in the context of a reinforcement learning problem. The importance behind MDPs in this problem is the Markov property that MDPs assume: the effects of some action taken in some state depends only on that state and not on prior states encountered. The second challenge mentioned earlier suggests that market prices are Markovian in nature, i.e. the probability distribution of the price of an asset depends only on the current price and not on the prior history. Therefore, it makes sense to formulate this problem as an MDP problem with the goal being to solve this MDP by finding an optimal policy. For each agent, the tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$ describes each problem. $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the transition probability distribution which determines how the environment changes based on the state and actions taken by the agent. $r : \mathcal{S} \to \mathbb{R}$ is the reward function. And $\gamma \in (0, 1)$ is the discount factor which determines the importance of future rewards. Therefore, at time-step t, the future discounted reward is given by

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} \cdot r_i$$

where T is the last time-step, potentially infinity.

However, one thing to note here is that for the case of market making, the state is more complex than it may seem. In fact, the state encountered is most often not the complete state since there exist many other traders in the environment. Therefore, instead of an MDP, this is a Partially Observable Markov Decision Process (POMDP). The state the agent observes, $\mathcal{S}'$, is some derivation of the true state, $\mathcal{S}$, i.e. $\mathcal{S}' \sim \mathcal{O}(\mathcal{S})$. Nevertheless, given a proper simulation of the environment, the agent should be able to optimize well against these unknowns.

As mentioned earlier, a reinforcement learning agent continuously goes through a cycle of states by interacting with the environment. In this context, there are two agents. The interaction between these agents and the environment are depicted in Fig. 3.

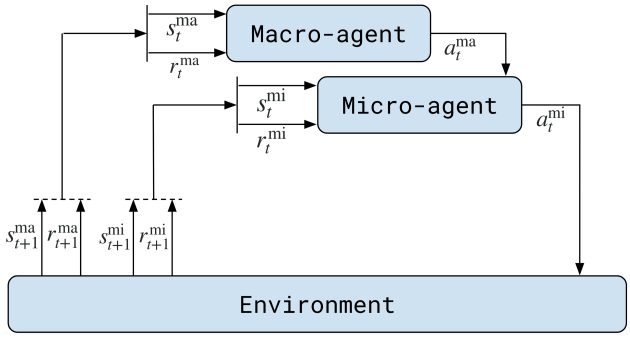

Fig. 3. Multi-agent Reinforcement Learning Framework

Both agents encounter states, and output actions. These actions collectively turn into a single action: placing an order in the orderbook. The environment takes this action and outputs a reward and the next state. As noted by the problem statement, discrete time-steps are chosen rather than continuous time-steps for the reason that continuous time-steps would not be possible in the real world since the web-socket data arrives at discrete time-steps. The specific reinforcement learning algorithm used for both agents is deep Q-learning. Recall that the goal is to find an optimal policy $\pi : \mathcal{S} \to \mathcal{A}$, a function mapping between states and actions, such that it maximizes the expected reward. Q-learning is a model-free value-based approach, which uses the action-value function $Q(s, a)$, where $s \in \mathcal{S}$ and $a \in \mathcal{A}$, to estimate

the expected reward, given some state-action pair, i.e. under policy π, the true value of being in state s and performing action a is given by

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t \mid \mathcal{S}_t = s, \mathcal{A}_t = a, \pi\right]$$

Now, typically the Q-values are updated using the bellman equation

$$Q_{t+1}(s_t,a_t) \\ = (1-\alpha)\cdot Q_t(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_a Q_t(s_{t+1},a))$$

However, in the case of this problem, a neural network is used to approximate the Q-value function, where the input is the state (instead of state-action pairs in Q-learning), and the output are the Q-values for the various actions. Thus, the Q-value function is parameterized by weights θ, i.e. assume that

$$Q(s,a;\theta) \approx Q^*(s,a)$$

In reinforcement learning, there is a trade-off between exploration and exploitation: how often do we want our agent to continue taking actions with the policy it has determined versus how often do we want our agent to explore taking new actions. To get a good balance between both worlds, a decaying ϵ-greedy approach is used, in which the agent takes random actions with probability ϵ, and takes the action determined by the policy with probability $1-\epsilon$, while ϵ decays over time. This encourages the agent to explore more at the beginning, and exploit more at the end, i.e. stick to the policy it has computed.

Another key concept that is utilized is experience replay. In online learning, we receive data sequentially. Therefore, if for some reason the market suddenly shifted, the agent may forget its past experiences as it aims to get a better advantage in the new state. By storing the agent's experiences in a memory data structure, and having the agent relive those randomly sampled batches of experiences often, it helps to alleviate the high temporal correlation in the data. Additionally, training on these randomly sampled batches, gives the feel of training on iid data. Therefore, as in supervised learning problems, this should yield better convergence for the function approximator to the optimal policy. This also allows to update the parameters of the function approximator with a stochastic gradient descent approach to minimize the loss.

IV. Details of the Macro-agent

The Macro-agent is responsible for making a discrete decision to buy, hold, or sell an asset by looking at the data from a macro level. The data used here is minute tick data. Although the data is collected over a span of multiple days, the entire data-set is not used for the reason that the agent may under-explore the more recent, more relevant states, and over-explore other states. Since most of the data had little volatility, while the more recent data was much more volatile, the agent would not have been able to perform well in volatile states due to exploring the less volatile states more often. Therefore, the recent minute tick data spanning the course of the last few days is used: November 15th, 2018 to November 17th, 2018. Fig. 4 shows the portion of the data that was chosen.

Fig. 4. Bitcoin Minute tick data (2018-11-15 00:00 to 2018-11-17 17:06)

To train the agent, the dataset is split into training and test sets. The training set consists of the time-steps prior to November 16th, 2018.

A. State

The state space at a certain time-step t consists of historical price data in the range $t-h$ to t, where h denotes how far back in history the agent looks. Furthermore, featurized data of various momentum and reversion indicators are also incorporated into the state space. These indicator features are computed in the following ways:

Market Indicator Features

One commonly used feature in trading is the z-score indicator, which essentially determines how far (in standard deviations σ) a data point x is from the mean μ:

$$z_x = \frac{x-\mu}{\sigma} \tag{1}$$

In a way, the z-score indicator reveals anomalies in data. It has also been proven to be a useful indicator in determining future trends [5]. Equation (1) can be applied by using an expanding window approach with a window size of n time-steps. Let p_t be the closing price at time-step t, let $\mathrm{SMA}(p_{t-n,t})$ denote simple moving average (SMA) of closing prices over n time-steps prior to t, and let $\mathrm{STDDEV}(p_{t-n,t})$ denote the standard deviation of closing prices over n time-steps prior to t. Thus, the z-score at time-step t for a window size of n is

$$z_t^n = \frac{p_t - \mathrm{SMA}(p_{t-n,t})}{\mathrm{STDDEV}(p_{t-n,t})}$$

The z-score indicator can be extended to volume as well. Thus, the following are the features that were extracted:

- *Price Level:* To determine price levels, the z-scores were calculated for the prices. This essentially expresses how far each of the prices in the time period are from the average price in the time period.
- *Price Change:* To determine price changes, the current price is compared to the average of a window of prices

prior to it, i.e. calculate for a window size n at time-step t

$$\mathrm{PC}_t^n = \frac{p_t}{\mathrm{SMA}(p_{t-n,t})} - 1$$

and take the z-score of the result.

- *Volume Level:* To determine volume levels, the z-scores were calculated for the traded volumes. As with price levels, this expresses how far the volume at each time step is from the volume in the time period.
- *Volume Change:* Similar to price change, volume change for a window size n at time-step t is calculated:

$$\mathrm{PC}_t^n = \frac{v_t}{\mathrm{SMA}(v_{t-n,t})} - 1$$

and the z-score of the result is taken.

- *Volatility:* To determine volatility, exponential moving averages (EMA) are used to determine the rate of price change over a span of n days. The reason for using EMA instead of SMA is that EMA gives more weight to recent data than SMA. Let p_t be the current price, and let n be the window size. The EMA at time-step t is given by

$$\mathrm{EMA}_t^n = \frac{2p_t}{n+1} + \frac{\sum_{j=t-n}^{t} p_j}{n}\left(100 - \frac{2}{n+1}\right)$$

Thus, the volatility over m days is given by

$$\mathrm{Volatility}_t^m = \frac{\mathrm{EMA}_{t,n} - \mathrm{EMA}_{t-m,n}}{\mathrm{EMA}_{t-m,n}}$$

Along with these market indicator features and the price data, there is an additional state parameter which will play an essential role in determining rewards. Each time the agent chooses to buy, the price at which it bought the asset is stored into a current assets list. This list of prices is stored as part of the state as well. Each time the agent chooses to sell, all of the bought assets are exhausted (sold). Thus, the historical prices, market indicator features, and current assets list form the state.

B. Action

There are three actions that the agent can choose:

1) *Buy:* If the decision to buy is chosen, then the agent is choosing to buy 1 bitcoin at the current opening price. Additionally, when it chooses to buy (as mentioned previously), the price gets appended to the current assets list. Note that the decision to buy 1 bitcoin is purely a design choice. The proposed framework should handle any quantity of bitcoins.
2) *Sell:* If the decision to sell is chosen, then the agent is choosing to sell all of its accrued assets at the current opening price.
3) *Hold:* If the decision to hold is chosen, then the agent will not do anything.

Algorithm 1: Macro-agent Reward Function

```
if action = hold then
    reward ← 0
else if action = buy then
    Append current open price to current assets list
    reward ← 0
else if action = sell then
    if there are no asssets to sell then
        reward ← −1
    else
        reward ← sell off all assets from current assets
          list and determine profit based on current open
          price
    end
Clip Rewards
```

C. Reward

After the agent takes an action, the environment outputs a reward for that action. Algorithm 1 depicts how rewards are handled.

Note that the rewards are clipped to $\{-1, 0, 1\}$ based on negative, zero, or positive rewards, respectively. This is done in order to have the agent deal with different types of market environments, e.g. high/low volatility, as well as reduce the impact of anomalous market shifts. Clipping rewards has proven to be a useful method when dealing with different scales of rewards when DeepMind applied it to various Atari games [6].

D. Deep Q-Network Architecture & Training

To parameterize the value function, a deep Q-network is used. A simple multilayer perceptron with two hidden layers (with ReLu activation functions) is chosen. Furthermore, the Adam optimizer is used for training. The Q-network takes in as inputs the price data, market indicator data and the current assets list, and outputs 3 Q-values which are then used to determine the optimal action. In Fig. 5 a depiction of the training framework for the macro-agent is provided.

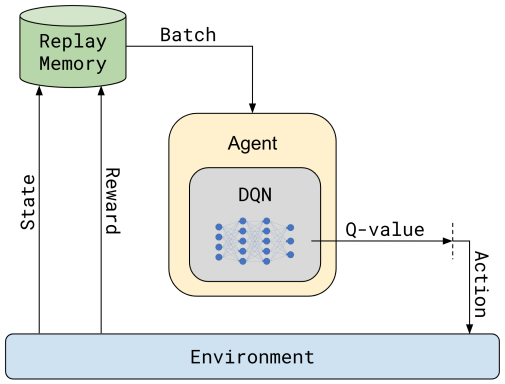

Fig. 5. Macro-agent Training Framework

The details of how exactly the interactions happen in this framework, are outlined in Algorithm 2. It outlines and gives

a closer look at the way the macro-agent is trained. Note that for each epoch the agent is trained on the entire training set.

Algorithm 2: Macro-agent Deep Q-Learning Training

Initialize replay memory M
Initialize ϵ, and discount factor γ
Initialize Q network with random weights
for *epoch* $= 1$ *to* E **do**
  $s_t \leftarrow$ **Reset** environment (environment outputs initial state)
  while *not done* **do**
    $a_t \leftarrow$ **Select** random action with probability ϵ or choose action $\max_a Q(s, a; \theta)$
    $s_{t+1}, r_t, done \leftarrow$ **Act** based on a (environment outputs new state, reward, and done)
    if *replay memory* M *is full* **then**
      Remove the first element from M
    Append $(s_t, a_t, r_t, s_{t+1}, done)$ to replay memory M
    $b \leftarrow$ **Sample** mini-batch from replay memory M at random
    $q \leftarrow$ **Initialize** empty array of size $|b|$
    foreach $(s_i, a_i, r_i, s_{i+1}) \in b$ **do**
      $$q_i = \begin{cases} r_i + \gamma \cdot \max_a Q(s_{i+1}, a_i; \theta) & \text{not } done \\ r_i & done \end{cases}$$
    end
    Apply gradient descent with loss $\mathbb{E}_{s,a,r,s'}\left[(q_i - Q(s_i, a_i; \theta))^2\right]$
    Decay ϵ
  end
end

E. Results

It is interesting to see how the macro-agent performs on its own. Although the closing and opening prices, the prices used to calculate profits for the macro-agent, are usually not indicative of the true prices where trades may happen, they can hypothetically be assumed so to evaluate the performance of the agent. For these results, the agent was trained on the training set and tested on the test set, as defined previously, for 500 epochs.

Now, to truly understand the performance of the macro-agent strategy, it was compared to two common investment strategies:

- *Buy and Hold investing:* Buy and Hold investing is a very naive long-term investment strategy where an investor chooses to buy an asset at the start of a time period and holds the asset in the hopes that it will accrue value over time. From the looks of Fig. 4 it is not expected for this strategy to yield any positive profit as the general trend (for the test set) is downwards. For this case, 10 Bitcoins are chosen to be bought and held over the time period.
- *Momentum investing:* Momentum investing is where an investor chooses to buy or sell an asset at a certain time-step given the performance of the asset in the last n steps. To implement this, a simple moving average of the prices with a window size of $n = 20$ is computed, i.e. 20 minutes prior. At each time-step, if the current price open price is less than the average, then 1 Bitcoin is bought. If the current open price is greater than the average, then all Bitcoins bought thus far will be sold. In the case they are equal, the decision to hold is made.

These investment strategies will be compared with the macro-agent, which is similar to momentum investing with the only difference being that the agent chooses when to buy, sell, or hold. The way the macro-agent handles buying and selling is equivalent to that momentum investing. Fig. 6 depicts the performance comparisons between the different strategies. In order to simulate the Buy and Hold investing strategy, at each time-step the profit is plotted assuming that the investor decided to sell at each time-step, i.e. if p_0 is the price the asset was initially bought at, at each time-step the difference $p_t - p_0$ is plotted.

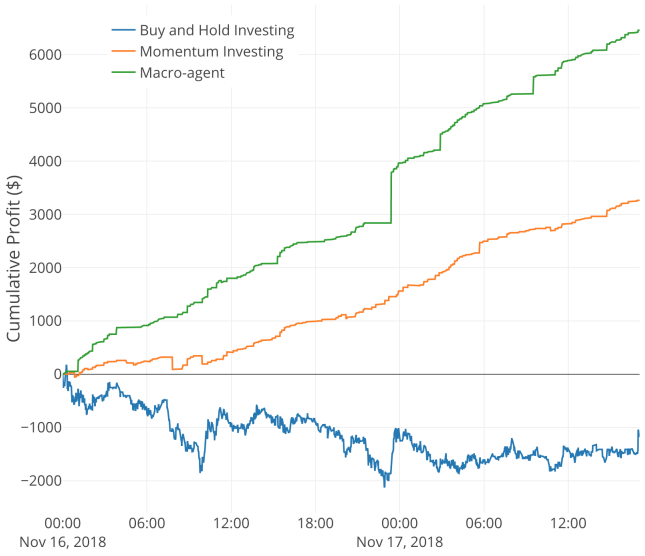

Fig. 6. Performances of Various Investment Strategies PNL-Realized Graph

Fig. 6 compares the strategies in terms of profit. The accumulated profit is plotted for the momentum investing strategy and the macro-agent. Notice that for the macro-agent the accumulated profit growth is not very volatile, indicating that the macro-agent is stable as well.

V. Details of the Mirco-agent

In the previous section the results showed that the macro-agent performed well compared to two other investment strategies. However, one issue with the macro-agent was that it assumed that the trade occurred at the opening prices. However, in real market environments a trade happening at the opening price is most often not the case. Therefore, there lies this uncertainty in the framework thus far of what price will, or should, the order be set at. This motivates the purpose

for the micro-agent. Many of the ideas used here are derived from [3] and [4].

A. Introduction

In markets, there is this concept of an orderbook. The orderbook is separated into two parts: bids and asks. The bid section contains all the prices and quantities buyers are willing to buy at. The ask section contains all the prices and quantities sellers are willing to sell at. Now, there are two main ways that a trader can place an order in the orderbook:

1) *Market Order:* When a trader places a market order, they are placing an order at the market price, i.e. the best price of the opposing side of the orderbook. Most often, this order is immediately filled. However, depending on the quantity that the order was placed for, the order might not be executed at the same prices, as it depends on how much quantity is available at market price. Another disadvantage is that market orders usually come with additional trading fees. Depending on the state, this may sometimes even yield negative profit.
2) *Limit Order:* When a trader places a limit order, they are essentially demanding that price (or a better price). A limit order guarantees the trader the price, however, it is possible for the order to never be filled. When a trader places a limit order, they stand behind all the traders that have placed an order at that price. Additionally, limit orders provide liquidity to the market, and so exchanges incentivize limit orders.

Once a trader creates an order, there is a match engine which attempts to match the trader's order to orders on the opposing orderbook side. Only when there is a matching order does a trade happen.

In this case, the agent should be able to optimally place orders in the orderbook. As noted previously, the agent has two order options: market order or limit order. However, there is a trade-off between placing market orders and placing limit orders. Therefore, the goal of the agent is to optimize on this trade-off: is there an action better than simply placing a market order at the beginning of the time horizon?

This is a challenging optimization problem since there are many unknown variables here:

- *Other Market Participants:* This agent is not the only one trading on the market. There are many other market participants that the agent interacts with.
- *Adversarialness:* Among these market participants, there are also algorithmic traders, some of who might be playing adversarially, e.g. whales.

Our agent must optimize over these unknowns as well. In order to do that, the match engine needs to be simulated. Since this problem is posed as a reinforcement learning problem, such a simulator is essential, as the agent needs proper reward feedback. An open-source matching engine [7] is used to help simulate the actual match engine. However, although this matching engine will be able to simulate well, it is only indicative to the given time period the data was collected for. Therefore, there is still a disadvantage to this.

B. Environment

In reinforcement learning problems, the environment is essential since it provides the agent with subsequent states, as well as reward feedback. The environment for this problem is vastly different than in the macro-agent. The environment takes in as input the order side, i.e. buy or sell, and the quantity. These inputs are provided by the macro-agent.

SUM	TOTAL	SIZE (BTC)	BID (USD)	ASK (USD)	SIZE (BTC)	TOTAL	SUM
1106.4034	1106.4034	0.324	3416.00000000	3421.36000000	0.342	1170.9533	1170.9533
2239.0516	1132.6482	0.332	3411.11600000	3421.39000000	1.500	5132.0850	6303.0383
7354.2466	5115.1950	1.500	3410.13000000	3428.41400000	1.000	3428.4140	9731.4523
7430.4043	76.1577	0.022	3409.00000000	3428.41500000	0.010	34.2841	9765.7365
7501.0930	70.6887	0.021	3402.00000000	3428.98000000	0.059	203.2175	9968.9539
10902.4790	3401.3860	1.000	3401.38600000	3433.00000000	0.015	51.4950	10020.4489
18071.1961	7168.7171	2.109	3399.65500000	3433.14400000	2.311	7933.8514	17954.3003
18074.6577	3.4617	0.001	3398.43000000	3433.62700000	0.571	1959.2090	19913.5093
26194.3529	8119.6952	2.395	3390.90000000	3440.99900000	2.584	8890.3737	28803.8830
26358.6350	164.2821	0.048	3389.88000000	3442.43300000	0.010	34.4243	28838.3073
60253.8150	33895.1800	10.000	3389.51800000	3454.99900000	10.000	34549.9900	63388.2973
60257.1955	3.3805	0.001	3380.46400000	3455.00000000	0.093	320.3288	63708.6261
60259.9943	2.7988	0.001	3378.28100000	3456.08800000	0.004	14.1271	63722.7533
60263.3716	3.3774	0.001	3377.38400000	3456.26800000	0.005	16.7312	63739.4844
60266.7442	3.3726	0.001	3372.56000000	3458.10500000	0.003	10.0441	63749.5285
63226.1481	2959.4039	0.880	3362.95900000	3460.00000000	1.000	3460.0000	67209.5285
63229.5111	3.3630	0.001	3362.95800000	3462.74000000	0.009	30.0000	67239.5285
64406.3283	1176.8173	0.350	3362.33500000	3462.97000000	0.010	34.6297	67274.1582
64409.6907	3.3623	0.001	3362.33400000	3462.97300000	0.020	69.2595	67343.4176
64443.5212	33.8305	0.010	3359.66000000	3463.01200000	0.010	34.6301	67378.0477
65299.6780	856.1568	0.255	3354.40000000	3463.17100000	0.020	69.2634	67447.3112
66660.8586	1361.1806	0.406	3350.00000000	3463.79000000	0.020	69.2758	67516.5870
83400.4386	16739.5800	5.000	3347.91600000	3463.80000000	0.010	34.6380	67551.2250
83435.8086	35.3700	0.011	3347.80500000	3463.87000000	0.010	34.6387	67585.8637
83469.2533	33.4447	0.010	3344.47400000	3464.23000000	0.010	34.6423	67620.5060
975.518 BTC			2903814.907 USD	481756.095 USD			132.202 BTC

Fig. 7. Example of an Orderbook State

The two major components of the environment are the matching engine and the orderbook. The orderbook will play a key role in the environment. Bid, ask, and trade data was collected from the web-socket. Additionally, at the start of data collection, a snapshot of the current orderbook state for the first 20 levels on each side is taken. Based on this snapshot, and the bid and ask data alone, the entire historical orderbook can be created. Fig. 7 depicts an example of an orderbook state.

The data is again split into a training and test set similar to the macro-agent. During training, the environment will start at a random time-step within the training set (at the start of a minute). The environment will last until the end of the minute. Within the minute, the agent will choose to place an order, the matching engine within the environment will attempt to match the order. If for some reason the order has not been matched until the last time-step, the environment will force a market order for all remaining assets. The environment provides reward feedback only when a trade has occurred, or when the time horizon has been exhausted. This single interaction is defined as one epoch.

C. State

As part of the state, the concept of private variables and market variables is used:

- *Private Variables:* Private variables will contain two features: how much quantity is remaining, and how much time is remaining.
- *Market Variables:* Market variables will contain orderbook states and trade data for the past 30 time-steps. An orderbook state contains the bid and ask prices and quantities up to 20 levels. Trade data consists of the

price and quantity at which the last trade occurred, as well as the order side the resulting trade occurred.

D. Action

The action the agent decides is the price at which to place a limit order. More specifically, the agent chooses an integer action $a \in [-50, 50]$, such that the price p_t at time-step t where

$$p_t = p_{m_t} + 0.10a$$

where p_{m_t} is the market price. Notice that there are a total 101 possible discrete actions the agent can choose from.

E. Reward

Recall that the goal is to optimally place limit orders. Therefore, the agent is rewarded accordingly. The reward function is simple: the difference between what the agent was able to sell at with the market price before the order was placed is calculated. The Volume Weighted Average Price (VWAP) is used to determine the aggregated price the agent was able to trade at. Thus, the reward function is

$$R_t = p_{m_t} - \frac{1}{\sum_i v_i} \sum_{p \in P} v_p \cdot p$$

where p_{m_t} is the market price, $\sum_i v_i$ is the total volume of assets, and P is all the prices the agent ordered at.

F. Deep Q-Network Architecture & Training

For this agent, a very similar training framework to the macro-agent is used. However, a different network architecture is used.

Dueling Deep Q-Network

For many states, the actions a on the edge of the orderbook may have little value to these states. Therefore, for these states, it is unnecessary to estimate the action values for these actions. Dueling Deep Q-Networks (DDQNs) provide a solution to this. First applied to Atari games, DDQNs not only help to speed up training, but also yield more reliable Q-values [8].

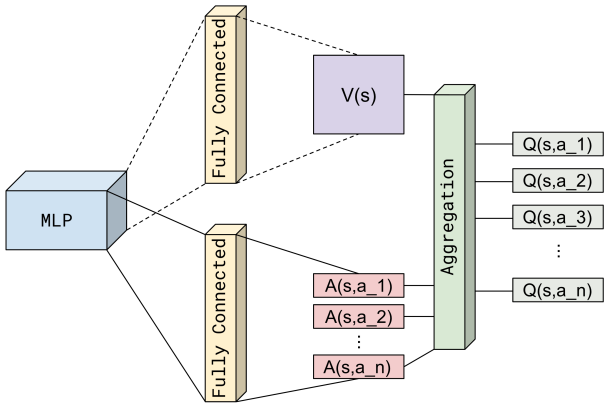

Fig. 8. Dueling Deep Q-Network Framework

Recall that a Q-value can be decomposed into two parts: $V(s)$, the value of being in a particular state s, and $A(s, a)$, the advantage of taking action a in state s, i.e. the Q-value is decomposed as

$$Q(s, a) = V(s) + A(s, a)$$

DDQNs decouple the prediction into two network streams (Fig. 8): value stream, which estimates $V(s)$, and advantage stream, which estimates $A(s, a)$. This decoupling allows the agent to learn which states might not be valuable enough to be worth exploring every action. After this decoupling, there is an aggregation layer which aggregates the two streams by the following:

$$Q(s, a; \theta, \alpha, \beta) = \hat{V}(s; \theta, \beta) + \left(\hat{A}(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} \hat{A}(s, a'; \theta, \alpha) \right)$$

where θ are the weights for the MLP, α are the weights for the advantage stream, and β are the weight for the value stream.

Algorithm 3: Micro-agent Deep Q-Learning Training

Initialize replay memory M
Initialize ϵ, discount factor γ
Initialize Q network with random weights
for *epoch* $= 1$ *to* E **do**
  $s_t \leftarrow$ **Reset** environment (environment outputs random state and gets inputs from macro-agent)
  while $t \leqslant 60$ *seconds* **do**
    $a_t \leftarrow$ **Select** random action with probability ϵ or choose action $\max_a Q(s, a; \theta)$
    $s_{t+1}, r_t, done \leftarrow$ **Act** based on a (environment outputs new state, reward, and done, based on match engine output)
    if *replay memory* M *is full* **then**
      Remove the first element from M
    Append $(s_t, a_t, r_t, s_{t+1}, done)$ to replay memory M
    $b \leftarrow$ **Sample** mini-batch from replay memory M at random
    $q \leftarrow$ **Initialize** empty array of size $|b|$
    foreach $(s_i, a_i, r_i, s_{i+1}) \in b$ **do**
      $q_i = \begin{cases} r_i + \gamma \cdot \max_a Q(s_{i+1}, a_i; \theta) & \text{not } done \\ r_i & done \end{cases}$
    end
    Apply gradient descent with loss $\mathbb{E}_{s,a,r,s'}\left[(q_i - Q(s_i, a_i; \theta))^2\right]$
    Decay ϵ
  end
end

The concept of dueling deep Q-networks is applied to the micro-agent. Besides the dueling concept, the rest of the training framework remains the same (see Fig. 3). The modifications to Algorithm 2 are detailed in Algorithm 3.

G. Results

In this section, the performance of the micro-agent strategy is evaluated. The basic goal behind this agent is for it to optimally place limit orders to buy or sell an asset within the allotted time horizon. It is interesting to see how this agent will perform in its ideal and un-ideal states. In order to evaluate the performance of this agent, two tests are run on two different scenarios:

1) Market Trending Downwards: In this scenario a time-step is chosen within the test set where the market is generally trending downwards is taken. Here, the ideal case is to buy an asset, since the price is generally decreasing.

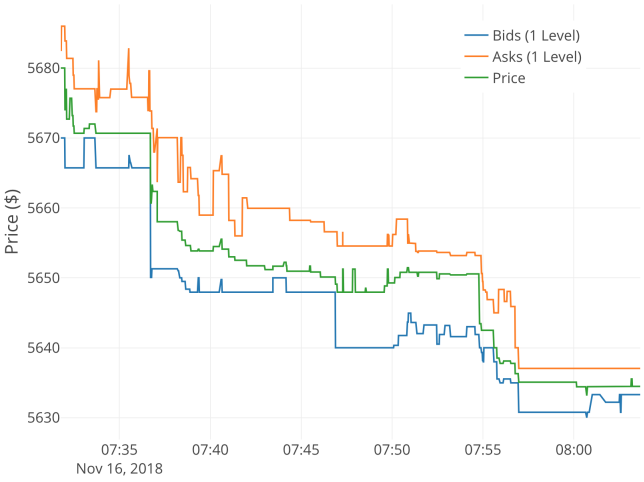

Fig. 9. Orderbook Visualization for Downward Trend

Fig. 9 depicts the snapshot chosen to test the agent on a downward trend. More specifically, the agent was tested on the `07:36` to `07:37` time horizon.

Buying an Asset

When buying an asset on an upward trend, the agent decided to take an action of -11. This meant setting a limit order to buy at $\$1.10$ lower than market price. This resulted in immediate execution. This was the right choice to make, since the market was trending downward.

Selling an Asset

When selling an asset on a downward trend, the agent decided to take an action of 29. This amounted to a limit order price of $\$2.90$ greater than the market price. Although this may not seem like the optimal decision, given that the agent was able to recognize the price drop, it decided to immediately sell off its asset. At such a price it is almost guaranteed for a trade to occur, as was the case here. The agent was able to sell off 1 bitcoin in the first step (before the market dropped).

2) Market Trending Upwards: In this scenario a time-step within the test set is chosen where the market is generally trending upwards. Here, selling an asset would be the ideal case, since the price is generally increasing.

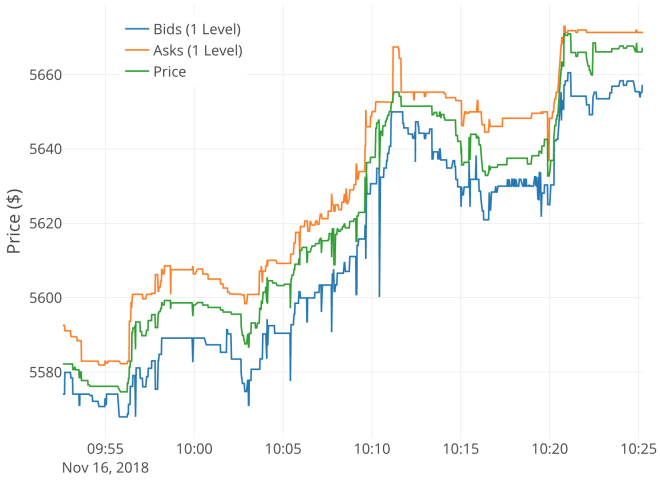

Fig. 10. Orderbook Visualization for Upward Trend

Fig. 10 depicts the snapshot chosen to test the agent on an upward trend. More specifically, agent was tested on the `10:09` to `10:10` time horizon.

Buying an Asset

When buying an asset on an upward trend, the agent decided to take an action of 18. This meant setting a limit order to buy at $\$1.80$ higher than market price. This resulted in immediate execution. This was the right choice to make, since the market was trending upwards.

Selling an Asset

When selling on an upward trend, the agent decided to take an action of -21. This meant setting a limit order to sell at $\$2.10$ lower than market price. This was a slightly risky decision to make since it may not have been matched. In fact, it took two steps for the agent to sell 1 bitcoin, i.e. it had to place 2 limit orders (each partially executed). In this case a negative action was the right choice to make since the price was increasing.

Based on these special cases, the micro-agent seems to be making the right decisions. When the price is expected to drop, the agent makes the right decision to sell off all assets immediately, or limit buy assets at a lower price than the market on the bid side. On the other hand, when the price is expected to grow, the agent makes the right decision to buy assets immediately, or sell assets at a lower price than the market on the ask side.

VI. Combining the Macro and Micro Agents

In Sections 4 and 5, an in depth overview of the frameworks for both the Macro and the Micro agents was given. Now, the two agents are combined to reflect the multi-agent framework defined in Fig. 3. The end-to-end pipeline contains 4 components, depicted in Fig. 11, which is briefly detailed:

1) *Web-socket:* Raw trade, bid, and ask data arrives through a web-socket that is connected to the Bittrex

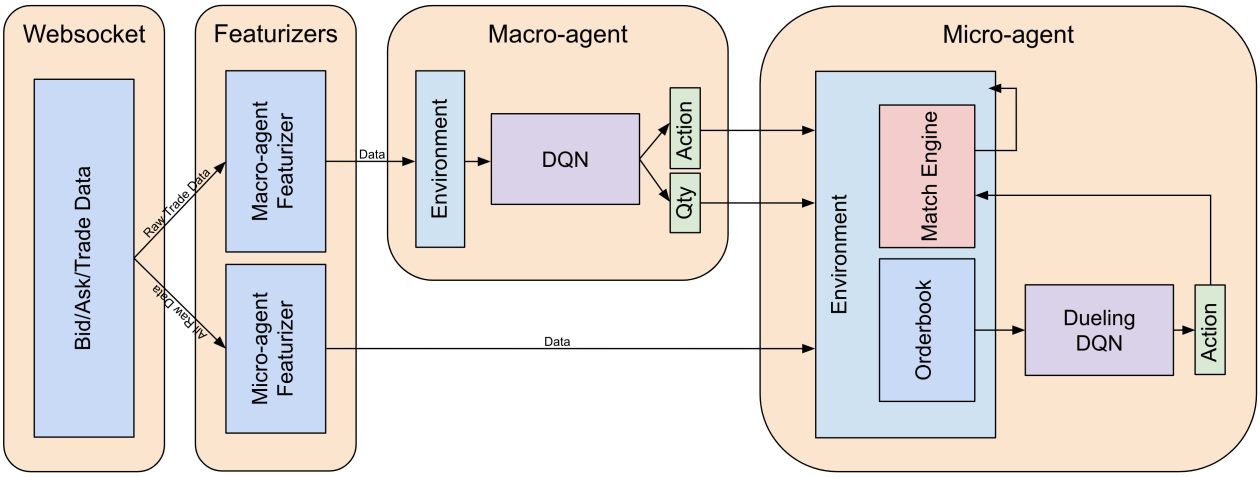

Fig. 11. Combined Multi-agent End-to-End Pipeline

exchange. As soon as data arrives, it is sent to the featurizers.

2) *Featurizers:* There are two featurizers, one for each agent. The Macro-agent featurizer computes the market indicator features and the minute data based on the raw trade data collected from the web-socket. The Micro-agent featurizer determines the new state of the orderbook based on all the raw data.
3) *Macro-agent:* As detailed in Section 4, the macro-agent determines the action to buy, sell or hold the asset. Notice that a quantity is also sent. The macro-agent keeps a running count of how many assets it currently has. If the decision to sell is chosen, then this count (total number of assets) is sent. If the decision to buy is chosen, then a quantity of 1 is sent.
4) *Micro-agent:* As detailed in Section 5, the micro-agent determines the action of where to place the order in the limit book. The side of the orderbook and the quantity is determined by the data provided by the macro-agent. Once the micro-agent has decided an action, this action gets fed into the match engine, which attempts to match the order. This decision is fed back into the environment, for the agent to decide to cancel the order and create a new order. If the order turns into a trade, then the orderbook is updated. Note that in practice the match engine is the actual exchange.

A. Final Results

The end-to-end pipeline is evaluated using the test set date range defined in Section 4 for the macro-agent. Similar to the evaluation of the macro-agent, the multi-agent strategy is compared to the Buy and Hold investing and Momentum investing strategies.

In Fig. 12 the performance comparisons between the Buy and Hold investment, Momentum investment, Macro-agent, and Multi-agent (Macro-agent and Micro-agent) strategies are given. It is interesting to see that the multi-agent approach under-performed in comparison to the macro-agent. In the previous evaluation of the micro-agent, it was noticed that

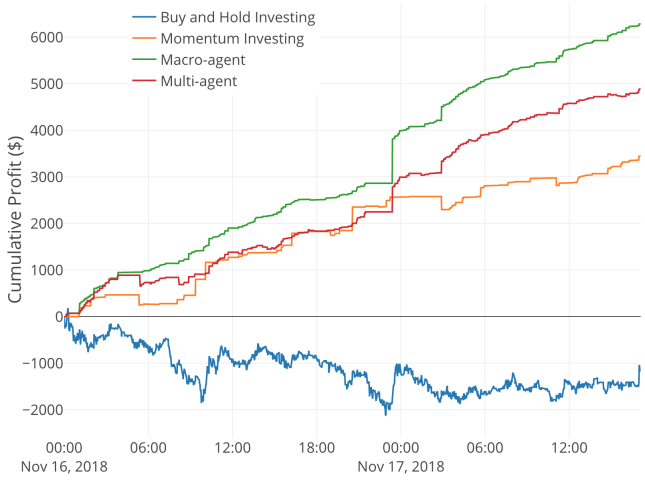

Fig. 12. Performances of Various Investment Strategies PNL-Realized Graph (including Multi-agent)

sometimes the agent would decide that the right decision was to place an order towards the opposing side of the orderbook, i.e. a price slightly worse than market price. Although this resulted in immediate execution, this may have been responsible for the drops in profit.

In terms of total orders placed by the micro-agent in the multi-agent setting, 91% of all orders placed were limit orders. Thus, the micro-agent was able to optimize placing limit orders well. Note that in the other strategies hypothetical prices were assumed for which market orders would have been placed for. This would also result in additional exchange fees.

Overall, the proposed multi-agent framework is able to optimize well. Additionally, the multi-agent approach is also a stable strategy since the profit accumulation is not volatile.

VII. Future Work

Although considerable progress was made in creating an end-to-end framework for optimizing market making, there are still some holes that need to be filled:

- *Risk:* One assumption made throughout this problem was that the agent was allowed to buy an unlimited number of assets. When testing the multi-agent strategy, it was found that there were cases when the agent was selling 84 bitcoins, with relatively low profit. The risk associated with this assumption is very high. The next step for this approach would be optimizing on this by adding constraints that limit the number of buy actions an agent takes.
- *Reward Engineering:* Reward engineering is an important problem in reinforcement learning. In the case of the macro-agent, it was found that the decision to hold an asset was very sparse. This may have resulted from the agent receiving a reward of 0 for the decision to hold. Perhaps setting the constraints as previously mentioned, with a better reward function may help the agent learn to hold assets for a longer time. Furthermore, incorporating exchange fees in the reward function will lead to an agent that can successfully interact with the actual exchange environment.
- *Market Simulator:* Although the simulator used in this framework did provide a decent reflection of the true market environment, it still lacked many variables, specifically the effect of other traders. The assumption that the agent was the only trader interacting with the orderbook was also made. One option here is to create a generative model that can make synthetic orders that simulate various traders.
- *Corrupted Data:* Another concern is corrupted data. On multiple instances, it was found that data would arrive out of order, or only partial (missing) data was received due to network issues. In practice, it is critical that the states match the actual exchange. Therefore, a solution to this is needed before this framework can be deployed on an actual exchange.

Overall, there are multiple assumptions that were made in this framework. The goal is to now eliminate these assumptions to provide a more robust framework.

VIII. Conclusion

A multi-agent reinforcement learning framework is provided to solve the problem of optimizing market making. The results show that the policy these agents were able to learn led to a stable trading strategy, resulting in a low-volatile linear growth in profit. Applying reinforcement learning to such a problem also shows us that reinforcement learning has the ability to perform well in complex environments, and is a viable concept that can be applicable to market making.

References

[1] Matthew F Dixon. "A High Frequency Trade Execution Model for Supervised Learning." arXiv preprint arXiv:1710.03870 (2017).

[2] Prakhar Ganesh, Puneet Rakheja. "Deep Reinforcement Learning in High Frequency Trading." arXiv preprint arXiv:1809.01506 (2018).

[3] Yuriy Nevmyvaka, Yi Feng, Michael Kearns "Reinforcement Learning for Optimized Trade Execution." https://www.cis.upenn.edu/ mkearns/papers/rlexec.pdf (2006).

[4] Marc Juchli "Limit order placement optimization with Deep Reinforcement Learning" https://repository.tudelft.nl/islandora/object/uuid:e2e99579-541b-4b5a-8cbb-36ea17a4a93a?collection=education (2018).

[5] "Use Z-Scores To Maximize Your Portfolio's Potential" https://seekingalpha.com/article/3224666-use-z-scores-to-maximize-your-portfolios-potential (2015).

[6] Hado van Hasselt, et al. "Learning values across many orders of magnitude" arXiv preprint arXiv:1602.07714 (2016).

[7] Gavin Chan. "Light Matching Engine" https://github.com/gavincyi/LightMatchingEngine (2017).

[8] Ziyu Wang, et al. "Dueling Network Architectures for Deep Reinforcement Learning" arXiv preprint arXiv:1511.06581 (2016).

[9] Jae Won Lee, Jangmin O. "A Multi-agent Q-learning Framework for Optimizing Stock Trading Systems" ftp://ftp.cse.buffalo.edu/users/azhang/disc/springer/0558/papers/2453/24530153.pdf (2002).

[10] Investopedia "Random Walk Theory" https://www.investopedia.com/terms/r/randomwalktheory.asp