Imperial College London

Imperial College London

Department of Mathematics

Deep Reinforcement Learning and Electronic Market Making

Author: Chenyu Liu (CID: 01203579)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2019-2020

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

I would like to thank my supervisor Dr Paul A. Bilokon for helping me with this thesis.

Abstract

Market making is a fundamental problem of modern algorithmic trading, where the market maker has to continuously provide liquidity to the market by offering buy and sell prices. The main challenge of market making is to manage the inventory risk. Traditional market making formulae rely on assumptions of the market, including modelling the arrival of orders with stochastic processes. In this paper, I proposed a model-free approach with Reinforcement Learning, using real-world crypto-currency data. A wide range of deep reinforcement learning algorithms are tested, and we achieved a positive average profit over the data set, and also outperformed benchmarks.

Contents

Abstrct **1**

1 Introduction and Background **6**

1.1 Background of Deep Reinforcement Learning 7

1.1.1 Markov Decision Process 7

1.1.2 Bellman Equation 10

1.1.3 Dynamic Programming 11

1.1.4 Monte Carlo Methods 13

1.1.5 Exploration vs Exploitation 13

1.1.6 Temporal Difference Prediction 14

1.1.7 Tabular Q-Learning 14

1.1.8 Deep Q-Learning 15

1.1.9 Extensions to Deep Reinforcement Learning 16

1.1.10 Policy Gradient 17

1.1.11 Actor-Critic Method 18

1.1.12 Continuous Action Spaces 19

1.2 Background of Electronic Market Making 20

1.2.1 Electronic Markets and the Limit Order Book 20

1.2.2 Grossman-Miller Market Making Model 21

1.2.3 The Avellaneda-Stoikov Model 23

2 Applications **27**

2.1 Data and Environment 27

2.1.1 Data Source 27

2.1.2 Environment Steps 27

2.1.3 Reward Function 28

2.2 Observation Space 29

2.2.1 Market State Observation 29

2.2.2 LSTM observations 30
2.2.3 Trading Agent Observation 31
2.2.4 Observation memory 32
2.3 Action Space 33
2.4 Agent 34
2.5 Computing Speed Analysis 35
2.6 Training process and hyper-parameter tuning 36
2.6.1 Hyper-parameters 36
2.6.2 Training Process 38
2.7 Performance 44
2.8 Further Development 46

A Source Codes **47**

Bibliography **78**

List of Figures

2.1 The LSTM-predicted probability distribution of midprice movement for 100 random states . . . 30
2.2 The Architecture of the entire agent . . . 35
2.3 The PnL, Position, Reward and Market Price of an Environment, at early stage of training . . . 38
2.4 The Actions of the Agent in this Environment, at early stage of training . . . 38
2.5 The PnL, Position, Reward and Market Price of an Environment, at mid stage of training . . . 38
2.6 The Actions of the Agent in this Environment, at mid stage of training . . . 39
2.7 The PnL, Position, Reward and Market Price of an Environment, at mid-to-late stage of training . . . 39
2.8 The Actions of the Agent in this Environment, at mid-to-late stage of training . . . 39
2.9 The PnL, Position, Reward and Market Price of an Environment, at late stage of training . . . 39
2.10 The Actions of the Agent in this Environment, at late stage of training . . . 40
2.11 The Maximum of Gradient of Neural Network . . . 40
2.12 The Variance of Gradient of Neural Network . . . 40
2.13 The L2-norm of Gradient of Neural Network . . . 41
2.14 The Advantage Function . . . 41
2.15 The Predicted Values of States . . . 41
2.16 The average Rewards of 16 environments . . . 42
2.17 The Entropy Loss . . . 42
2.18 The Value Loss . . . 43
2.19 The Policy Loss . . . 43
2.20 The Total Loss . . . 43
2.21 The Distribution of PnL . . . 44
2.22 The Distribution of PnL, Given less than 10 . . . 44

2.23 The Distribution of PnL, Given more than 10 44
2.24 The Distribution of PnL, With random action 45

Chapter 1

Introduction and Background

Reinforcement learning is a rapidly developing subfield of machine learning, which focuses on training an agent to participate in a complicated environment, make observations, take optimal actions based on these observations and gain a maximal reward. The environment is usually a very large space, and the agent can only navigate a tiny proportion of it, which is very different from other machine learning problems. The environment may also have non-static input-output relation, but rather a dynamic and stochastic one. Despite the difficulties, reinforcement learning has seen repeated successes in some problems like Atari 2600 games, GO, and DOTA. This success is drawing the attention of both the general public and the researchers to this field.

In this chapter we will first introduce some basic concepts of deep reinforcement learning, as well as some extensions and modifications to the original basic setup that will be used in our code. All these variants of the algorithms are tested in our environment, but not all of them perform well or even converge at all. They are included in the thesis to show how the theory of reinforcement learning is developed over the past few years, and to show how certain algorithms solve the market making environment better than the others.

Some theoretical works on electronic market making will also be reviewed. These equations are based on the assumption of market orders and mathematical deductions instead of real market data and computer-based tests, but they give a very good idea of how a good market maker should perform.

1.1 Background of Deep Reinforcement Learning

1.1.1 Markov Decision Process

The foundation of reinforcement learning is the **Markov Decision Process (MDP)**. However, before talking about it, we must formalize the simplest child of Markov family: the **discrete Markov Process**.

In a discrete Markov Process, the system has many different **states**, which can be observed. The state this system is in changes after each time step, and forms a sequence of states, which is called **Markov Chain**. However, the state has a restriction on how it evolves, which is a **Markov Property**: The future dynamics of the system must depend only on the current state, not the history of the evolution of the states. In the case that the state space is finite, we may write down a **Transition Matrix**. In this matrix, the element in i^{th} row and j^{th} column, T_{ij}, represents the probability of transition from state i to state j.

The next member in the Markov family is the **Markov Reward Process**. On top of the Markov Chain setup, after each state-to-state transition, a (random) reward is given to the observer. In this case, instead of the transition matrix, we need a tensor to record this:

$$\mathbb{P}(s', r|s) = \text{Probability of transition to state s' from state s, while receiving reward r} \tag{1.1.1}$$

In some cases, we may be unable to observe the state directly. Instead we observe an observation dependent on the state. This leads to an **Observation Space**, and **Observation Matrix** or **Emission Matrix**:

$$\mathbb{P}(o|s) = \text{Observe o while the system is in state s} \tag{1.1.2}$$

This is called **Partially Observable Markov Process**, or **Hidden Markov Model**. There is a lot of active research on HMMs, but we will now talk about them is this thesis.

Finally, we take into account the actions of the agent in this environment, to create a mathematical formulation of **environment, observation, agent, action** and **rewards**.

In a Markov Decision Process:

- States are not directly observable, but some observations based on the state are known. The observation space and state space may or may not have an explicit bijection between them.
- An agent is taking actions based on the observations, and the action changes the underlying state of the environment, returning a different observation and a reward.

- How state transits depends on the last state of the system and the action of the agent, usually in a probabilistic way.
- Given a state-action pair, the environment returns a possibly random reward to the agent.
- The agent's goal is often to maximise the total reward received, with future rewards discounted by a factor.

To formalize this process, we introduce the following symbols and notations:

- (Implicit) Time: t
- State : S_t
- Action : a_t
- Next state from state S_t following action a_t: S_{t+1}, with probability $\mathbb{P}(S_{t+1}|S_t, a_t)$
- Reward given to the agent while it transits to S_{t+1}: r_t, with probability $\mathbb{P}(S_{t+1}, r_t|S_t, a_t)$
- Policy of the agent: Probability of taking action a_t from state S_t: $\pi(S_t, a_t)$
- The objective is to maximise discounted return: $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$, including the possibility of $T = \inf$ or $\gamma = 1$ but not both. The discounted total reward instead of the direct next reward is maximised, such that the agent can learn to sacrifice the short-term reward for a bigger long-term benefit.

The dynamics of **MDP** is described by the probability of each possible pair of reward r and state s', from state s, after action a:

$$\begin{aligned} p(s', r|s, a) &= \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \\ &= \text{Probability of transition to state s' with reward r, from state s, after action a} \end{aligned} \tag{1.1.3}$$

We can now define a policy:

$$\pi(a|s) = Pr(A_t = a|S_t = s) \tag{1.1.4}$$

Given a policy, we can define the value of state:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s] \tag{1.1.5}$$

Notice that this value of state is dependent on the policy. It should not be considered as an intrinsic property of the state.

Similarly, for a state-action pair, we can define its Q-value:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a] \tag{1.1.6}$$

We can also reverse this relation. Starting from a value function $v(s)$, or a state-action function $q(s,a)$, we can directly obtain an optimal policy according to these functions:

$$\pi_v(s) = \arg\max_a \sum_{s'} p(s'|s,a)v(s') \tag{1.1.7}$$

$$\pi_q(s) = \arg\max_a q(s,a) \tag{1.1.8}$$

We call any method that tries to learn the p-function **P-Learning**, and any method that tries to learn to q-function **Q-Learning**

1.1.2 Bellman Equation

A useful property of the value functions is the recursive property:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{1.1.9}$$

This is called **Bellman Equation**. Starting from this equation, if we consider an **Optimal State-value Function**, denoted v_*, which is defined by:

$$v_*(s) = \max_\pi v_\pi(s) \tag{1.1.10}$$

and similarly, **Optimal Action-value Function** $q_*(s, a)$:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{1.1.11}$$

Notice that v_* is an optimal value function and its consistency condition can be written without reference to any specific policy:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \sum_{k=0}^{\infty} \gamma^{k+1} r_{t+k+2} | S_t = s] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{1.1.12}$$

This equation is called the **Bellman optimality equation**. For an environment with finite states and known dynamics, this linear system of equations may be solved with a variety of methods. However, in real world problems, this approach has two major limitations. Firstly, the number of states is often too large or even infinite for the system to be solved directly. Even a small system with roughly 100 states could be computationally too expensive to solve. Another issue is that the dynamics of the system is also usually not known. Without knowing the exact probabilities, it is impossible to solve the equations and find a stable policy.

1.1.3 Dynamic Programming

Dynamic Programming is a collection of algorithms used to compute an optimal policy in a MDP. The key idea of DP is to use the recursive property of Bellman optimality equation for P-values and Q-values, to organize the search of optimal policy.

Policy Evaluation

Given the recursive property, we can turn the **Bellman Equations** into an update rule. Consider a sequence of approximate value functions $v_0, v_1, v_2, \ldots$, with the successive update rule:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned} \tag{1.1.13}$$

This is called **Iterative Policy Evaluation**. Given any policy π, we can start with any value functions and apply the updates, and the sequence of value functions should converge to the value function corresponding to this policy. It is possible to prove this sequence will converge to the "correct" value function associated with this policy.

The convergence may be slow, but we can often accept a close approximation of the "correct" value function. We can track the maximum difference in the value of states after each step of update, and terminate the process once the maximum difference is smaller than a threshold.

Policy Improvement

We will first introduce the definition of "a better policy".

Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{1.1.14}$$

Then the policy π' must be better or as good as policy π, which gives greater expected rewards for all states s:

$$v_{\pi'}(s) \geq v_\pi(s) \tag{1.1.15}$$

This is called the **Policy Improvement Theorem**. For example, we can easily prove that, from an old policy π, if we compute the q values of this policy, and build a new greedy policy:

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{1.1.16}$$

This greedy policy satisfies the condition of the Policy Improvement Theorem. This tells us that the greedy policy is an improvement on the original one.

Policy Iteration

Value iteration effectively combines policy evaluation and policy improvement. Starting from any simple policy, apply policy evaluation to obtain the state value function, and then apply policy improvement to obtain a better policy. By repeating these two steps, our policy should converge towards the optimal one.

$$\begin{aligned}
\pi_0 &\xrightarrow{\text{Policy Evaluation}} v_{\pi_0} \\
v_{\pi_0} &\xrightarrow{\text{Value Iteration}} \pi_1 \\
\pi_1 &\xrightarrow{\text{Policy Evaluation}} v_{\pi_2} \\
\ldots &
\end{aligned} \tag{1.1.17}$$

Value Iteration

In policy iteration, each of its iterations involves policy evaluation, which is also an iterative process that converges slowly. However, it is unnecessary to complete the policy evaluation process. We may only take a few steps of policy evaluation and move on to the next step of value iteration, without waiting for a full convergence.

1.1.4 Monte Carlo Methods

The policy evaluation process in the last section involves considering all possible state-action-reward tuples, which is impractical in an environment with a huge state space. The idea of the **Monte Carlo Method** is, given a policy π, generate a number of full episodes. Only for each state-action pair in these generated episodes, update the values according to discounted rewards from the full episode. This method is very useful when we have little knowledge about the dynamics of the environment.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{1.1.18}$$

where G_t is the actual discounted return from an episode. There are also two versions of Monte Carlo Methods, the **First-Visit Monte Carlo** and the Every-Visit Monte Carlo. After generating a full episode of game, in some cases, we may see the same (s, a, r, s') tuple appears multiple times. The First-Visit Monte Carlo believes that only the first one should be used for learning, while the Every-Visit Monte Carlo methods uses all of them. Empirically,the performance of First-Visit Monte Carlo is slightly better, but the difference is minor.

1.1.5 Exploration vs Exploitation

In the Monte Carlo Methods, we often need to generate an entire episode of game. However, if we start from a very bad policy, or if we run into a policy that incorrectly favours certain actions, we may never get to try and understand how some other actions perform. In other words, it is crucial to let the agent explore a large part of the policy space, before converging to an optimal one. The most basic method is call **ϵ-greedy action selection**.

In this method, whenever we select an action, there is a probability of ϵ that this selection is completely random, regardless of what our current policy/model outputs. The optimal action based on our current learning is selected with only the probability of $1 - \epsilon$. This parameter of ϵ is usually decreased over time, start at 1 at the beginning of the training, such that the agent can fully explore different actions. As the ϵ slowly decreases to 5% or even to 0, the agent can explore some variations base on the current policy.

Other Exploration vs Exploitation techniques include Upper Confidence Bound etc., but they are not as useful in the context of a complicated reinforcement learning problem.

1.1.6 Temporal Difference Prediction

In the Monte Carlo method, notice that the update is carried out only at the end of a full episode. This may give rise to a large variance if the length of the episode is large.
In TD(0) methods, we reuse the existing knowledge of the value of the next time step:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{1.1.19}$$

In effect, the target of update for the Monte Carlo method is G_t, while the target for the TD(0) method is $R_{t+1} + \gamma V(S_{t+1})$. Studies show that switching to the TD(0) method usually improves performance.
If we apply TD(0) to the Q-Learning, we get the **One-Step Q-Learning** algorithm:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{1.1.20}$$

We can easily extend this to **n-step Q-Learning**, as an extension to the original Deep Q-Learning in the later chapters.

1.1.7 Tabular Q-Learning

We can apply the value iteration technique to Q-learning by tracking the value of all state-action pairs:

1. Start with an empty table of $Q(s, a)$ for all pairs of (s, a)
2. Obtain (s, a, r, s') from the environment. Any Exploration vs Exploitation technique may be used here.
3. Make a "blending" update according to the Bellman equation:
   $Q_{s,a} \leftarrow (1 - \alpha)Q_{s,a} + \alpha(r + \gamma(\max_{a' \in A} Q_{s',a'}))$
4. Check convergence condition and repeat from step 2.

1.1.8 Deep Q-Learning

The tabular Q-learning method iterates over the full set of states, while this is impossible in some cases. The state space could be far too large or even continuous. The idea of deep Q-learning is to build a neural network to approximate $Q(s, a) \approx Q(s, a, \theta)$. The simplest form of deep Q-learning is like this:

1. Initialize a neural network to approximate $Q(s, a)$. The dimension of input and output of the neural network should be the same as the dimension of state space and size of action space.
2. Interact with the environment and obtain tuple (s,a,r,s').
3. Calculate the loss: $\mathcal{L} = (Q_{s,a} - r)^2$ if episode ends, and $\mathcal{L} = (Q_{s,a} - (r + \gamma \max_{a' \in A} Q_{s',a'}))^2$ if the episode has not ended. This is the same idea as in the TD(0) method.
4. Update $Q(s, a, \theta)$ using stochastic gradient descent
5. Repeat from step 2 until converged

However, this simple algorithm does not work very well. One reason is that we don't have a clever way to interact with the environment. If we take actions randomly, the chance of ever winning a game and receiving the rewards is very small, thus the program learns very slowly. To solve that, we apply the **epsilon-greedy** method: With probability ϵ, we act randomly. With probability $1 - \epsilon$, we follow the greedy action. This parameter ϵ decreases from 1.0 to 0.01 over the course of learning.

Two other modifications, Replay Buffer and Target Network, are mentioned in the paper **Playing Atari with Deep Reinforcement Learning**[1] and they both are aiming at increasing learning stability. The final version of the algorithm in that paper has the following steps:

1. Initialize neural networks $Q(s, a, \theta)$ and $\hat{Q}(s, a, \hat{\theta})$ randomly. Set $\epsilon = 1.0$. Build an empty buffer.
2. With probability ϵ, select a random action a. Otherwise, select greedy action $a = \arg\max_a Q_{s,a}$.
3. Execute action a and observe reward r and next state s'.
4. Store (s, a, r, s') into the buffer.
5. Sample a random minibatch from the buffer. Calculate target $y = r$ if episode ends, and $y = r + \gamma \max_{a' \in A} \hat{Q}_{s',a'}$ otherwise.
6. Calculate loss: $\mathcal{L} = (Q_{s,a} - y)^2$. Update $Q_{s,a}$ using Stochastic Gradient Descent.
7. Repeat from step 2. For every N steps, copy weights from Q to $\hat{Q}$.

1.1.9 Extensions to Deep Reinforcement Learning

In the paper **Rainbow: Combining Improvements in Deep Reinforcement Learning**[2], a few improvements of the DQN are combined into a hybrid method. In this section, we will briefly introduce these improvements.

N-step DQN

The first improvement was first introduced in the paper **Learning to Predict by the Methods of Temporal Difference**[3]. From the Bellman Equation, we expand the recursive relation for more steps, so the estimation of Q is more stable. For example, a 2-step DQN would have the following update rule:

$$Q(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a') \tag{1.1.21}$$

This requires saving the experience in the form of $(s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$, instead of the original (s_t, a_t, r_t, s_{t+1}) tuple. Rolling out more steps means using more memory and calculation time, but the experiments show that this method gives minor improvement when the step number is more than 2. This is why the 2-step DQN is the most favourable one in most cases.

Double DQN

In the paper **Deep Reinforcement Learning with Double Q-Learning**[4], authors found a methods to fix the overestimation of DQN. In the basic DQN, the target value for Q is:

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_{a} Q(s_{t+1}, a_{t+1}) \tag{1.1.22}$$

This is changed to :

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_{a} Q'(s_{t+1}, \arg\max_{a} Q(s_{t+1}, a)) \tag{1.1.23}$$

The Q' is called Target Network. We can see that the actual action for the next state is chosen by the trained network, but the values used to estimate and update the parameters are taken from the target network. The target network doesn't get updated every time. Instead, its parameters are copied from the original network once every fixed number of steps, usually 100.

Noisy Networks

The next improvement is about explorations of the environment. Instead of controlling the ϵ to explore, in the paper **Noisy Networks for exploration**[5], noise is added to the weights of the fully-connected layers of the network by adding a normally distributed noise to the output. The

size of the noise, i.e. the standard deviation of the normal distribution, is also adjusted using back-propagation. Over the course of learning, if the Q-value for some state-action pair is stable, the noise is automatically reduced, similar to a reduced ϵ value.

Prioritized replay buffer

In the paper **Prioritized Experience Replay**[6], a new method is introduced to increase learning speed. After every interaction with the environment, the experience s, a, r, s' is stored into a replay buffer, together with a default weight of usually 1. Before adjusting the parameter of the network, a batch of experiences are sampled from the buffer, according to its weight. After calculating the losses of the network and the back propagation, the weights of these experiences are updated according to the training loss. Experience tuples with high losses have a higher weight or priority; thus it is more likely to be selected and learned again next time.

Dueling DQN

In the paper **Dueling Network Architectures for Deep Reinforcement Learning**[7], the value of $Q_{s,a}$ is divided into two parts: $V(s)$ and $A(s, a)$, each approximated by a neural network.. The $V(s)$ is the value function of the state, and $A(s, a)$ aims to capture how much extra value an action can bring to us. The $A(s, a)$ is set to have an average of 0 over the action space, so the final approximation of Q value is $Q(s, a) = V(s) + A(s, a) - \bar{A}(s)$, where $\bar{A}(s)$ is the average $A(s)$ over all actions.

Categorical DQN

Finally, the most complicated improvement is proposed in the paper **A Distributional Perspective on Reinforcement Learning**[8]. Instead of estimating a single Q-value for each state-action pair, this methods is trying to predict the distribution of discounted reward. The Bellman Equation now takes a distributional form $Z(x, a) \overset{D}{=} R(x, a) + \gamma Z(x', a')$. The loss is calculated by Kullback-Leibler divergence.

1.1.10 Policy Gradient

Instead of training a neural network to learn the Q-values and than take action based on the largest Q-value, it is more straightforward to train a neural network to directly return a policy in the form of probabilities. Suppose there are n actions, we build a network with n outputs, apply a softmax function on the outputs to turn them into a probabilistic policy, and finally train it with Cross

Entropy Loss, which is also called **Policy Gradient**:

$$Loss_{PolicyGradient} = -Q(s, a) * log(\pi(a|s)) \tag{1.1.24}$$

An algorithm called **REINFORCE** is developed based on this policy gradient:

1. Initialize the network randomly
2. Play N full episodes and save these experiences
3. Calculate the discounted total rewards for each state-action pairs in the experiences
4. Calculate the loss $\mathcal{L} = -\sum_{k,t} Q_{k,t} log\pi(s_{k,t}, a_{k,t})$
5. Perform SGD to update weights
6. Repeat from step 2

1.1.11 Actor-Critic Method

One way to improve the stability of PG is to reduce the variance. Suppose our rewards are always positive, but only differ in magnitude. While applying the policy gradient, a positive reward means we are pushing our agent to take all actions more often, instead of encouraging some actions and discouraging the others. Thus, we train a separate network to learn the average values of the states, and subtract it from the reward. This means total reward is called baseline. The policy part of the network is called *actor*, and the value part of the network is called *critic*. The algorithm looks like this:

1. Initialize network with random parameters
2. Play N steps in the environment using the current policy π_θ, and save these experience.
3. Calculate the discounted accumulated reward.
4. Perform Policy Gradient update $\partial\theta_\pi \leftarrow \partial\theta_\pi + \nabla_\theta log\pi_\theta(a_i|s_i)(R - V_\theta(s_i))$
5. Perform Value update $\partial\theta_v \leftarrow \partial\theta_v + \frac{\partial(R-V_\theta(s_i))^2}{\partial\theta_v}$
6. Repeat from step 2

1.1.12 Continuous Action Spaces

In some cases, the action space could be continuous. This could be an angle of the robot arm, a force applied to an object or a direction the agent is moving. The usual methods cannot deal with a continuous action space.

Take A2C as an example, we will talk about how to change it to learn a continuous action space. Suppose there are N continuous actions, we build a network that outputs:

- value: Dimension = 1
- μ: Dimension = N
- σ^2: Dimension = N

The value part again serves as a baseline, and is trained by Mean Squared Error Loss. The action the agent takes is now a N-dimensional normally-distributed variable, with mean μ and variance σ^2. More precisely:

$$log\pi_\theta(a|s) = -\frac{(x-\mu)^2}{2\sigma^2} - log\sqrt{2\pi\sigma^2} \tag{1.1.25}$$

And the entropy for such a policy, $ln\sqrt{2\pi e\sigma^2}$, is added to the final loss function.

1.2 Background of Electronic Market Making

1.2.1 Electronic Markets and the Limit Order Book

Most of the financial contracts are now traded in electronic markets today. From stocks, FX, to bonds and derivatives, everything that is traded regularly with a large volume is now traded on electronic markets, given how fast and accurate it is. In the most basic setup, an electronic market has two types of orders: Market Orders and **Limit Orders**. A Market Order means a participant of the market wants the order executed immediately, at the best price this market can offer at this moment. This is a very aggressive order, in contrast to a Limit Order, which is considered more passive. The Market Order only includes the quantity that someone wants to trade, but the Limit Order also includes the worst price for this trade. Suppose someone is trying to buy a certain quantity of security, a Limit Order will include the maximum price he is willing to pay for this trade. For a sell order, this would be a minimum price. Such a Limit Order will not be immediately executed.

Limit Orders are organized in a **Limit Order Book**. Ths LOB keeps track of all ongoing orders. It records the accumulated quantity of Limit Orders at any price, and broadcast this data to all market participants. The system then executes all possible trades with a defined algorithm. This algorithm usually prioritizes Market Order and then Limit Orders, in a price-time priority. Market Order is fullfilled by matching its quantity with the best price for this quantity on the market, and earlier Limit Orders at these prices will be executed first. Limit Orders that don't immediately match anything will be kept on the LOB, to wait for further market price movements.

This is not the only setup for an Electronic Market. Some markets use a **pro-rata** rule, which means Market Orders are shared among the Limit Orders at the same price, proportional to their volume, instead of on a first-come-first-serve basis. Some markets even mix up the time-priority and pro-rata rules.

Another difference between electronic markets is transparency. Large exchanges tend to be regulated, and the information about the orders are shared openly to all market participants. However, in some other exchanges, these information might not be all publicly available. Certain exchanges make a profit by selling information of the order flow.

However, the advantage of these smaller exchanges is that they provide a wider range of types of orders, for example, Hidden Order, Iceberg Order, Immediate-or-Cancel Order, Fill-or-Kill Order. Exchanges also charge different prices for transactions and a direct feed of data.

1.2.2 Grossman-Miller Market Making Model

This is the model[9] provided by Frossman and Miller in 1988 to capture the behaviour of market makers. We will look at a rephrased version[10] of it.

Consider n identical Market Makers for a single asset and three dates $t \in \{1, 2, 3\}$. At date 1, a liquidity trader, LT1, sells i units of the asset at time 1, and another liquidity trader, LT2, buys i units of the asset at time 2. The prices at time 3 is $S_3 = \mu + \epsilon_2 + \epsilon_3$, where μ is constant, ϵ_2 and ϵ_3 are independent normally distributed variables with mean 0 and variance σ^2, representing the price change between time 1 and 2 and price change between time 2 and 3. Assume all market participants and risk-averse and have utility function $U(X) = -exp(-\gamma X)$.

At time 3, everything is settled in cash and all market participants should leave with 0 units of asset. However, we want to know how much assets everyone is holding at time 2. Assume that at time 2, the n MMs and the LT1 hold q_1^{MM} and q_1^{LT1} units of asset respectively.

We can now formulate the problem as a optimisation process: Agent j will choose q_2^j to maximise his expected utility given ϵ_2 is realised and made public before t=2:

$$max_{q_2^j} \mathbb{E}[U(X_3^j)|\epsilon_2] \tag{1.2.1}$$

subject to $X_3^j = X_2^j + q_2^j S_3$ and $X_2^j + q_2^j S_2 = X_1^j + q_1^j S_2$ This problem is concave and it has a solution of:

$$q_2^j = \frac{\mathbb{E}[S_3|\epsilon_2] - S2]}{\gamma\sigma^2} \tag{1.2.2}$$

for all agents.

From our assumptions on S_3, we can show that:

$$\mathbb{E}[U(X_3^j)|\epsilon_2] = -exp(-\gamma(X_2^j + q_2^j \mathbb{E}[S_3|\epsilon_2]) + \frac{1}{2}\gamma^2 (q_2^j)^2 \sigma^2) \tag{1.2.3}$$

Thus the solution of the optimisation problem shoudl be:

$$q_2^j = \frac{\mathbb{E}[S_3|\epsilon_2] - S_2}{\gamma\sigma^2} \tag{1.2.4}$$

We can now start to solve for equilibrium price S2.

$$0 = i + q_1^{LT2} = nq_1^{MM} + q_1^{LT1} + q_1^{LT2} = nq_2^{MM} + q_2^{LT1} + q_2^{LT2} = (n+2)q_2 \tag{1.2.5}$$

Thus $S_2 = \mu + \epsilon_2$.

Consider now what happens at time 1. We can solve a optimisation problem similarly:

$$max_q \mathbb{E}[U(X_2^j)] \tag{1.2.6}$$

subject to $X_2^j = X_1^j + q_1^j S_1$ and $X_1^j + q_1^j S_1 = X_0^j + q_0^j S_1$.

The solution to this is also similar:

$$q_1^j = \frac{\mathbb{E}[S_2] - S_1}{\gamma\sigma^2} \tag{1.2.7}$$

and the equilibrium price S_1 is:

$$i = nq_0^{MM} + q_0^{LM1} = nq_1^{MM} + q_1^{LT1} = (n+1)\frac{\mu - S_1}{\gamma\sigma^2} \iff S_1 = \mu - \gamma\sigma^2 \frac{i}{n+1} \tag{1.2.8}$$

The difference between the expectation of future prices $\mathbb{E}(S_3) = \mu$, and the price given by the market makers to the liquidity traders, S_1, equals $\gamma\sigma^2\frac{i}{n+1}$. We can immediately observe that:

- The more risk averse these market maker is (i.e. larger γ), the higher the trading cost.
- The more volatile the market is (i.e. larger σ), the higher the trading cost.
- Larger order size (i) implies a higher trading cost.
- Having more Market Makers (i.e. Larger n) decreases the trading cost.

1.2.3 The Avellaneda-Stoikov Model

Model Framework

This is a model proposed by Avellaneda and Stoikov for market making in the stock market. It focuses on the quote-driven markets, and gives a framework to analyse the optimal strategy for a market maker under its assumptions.
Assume that the market price of an asset follows a Brownian Motion:

$$dS_t = \sigma dW_t \tag{1.2.9}$$

The market maker aims to continuously propose bid and ask prices for everyone else to buy or sell the asset. These prices are denoted by two stochastic processes $(S_t^b)_t$ and $(S_t^a)_t$.
To model the incoming transactions that happen randomly on the timeline, we denote them by two point processes $(N_t^b)_t$ and $(N_t^a)_t$. We assume that every time a transaction happens, the size of the transaction is always one unit. Thus, the inventory of the market maker is:

$$q_t = N_t^b - N_t^a \tag{1.2.10}$$

The intensity of the point processes is assumed to be a function of the price skews:

$$\lambda_t^b = \Lambda^b(\delta_t^b)\mathbf{1}_{q_{t-}<Q} \tag{1.2.11}$$

and

$$\lambda_t^a = \Lambda^a(\delta_t^a)\mathbf{1}_{q_{t-}>-Q} \tag{1.2.12}$$

where $\delta_t^b = S_t - S_t^b$ is the bid skew, $\delta_t^a = S_t^a - S_t$ is the ask skew, and Q is the maximum inventory the market maker is allowed to hold on to. This means when we hit the hard limit of inventory size, we would immediately stop providing liquidity on one side, to decrease the size of the inventory we have.
In the paper by Avellaneda and Stoikov, they focused on the case where

$$\Lambda^b(\delta) = \Lambda^a(\delta) = Ae^{-\kappa\delta} \tag{1.2.13}$$

with $A > 0$ and $\kappa > 0$.
The amount of cash on the market maker's hand is denoted by a process $(X_t)_t$, which has the following dynamics:

$$dX_t = S_t^a dN_t^a - S_t^b dN_t^b = (S_t + \delta_t^a) dN_t^a - (S_t - \delta_t^b) dN_t^b \tag{1.2.14}$$

And finally, the utility function that market makers try to optimise is the CARA uitility:

$$\mathbb{E}[-exp(-\gamma(X_T + q_T S_T - l(q_T)))] \tag{1.2.15}$$

where the $l(q_T)$ is a risk-liquidity premium.

The Hamilton-Jacobi-Bellman equation and its solution

The Hamilton-Jacobi-Bellman equation is a method to turn an optimisation problem into equivalent a partial derivative equation.[11] Consider a deterministic optimal control over the time period $[0, T]$

$$V_T(x(0), 0) = \min_u \left(\int_0^T C[x(t), u(t)]dt + D[x(T)] \right) \tag{1.2.16}$$

where $C[\cdot]$ is the scalar cost rate function and $D[\cdot]$ is a function that gives the bequest value at the final state, $x(t)$ is the system state vector, $x(0)$ is assumed given, and $u(t)$ for $0 \leq t \leq T$ is the control vector that we are trying to find.
The system must also be subject to:

$$\dot{x}(t) = F(x(t), u(t)) \tag{1.2.17}$$

where $F[\cdot]$ gives the vector determining physical evolution of the state vector over time.
If we apply this to our CARA utility function, and perform an expansion of $V(X_t, t)$ according to Ito's rule, we are left with the following equation to solve:

$$\begin{aligned} 0 = \partial_t u(t, x, q, S) + \frac{1}{2}\sigma^2 \partial^2_{SS} u(t, x, q, S) \\ + \mathbf{1}_{q<Q} \sup_{\delta^b} \Lambda^b(\delta^b)[u(t, x - S + \delta^b, q + 1, S) - u(t, x, q, S)] \\ + \mathbf{1}_{q>-Q} \sup_{\delta^a} \Lambda^a(\delta^a)[u(t, x + S + \delta^a, q - 1, S) - u(t, x, q, S)] \end{aligned} \tag{1.2.18}$$

for $q \in \{-Q \cdots + Q\}$ and $(t, S, x) \in [0, T] \times \mathbb{R}^2$, with the terminal condition

$$u(T, x, q, S) = -exp(-\gamma(x + qS - l(q))) \tag{1.2.19}$$

In the case of exponential intensities, this can be solved by considering the following anastz:

$$u(t, x, q, S) = exp(-\gamma(x + qS))v_q(t)^{-\frac{\gamma}{\kappa}} \tag{1.2.20}$$

We plug this into the HJB PDE, and we can show that $(v_q)_{|q|<Q}$ must satisfy:

$$\frac{d}{dt} v_q(t) = \frac{\kappa}{2}\gamma\sigma^2 v_q(t) - A(1 + \frac{\gamma}{\kappa})^{-(1+\frac{\kappa}{\gamma})}(\mathbf{1}_{q<Q} v_{q+1}(t) + \mathbf{1}_{q>-Q} v_{q-1}(t)) \tag{1.2.21}$$

This is a linear system of ODEs, and it can be solved with appropriate terminal conditions.
Finally, the solution to our optimisation problem is:

$$S^{b^*}(t,S,q) = S - \frac{1}{\kappa}ln(\frac{v_q(t)}{v_{q+1\,t}}) - \frac{1}{\gamma}ln(1+\frac{\gamma}{\kappa}) \tag{1.2.22}$$

$$S^{a^*}(t,S,q) = S + \frac{1}{\kappa}ln(\frac{v_q(t)}{v_{q+1\,t}}) + \frac{1}{\gamma}ln(1+\frac{\gamma}{\kappa}) \tag{1.2.23}$$

This optimal solution has three parts:

- The current reference price S
- Static part $\frac{1}{\gamma}ln(1+\frac{\gamma}{\kappa})$, which is to do with the intensity of orders (κ) and how risk-averse the market maker is (λ)
- A stochastic part $\frac{1}{\kappa}ln(\frac{v_q(t)}{v_{q+1\,t}})$ that depends on the current position of the market maker's inventory.

The market maker is risk averse, so it will try to lower the size of its position by keeping the spread of its quotes while shifting them towards one direction slightly, such that there is a greater chance to have a trade on one side than the other, effectively lowering its position.
This solution qualitatively tells us how a good market maker should quote. We would like to see the agent we trained to have these properties as well.

Chapter 2

Applications

2.1 Data and Environment

2.1.1 Data Source

The data in this project is collected from a Bitcoin exchange between 2019.05.21 – 2019.06.12. We have an order book of depth 10. We only use the rows where either the best bid price or the best ask price has changed from the last rows.

2.1.2 Environment Steps

Our agent will make a decision on its market-making price and volume, and the environment will step forward to the next time when either the best bid price or the best ask price has changed. If the new bid/ask price matched the bid/ask price offered by the agent, a trade is made.

The total number of steps within one environment is controlled by a variable, environment length. This start from 3000, which equals 1-3 hours of data. As the training continues, this length increases to 20000, which equals 24 hours.

The reason to start from a short environment length is that at the beginning of the training, the agents sometimes accumulate a huge positive or negative position, which magnifies the reward. The value part of the network has a lot of trouble learning the correct value of these states.

A second reason to start with a short environment length is that our A2C agent is gathering data from 16 environments simultaneously. A long environment length means the data of these 16 environments has a huge chance to overlap, which increases over-fitting.

However, it is also important to increase the environment length over training. The data towards the end of each environment has nothing to discount and add to them from future, which alters the value of these states and decreases the importance of the actions taken based on these states.

2.1.3 Reward Function

The natural choice of reward is the change in PnL. More precisely:

$$\begin{aligned} Reward &= PnL_t - PnL_{t-1} \\ &= Cash_t + Position_t * (MidPrice_t - MidPrice_{t-1}) \end{aligned} \tag{2.1.1}$$

Without discount ($\gamma = 1$), the total reward the agent learns to maximise is just the final PnL. With a γ closer to 1, the agent learns to optimise a longer-term PnL, instead of a shorter-term PnL. However, a γ very close to 1 gives a larger impact of the effect of end of environment, and also more over-fitting. I chose gamma to be 0.999, which means a half-life of $-ln(2)/ln(1000) \approx 700$.
In reality, we do not just maximise the expectation of return. We want to take into account the risk of having a huge variance, and maximise a utility function. Thus, we want to give a penalty for having a large variance or a large position.
The first method attempted is to directly subtract a function of position from the reward. For example, the following functions are tried:

$$\begin{aligned} Peanlty &= f(position) \\ f(x) &= \alpha x^2 \\ f(x) &= \alpha |x| \\ f(x) &= \alpha ln(1+x) \end{aligned} \tag{2.1.2}$$

In my experience, the logarithmic penalty gives the best result.
However, [12] presented an approach called Asymmetrically dampened PnL. If the reward is positive, it is multiplied by a factor smaller than 1. If the reward is negative, it is unchanged. Suppose we have a large position, the market moves (roughly) in either direction with equal probability in the long term, so the expected reward will be negative is we keep a large position in our hand. Thus the agent can learn to lower its position without directly giving a penalty.
A combination of the logarithmic penalty and the asymmetrical dampened PnL is used in my code.

2.2 Observation Space

2.2.1 Market State Observation

Market observations are the observables derived from the order book.

- Bid/Ask Price: $Price^i_{Direction}$ for both directions and $i = 1 \dots 10$
- Midprice: $\frac{Price^1_{Bid}+Price^1_{Ask}}{2}$
- Bid/Ask Skew: $Price^i_{Direction} - Midprice$ for both directions and $i = 1 \dots 10$
- Spreads: $Price^i_{Ask} - Price^i_{Bid}$ for $i = 1 \dots 10$
- Price Level Distance to Midprice: $\frac{Price^i_{Direction}}{Midprice} - 1$ for both directions and $i = 1 \dots 10$. This observation is proposed in [13]
- Bid/Ask Size: $Volume^i_{Direction}$ for both directions and $i = 1 \dots 10$
- Bid/Ask Size: $Volume^i_{Direction}$ for both directions and $i = 1 \dots 10$
- Cumulative Bid/Ask Size: $\sum_{j=1}^{i} Volume^j_{Direction}$ for both directions and $i = 1 \dots 10$
- Order Imbalance Ratio: $\frac{CumulativeVolume^i_{Ask} - CumulativeVolume^i_{Bid}}{CumulativeVolume^i_{Ask} + CumulativeVolume^i_{Bid}}$
- Delta time: Time from last step
- Relative Strength Index (30/90/270) on midprice

The Relative Strength Index is an indicator used in technical analysis of the market, measuring the momentum of price. It is defined by splitting the price movement into two parts: positive and negative. The moving averages of these two parts are calculated, and RSI is defined by:

$$RSI = 100 - \frac{100}{1 + RS} RS = \frac{SMMA(U, n)}{SMMA(D, n)} \tag{2.2.1}$$

where SMMA is the smoothed moving average function, U and D are the upward and downward price change processes.

2.2.2 LSTM observations

Origin of Idea of LSTM

In the paper [14], the author suggests that we can pre-train a LSTM with supervised learning on the market observations, and take the LSTM outputs as observables for reinforcement learning. This is exactly what I did here.

The LSTM architecture

The LSTM in my code has 1 layer, 100 hidden cells. Its output is then connected to a linear layer. The majority of price movements between two successive ticks have a magnitude of less than or equal to 3. The market tick size is 0.5, which means the price movement is always a multiple of 0.25, since it is possible for only one side of the price moves. The midprice movement data is clipped by $[-3, 3]$, which gives 25 possibilities. Thus the output size of the linear layer has to be 25.

The output from the linear layer will pass a softmax function and will be treated as a discrete probability distribution. A cross-entropy loss is used to train the LSTM.

Performance of LSTM

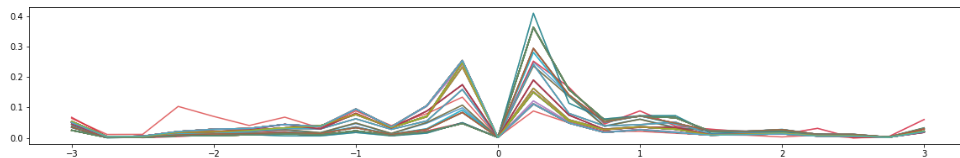

Figure 2.1: The LSTM-predicted probability distribution of midprice movement for 100 random states

A output of 100 randomly selected states are plotted. We can see that the LSTM model correctly understands that the price movement is $+-0.25$ in most cases, and the price movement is never zero.

For some states, the LSTM can predict the price movement with a degree of certainty, up to 40% confidence in the price will move up by 0.25 in the next tick. However, for some other states, the LSTM is a lot less confident.

To better understand the performance of this LSTM, I dropped the size of the price movement, leaving only $+1$ and -1 of direction. I compared the sign of the correct direction of price movement, and the sign of the most probable price movement predicted by the LSTM. The final LSTM

has a 69.3% probability of correctly predicting this direction, which is very impressive.

2.2.3 Trading Agent Observation

These are the observations of the trading agent's own records.

- Cash: The agent starts with 0 cash, but it can gain/borrow cash with 0 cost for the purpose of providing liquidity. This is a valid assumption because we're working with just a few days of data.

- Position: The agent knows its long/short position in bitcoin. No explicit upper limit for the position is set, but we used Asymmetrically dampened PnL to tell the agent not to accumulate a large position.

- PnL: This is the mark-to-market PnL of the agent in one environment, calculated by $Cash + Position * MidPrice$.

- Unrealized PnL: Within one environment, The agent can return to a state of nearly 0 position multiple times. The Unrealized PnL is the PnL since the last time the agent clears all its positions.

2.2.4 Observation memory

We're dealing with is a time-series, and the usual model to deal with time series is a recurrent neural network. However, in our case of either deep Q-learning or A2C, we perform one training step after receiving one (s, a, r, s') tuple, while for a recurrent neural network setup, we need the entire episode or the entire chain of state-action-reward to perform one training. This means we need to gather a lot more experience from the environment to perform the same level of training, which is very slow.

As a result, our network for estimating Q-value or deciding the policy has a simple feed-forward architecture, with no recurrent part in it. However, it is still important for our agent to remember the last few observations and be able to make decisions based on a brief piece of history, not just the current state.

This is done by a wrapper outside the environment, recording the observations. The dimension of the observation space before this wrapper is 94. I want the agent to make decisions based on the last 5 observations. After the wrapper, the observation space has a dimension of $5 * 94$.

2.3 Action Space

After each observation, the agent has to decide 4 things: the Bid/Ask price it is willing to provide liquidity at, and the volume it is willing to buy/sell.

The first attempt is to consider the Bid/Ask price actions as discrete, but the volume actions as continuous. The neural network must decide on which pair of Bid/Ask skew it wants to take, and also pick 2 volumes for them from a normal distribution. However, the volume has a hard lower limit of zero, and the normally distributed normal action often hits the boundary, and the agent eventually decides to frequently not provide any liquidity to the market. This is not what we want. I tried a slightly different setup where the volume number of drawn from a log-normal distribution instead of a normal distribution. This time, the volume that agent is willing to quote is often larger than the entire market, which doesn't do anything because nobody on the market can take such volume, and the agent is not learning anything.

Finally, I decided to simplify the situation by setting a constant bid/ask volume, and the Bid/Ask prices are from a discrete set of choices, with fixed distances from the last midprice.

	Ask_Skew	Bid_Skew	Ask_Size	Bid_Size
Action 0	0.5	0.5	100	100
Action 1	1	1	100	100
Action 2	2	2	100	100
Action 3	0.5	2	100	100
Action 4	2	0.5	100	100
Action 5	1	1.5	100	100
Action 6	1.5	1	100	100

If we look at the table, the first 3 actions have the same skew on both sides, which means the agent is willing to take equal amount of orders from both directions. For action 3-6, the skews are different for bid and ask, which gives the agent the option to have a greater chance to trade on one side than the other.

Some special actions, for example an action to clearing all positions with a market order, or an action to do nothing, are considered and tested. However, including these actions dramatically decreases the performance of the learning process, and I excluded them in this thesis.

2.4 Agent

A lot of agents are implemented and tried. I spent a long time testing different extensions of the deep Q-learning.

The first attempt is to combine all the extensions, just like the rainbow paper. However, categorical DQN is causing a lot of problems. In the original setup for the categorical DQN, there are only two actions. While in our case, there are often more than 5 actions. We also have a very wide range of possible discounted total reward, from -100 to +50. If we set the support of the distribution smaller than this, once the agent receives rewards outside this range, the estimation of the Q-values becomes incorrect and unstable.

The natural huge noise from our data also makes the noisy network method inappropriate. The standard deviation of the noise never managed to reduce to zero, which means any estimation of the Q-value will have a huge uncertainty, and the network doesn't converge to a well-defined policy.

After removing these 2 extensions, a version of DQN with the other 5 extensions managed to converge to a sub-optimal solution: The agent managed to learn to reduce the risk of losing a lot of money, but mean PnL is negative, which means we still losses money slowly over time.

Thus, I switched to an Advantages Actor-Critic agent with the following architecture.

The market observations, LSTM observations and the agent observations are combined and fed into a layer that remembers the last 5 observations. All these 5 observations are treated as input of the main neural network. These inputs are first sent to a common linear layer of size 256, followed by a activation function of Leaky ReLU. This output of the common part is sent to the policy part and the value part separately. Policy is trained by the policy gradient, with a baseline of the value output; Value is trained by Mean Square Error; Exploration is encouraged by adding entropy of the output policy to the total loss function.

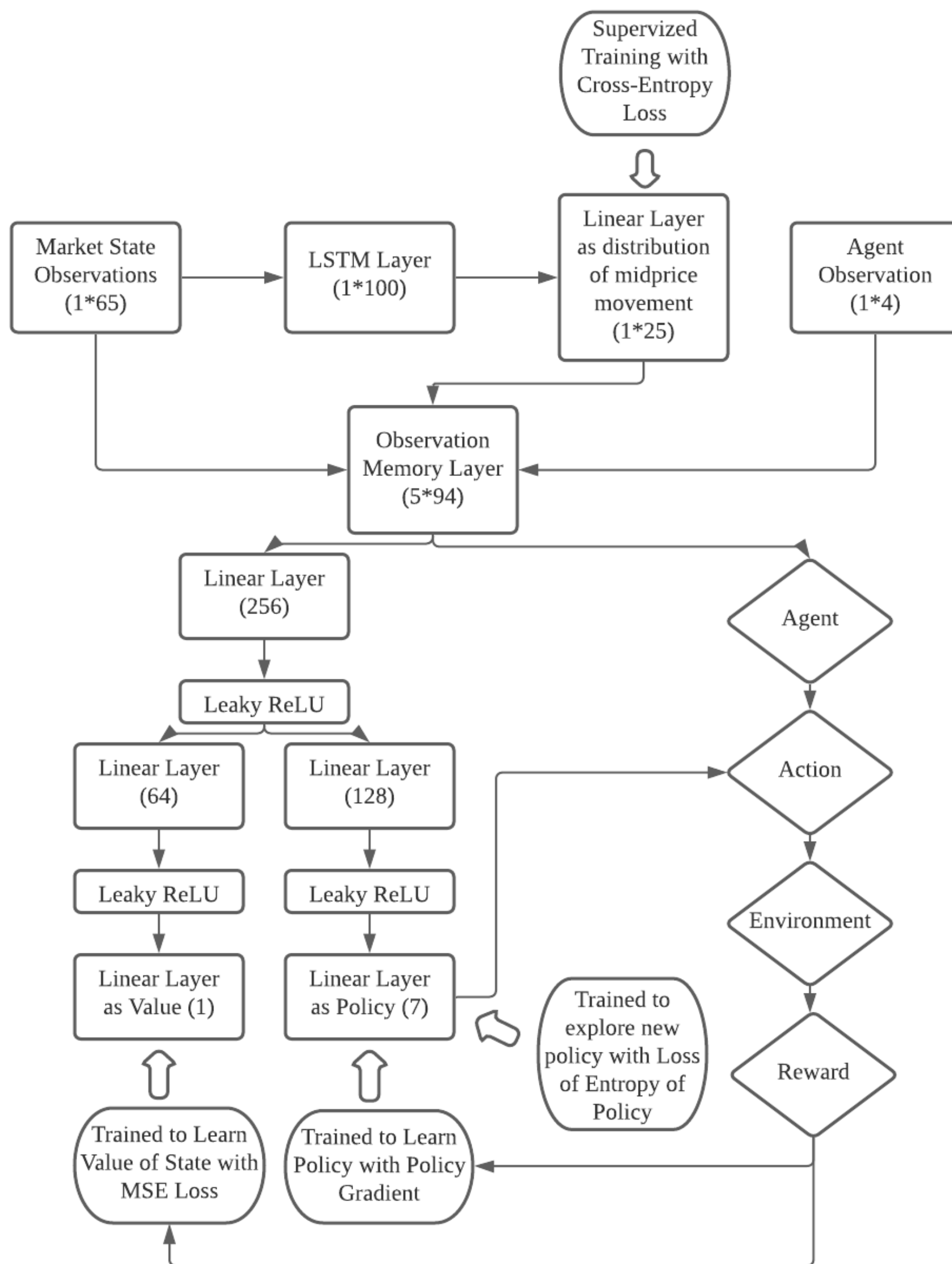

Figure 2.2: The Architecture of the entire agent

2.5 Computing Speed Analysis

This project runs on a Google Colab with a Tesla V100-SXM2-16GB GPU. The training of the neural network is computed with this GPU using PyTorch, while the simulation of the trading environment is performed by the CPU. The structure of the reinforcement agent is build upon a package called PTAN, which is proposed in this book.[15] Without any training of the Network, the CPU can generate experiences at the speed of 130 frames per second, or 1 frame per 7.6 milliseconds. I tested using parallel processing to simulate multiple environments at the same time,

but the overall speed has not changed.

With the GPU training of the neural network, the speed drops to 110 frames per second, which is 9 milliseconds per frame. A few line profilers in this book[16], including *lprun*, are used, and they show that the bottleneck of the speed is on the CPU side, where the simulation of trading environment is slow.

2.6 Training process and hyper-parameter tuning

2.6.1 Hyper-parameters

A2C agent parameters

- NUM_ENVS: 16. This is the number of environments that are running and collecting data in a synchronized way. A larger NUM_ENVS may make the learning process more stable and robust, but 16 is chosen because of the memory limit on my computer.

- $REWARD_STEPS$: 16. This is the number of steps the A2C agents uses to compute the advantage function, by rolling out the Bellman Equation. It is usually between 10-20.

Learning rate: Lambda

- $LEARNING_RATE$: 0.00003. This is the initial learning rate. What is a good learning rate depends on the architecture of the network. A wide range of learning rates are tested, and the I found the agent learns most stably at around 0.00003.

- $LEARNING_RATE_DECAY$: 0.99999. The learning rate follows an exponential decay. Since my $ENVIRONMENT_LENGTH$ is different in each epoch, it makes more sense to adjust the learning rate after each step of the optimiser, instead of after each epoch.

Exploration: Epsilon and Beta

- $EPSILON$: $1 \rightarrow 0.03 \rightarrow 0.02$ as the $STEP_INDEX$ increases from $0 \rightarrow 1000000 \rightarrow 2000000$. This is the probability of the agent will take a completely random action regardless of the neural network output. This is especially important at the early stages of training. Over the early period of learning, it is very easy for the network to decide to take one action with 100% probability, which could lead to the agent gathering a very large position of bitcoin. The rewards become negative and volatile, and the value network fails to learn anything. Having an $EPSILON$ parameter makes it possible for the agent to take random actions and keep learning in such a situation.

- $BETA$: 0.01 $\rightarrow$ 0.0001 as the $STEP_INDEX$ increases from 0 $\rightarrow$ 2000000. This is the coefficient of the entropy loss. A larger $BETA$ means our agent will focus more on trying out all actions, thus increases exploration. The action space has 7 actions, and the entropy is minimised when we select all actions with equal probability. In this case, the entropy loss is $-ln7 \approx -1.9$. In comparison, the loss for MSE and Policy is about 0.1 after a certain amount of training. In order to have a sensible policy, the β is chosen such that all these losses have a similar magnitude. Thus, β is chosen to be 0.01 at the beginning of the training. It is than linearly reduced to 0.0001, so the agent is able to focus on learning the policy.

Gradient Clip

- $CLIP_GRAD$: 0.1. At the beginning of the training, the gradient along some dimension could be very large, and the learning will be very unstable. Thus, we limit all the gradients to between -0.1 and 0.1.

2.6.2 Training Process

The training process is tracked and monitored by looking at the actions and PnLs of the a single environment. Some key parameters are also recorded and plotted by using a Tensorboard.

PnLs and Actions in Sample Environments

At the beginning of the training, the agent is basically taking random actions. A large negative position is allowed to build up, and the final PnL passively mostly depends on the market movement.

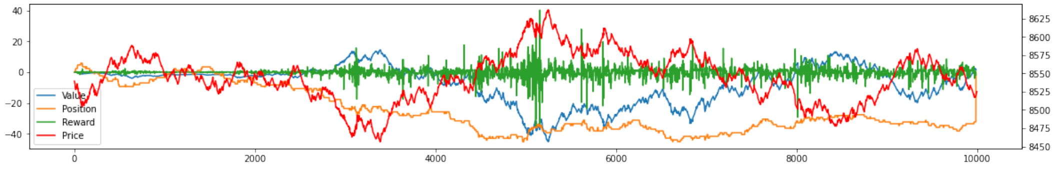

Figure 2.3: The PnL, Position, Reward and Market Price of an Environment, at early stage of training

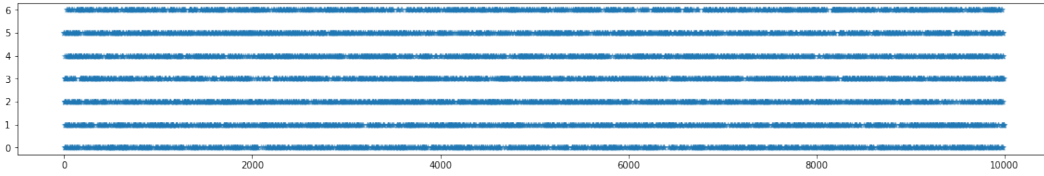

Figure 2.4: The Actions of the Agent in this Environment, at early stage of training

As we keep learning, the model occasionally runs into a sub-optimal policy, where it always quotes evenly on both sides, with a large spread. Again, large positions are allowed to built up, and the final PnL mainly depends passively on the market price movement.

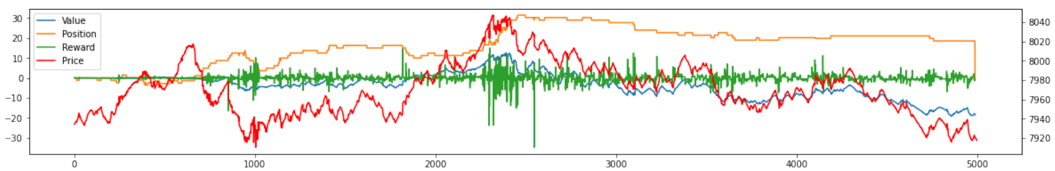

Figure 2.5: The PnL, Position, Reward and Market Price of an Environment, at mid stage of training

Later, the agent learns to limit its position. Its position alternates between positive and negative. All actions have some probability to be selected in some states. The PnL is negative, but it has little direct correlation to the market price.

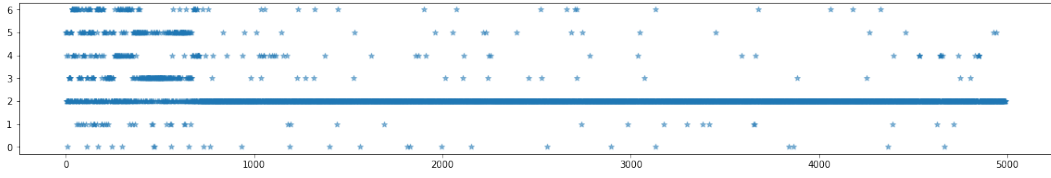

Figure 2.6: The Actions of the Agent in this Environment, at mid stage of training

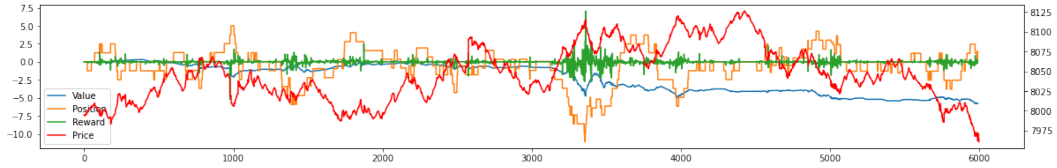

Figure 2.7: The PnL, Position, Reward and Market Price of an Environment, at mid-to-late stage of training

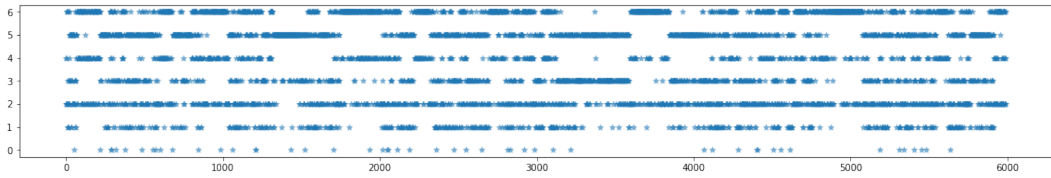

Figure 2.8: The Actions of the Agent in this Environment, at mid-to-late stage of training

Eventually, the agent manages to bring its PnL to very close to zero, and even positive. By looking at the actions, we can see that action 2, which corresponds to a large even quote (2 on both sides), is the most common action. This makes sense because when the agent is uncertain about the direction of market movement, this is the safe action to take.

From time to time, the agent also takes actions of 3-6, which are the one with uneven skews. This means our agent is now actively controlling its position based on its observations. This is a characteristics of a trained market making agent.

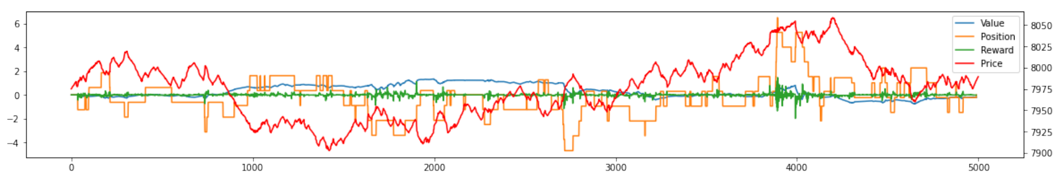

Figure 2.9: The PnL, Position, Reward and Market Price of an Environment, at late stage of training

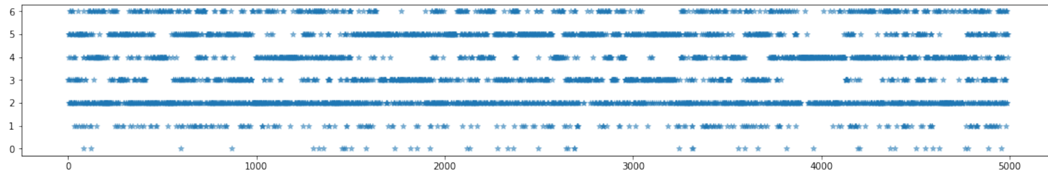

Figure 2.10: The Actions of the Agent in this Environment, at late stage of training

Gradients

After each back propagation, the maximum, variance and $L-2$ norm of the gradients are calculated. The gradients are very small but don't decrease to 0. This is very much expected because of the randomness in data. Each state should have a distribution of value instead of a single number. Unfortunately, categorical DQN requires a huge output dimension, and it is computationally too expensive to use.

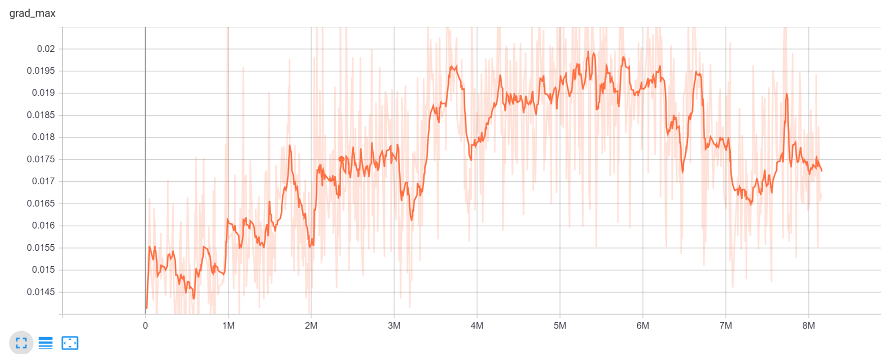

Figure 2.11: The Maximum of Gradient of Neural Network

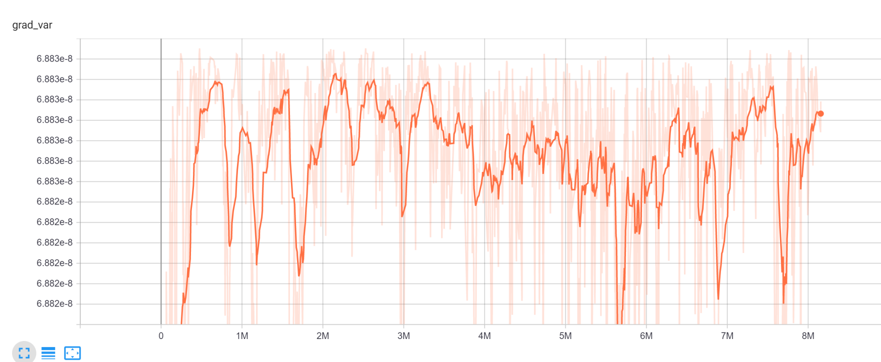

Figure 2.12: The Variance of Gradient of Neural Network

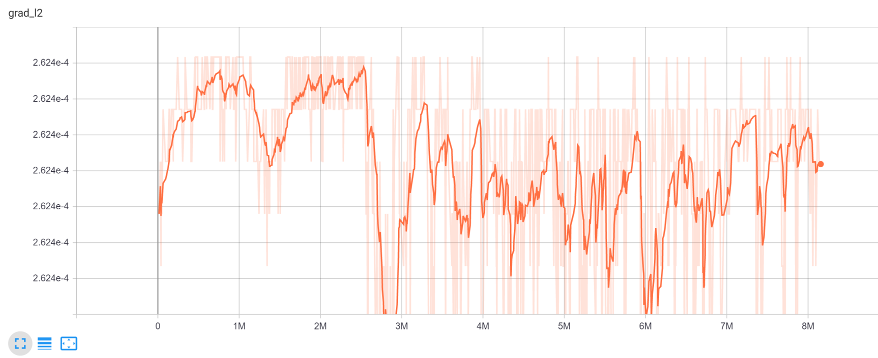

Figure 2.13: The L2-norm of Gradient of Neural Network

Neural Network Outputs

The batch rewards and the values of states are also tracked. Notice that the rewards are never positive despite having positive mean PnL at the later stages of the training. This is because we are using positional penalty and asymmetrically damped PnL, which takes away some PnL in the reward. Notice that the length of our environment is also increasing, which mean a constant slightly-negative rewards actually implies positive mean PnL.

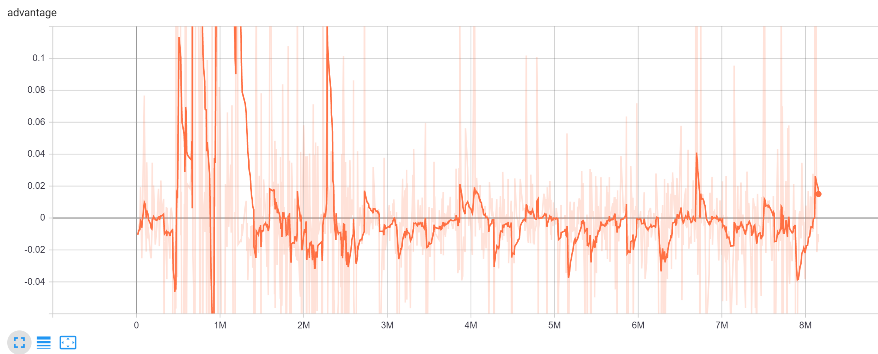

Figure 2.14: The Advantage Function

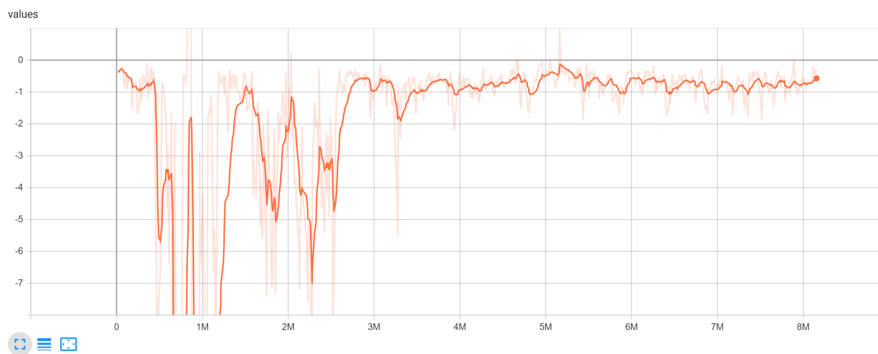

Figure 2.15: The Predicted Values of States

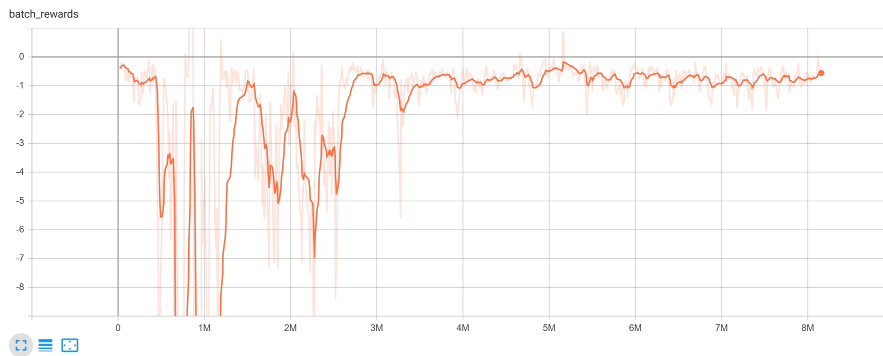

Figure 2.16: The average Rewards of 16 environments

Losses

Finally, we look at the losses. The entropy loss tells how certain the agent is while selecting actions. An entropy of -1.9 means the agent is taking a random policy, while an entropy of 0 means the agent has 100 percent confidence to take a certain action. At the beginning of training, the entropy quickly decreases to -1.7, which is a result of out large $BETA$. The entropy gradually decreases towards a final value of -0.6, meaning the agent now understands the difference between different states, and it can return a policy with confidence. It makes sense that the final entropy is non-zero. Our actions are very similar to each other. It is impossible to guarantee that one action is definitely better than the others given a state, because of the randomness of the market data.

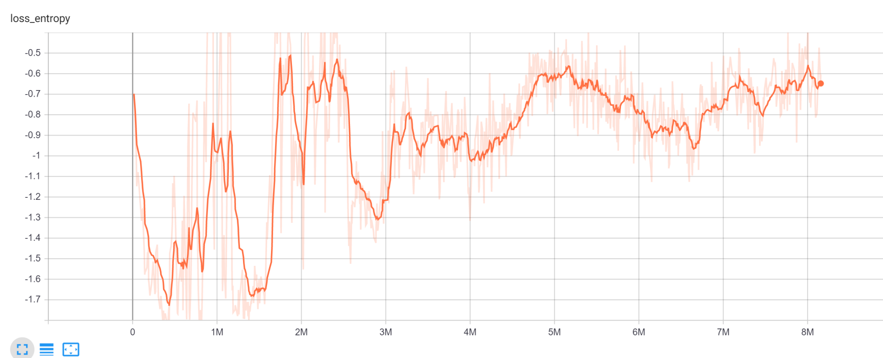

Figure 2.17: The Entropy Loss

The value loss and policy loss have large spikes in them. One reason is that the data is not uniform. The nature of the data is different when there is no large market order or important sudden market information, compared to the normal time period while only market makers and small traders are involved. Another reason could be because of the effect of end-of-environment, which shifts the value of the states away from the normal region.

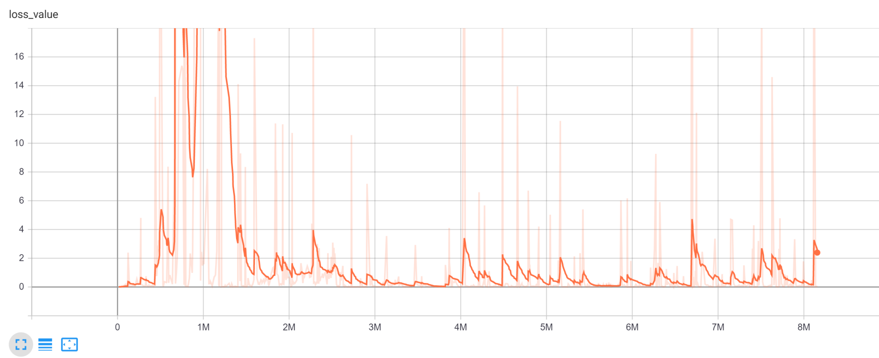

Figure 2.18: The Value Loss

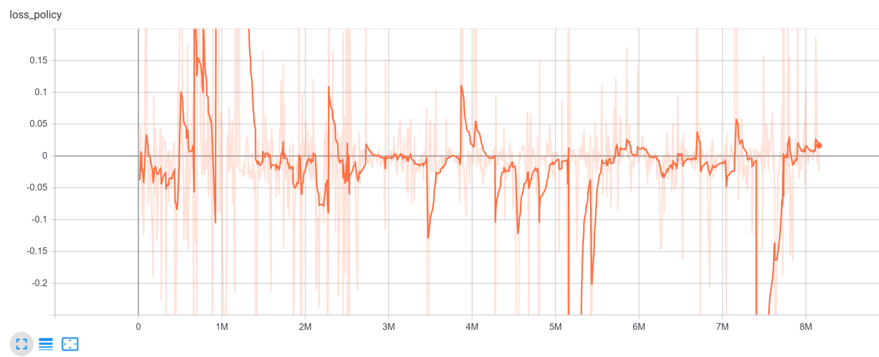

Figure 2.19: The Policy Loss

The total loss is plotted on a logarithmic scale. It is decreasing, but not converging to zero, as the data has huge randomness.

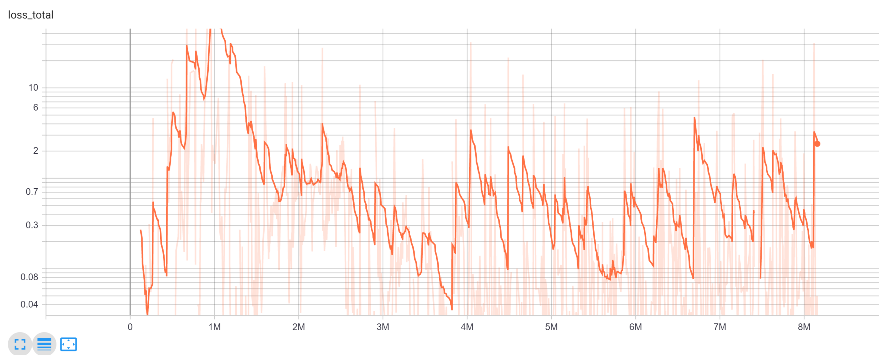

Figure 2.20: The Total Loss

2.7 Performance

The performance is evaluated with the following method: A piece of data of length 10000, which represents roughly 10 hours, is randomly selected. The agent take action based on the neural network output, and the final PnL is calculated. Repeat 100 times to plot a distribution of final PnL.

We achieved a Mean PnL of 0.51 and standard deviation of 10.1. I repeated this 4-hour perfor-

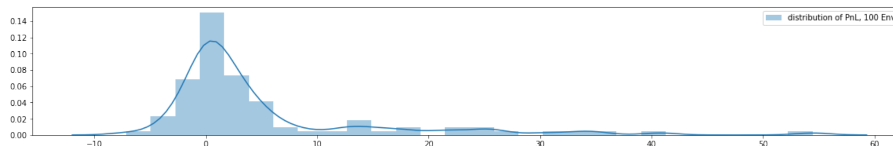

Figure 2.21: The Distribution of PnL

mance evaluations process multiple times, and they all give positive PnL.

By looking at the graph, we can see that the distribution of PnL is skewed. The agent learned to avoid any huge losses, with a maximum loss of less than 10. But it can occasionally gain a huge amount of money, with a profit up to 50. We can look at the distribution more closely by looking at the part with profit less than 10 and the part with profit greater than 10 separately.

Looking at the part with PnL between -10 and +10, we can observe that the PnL is more likely to

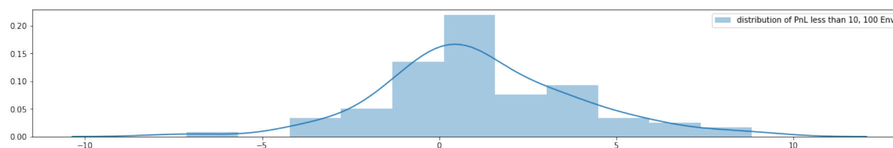

Figure 2.22: The Distribution of PnL, Given less than 10

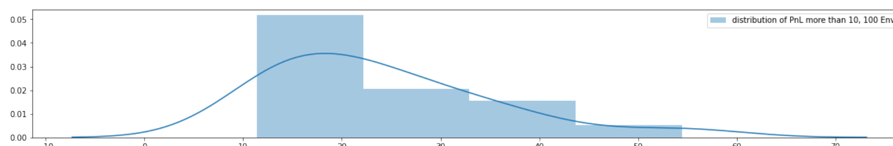

Figure 2.23: The Distribution of PnL, Given more than 10

be negative than positive. This is the cost of actively trying to keep a low magnitude of position. However, when the agent is confident with the direction of market movement, it allows itself to take a large position and make a huge profit from it. This is how the overall average PnL becomes positive.

The performance is compared with a benchmark, where the agent takes random action from the action space. In this case, the distribution is very even, with frequent loss of over 50. The mean PnL is slightly negative, and the standard deviation of PnL is at a much larger 30.
If we consider our goal as to maximise a utility function that is risk-averse, it is obvious that this agent outperforms the benchmark.

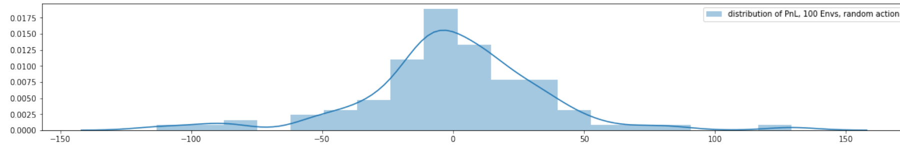

Figure 2.24: The Distribution of PnL, With random action

2.8 Further Development

There are a few areas that the result of this thesis may be further developed on:

- Testing these agents on a larger data set could potentially dramatically improve the performance. In some other papers in this field, the agents are trained with up to 3 years of data, across multiple assets, while I only have 1 month of data on bitcoin. With such a short span of data, the agent we trained may not be able to cope with certain tail events of market movement.
- More computational power would also be great. The size of our model is limited by the speed. It takes about 24 hours for the program to run 10 million frames.
- Quoting prices in a volume space. In real world high-freq trading, instead of directly quoting the prices, people sometimes quote with a volume number that is translated to a price according to the current cumulative bid/ask volume. For example, instead of directly saying the price of my limit order is 0.5 above the midprice, I can say I want my price to be at the place where the market cumulative ask volume is 10 million. This means a market order of 10 million has the impact of moving the market such that my order becomes the best ask price. Any market order larger than 10 million means my order will be executed.

Appendix A

Source Codes

```
# -*- coding: utf-8 -*-
"""StockEnv-v7.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1i-ZQPN6zjcgZdlJ3isb-WiC82G0CZFzp

# Deep Reinforcement Learning and Electronic Market Making
"""


!pip install ptan==0.3
!pip install PyDrive

# Commented out IPython magic to ensure Python compatibility.
import gym
from gym.utils import seeding
import enum
import numpy as np
import pandas as pd
from functools import reduce
import matplotlib.pyplot as plt
# %matplotlib inline
# %load_ext tensorboard
```

```
import seaborn as sns
import time
import collections
from collections import namedtuple, deque
import math
import scipy
import torch
import torch.nn as nn
import torch.nn.utils as nn_utils
import torch.nn.functional as F
import torch.optim as optim
from torch import autograd
from torch.utils.tensorboard import SummaryWriter
from IPython import display
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
from IPython.display import clear_output
import sys
import ptan
import warnings
from datetime import datetime
from google.colab import files
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
warnings.filterwarnings("ignore", category=UserWarning)
print(torch.cuda.get_device_name(0))
device="cuda"
plt.rcParams["figure.figsize"] = (20,3)

!rm -rf ./runs/
import os
logs_base_dir = "runs"
os.makedirs(logs_base_dir, exist_ok=True)
```

```
HYPERPARAMS = {
    'stockenv': {
        'stop_reward':          200,
        'run_name':             'StockEnv',
        'learning_rate':        0.00003,
        'learning_rate_decay':  (1-1/100000),
        'min_learning_rate':    0,
        'gamma':                0.999,
        "epsilon_start":        0.3,
        "epsilon_final":        [0.03,0.02],
        "epsilon_final_frame":        [500000,1000000],
        'ENV_TRAIN_COUNT':      1,
        "position_penalty":     0,
        "positive_reward_reduction": 0.1,
        "ticks_to_look_back":   5,
        "order_book_depth":     6,
        "ENTROPY_BETA":         [0.01,0.005,0.001,0.0001,0],
        "ENTROPY_BETA_FRAME":          [0,500000,1000000,2000000,3000000],
        "NUM_ENVS":             16,
        "REWARD_STEPS":         16,
        "CLIP_GRAD":            0.1,
        "DATALENGTH_INCREASE_FREQ":          4,

    },
}
params=HYPERPARAMS["stockenv"]

"""## 1. Data and Feature Selection

### 1.1 Data Collection

#### From QPython.Connection

print("Start: ",time.ctime())
```

```
datelist=["2019.05."+"{:02d}".format(x) for x in range(22,32,1)]+["2019.06."+"{:02d}"
datadate="2019.05.21"
q = qpython.qconnection.QConnection(host='3.9.195.248', port=41822, pandas=True)
q.open()
all_data=q("select from orderBook where date="+datadate+", time within 00:00:00.000 2
print("Done:",datadate," at ",time.ctime())
for datadate in datelist:
  tempdata=q("select from orderBook where date="+datadate+", time within 00:00:00.000
  all_data=all_data.append(tempdata)
  print("Done:",datadate," at ",time.ctime())
q.close()
print("Finish: ",time.ctime())
"""

#all_data.to_csv('data.csv')

#files.download('data.csv')

"""#### From Google Drive"""

auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

downloaded = drive.CreateFile({'id':"1ceUD9i8UVZkj2CY9-LrtXEHvrM1Bg4Rn"})
downloaded.GetContentFile('data.csv')

all_data = pd.read_csv('data.csv')

all_data.drop(columns=["Unnamed: 0"],inplace=True,axis=1)

"""#### Data Inspection"""

all_data.reset_index()
```

```
all_data.shape

all_data.head()

all_data.plot(x="time",y="bid1")

"""### 1.3 Data PreProcessing"""

all_data_copy=all_data.copy()
all_data_copy.sort_values(by=["date","time"],inplace=True)
all_data_copy.columns

def converttime(timestr):
  return datetime.strptime(timestr[7:-3], '%H:%M:%S.%f')

def RSI(priceseries,n):
  delta = priceseries.diff()
  dUp,dDown=delta.copy(), delta.copy()
  dUp[dUp<0]=0
  dDown[dDown>0]=0
  RolUp=dUp.rolling(n).mean()
  RolDown=dDown.abs().rolling(n).mean()
  RS = RolUp / RolDown
  rsi= 100.0 - (100.0 / (1.0 + RS))
  return rsi

orderbookdepth=params["order_book_depth"]
tickstolookback=params["ticks_to_look_back"]
all_data_copy["time"]=all_data_copy["time"].apply((lambda row: converttime(row)))
all_data_copy.drop([x+str(y) for x in ["bid","ask","bidSize","askSize"] for y in range
all_data_copy["deltatime"]=(all_data_copy["time"]-all_data_copy["time"].shift(1))
all_data_copy["deltatime"]=all_data_copy.apply(lambda row: row["deltatime"].total_seco
all_data_copy["midprice"]=(all_data_copy["ask1"]+all_data_copy["bid1"])/2
all_data_copy["RSI_30"]=RSI(all_data_copy["midprice"],30)
all_data_copy["RSI_90"]=RSI(all_data_copy["midprice"],90)
```

```
all_data_copy["RSI_270"]=RSI(all_data_copy["midprice"],270)
for j in range(orderbookdepth):
    all_data_copy["PLDM_Ask"+str(j+1)]=(all_data_copy["ask"+str(j+1)]/all_data_copy["mi
    all_data_copy["PLDM_Bid"+str(j+1)]=(all_data_copy["bid"+str(j+1)]/all_data_copy["mi
    all_data_copy["CumulativeAskSize"+str(j+1)]=np.sum(pd.concat([all_data_copy["askSiz
    all_data_copy["CumulativeBidSize"+str(j+1)]=np.sum(pd.concat([all_data_copy["bidSiz
    all_data_copy["oir"+str(j+1)]=(all_data_copy["CumulativeBidSize"+str(j+1)]-all_data_
    all_data_copy["Spread"+str(j+1)]=all_data_copy["ask"+str(j+1)]-all_data_copy["bid"+
    all_data_copy["bid"+str(j+1)]-=all_data_copy["midprice"]
    all_data_copy["ask"+str(j+1)]-=all_data_copy["midprice"]
all_data_copy["midprice_-1"]=all_data_copy["midprice"]-all_data_copy["midprice"].shift
#for i in range(tickstolookback):
#  for j in range(orderbookdepth):
#     for d in ["bid","ask","bidSize","askSize","oir"]:
#         all_data_copy[d+str(j+1)+"_"+str(-i-1)]=all_data_copy[d+str(j+1)].shift(i+1)
#  all_data_copy["deltatime_"+str(-i-1)]=all_data_copy["deltatime"].shift(i+1)
all_data_copy.drop(all_data_copy.head(271).index,axis=0,inplace=True)
all_data_copy.drop(all_data_copy.tail(1).index,inplace=True)
all_data_copy=all_data_copy.reset_index()

for index in all_data_copy.columns:
    try:
        print(index,": Avg: ",np.mean(all_data_copy[index][:10000]),"+-",np.std(all_data_c
    except:
        print(index,": Min:  ",np.min(all_data_copy[index][:10000]),"Max:",np.max(all_data

"""### 1.4 Recurrent Neural Network

#### Define Network
"""


lstmrange=3
lstmclassnum=lstmrange*8+1

class LSTM(nn.Module):
```

```
    def __init__(self, input_size=1, hidden_layer_size=100, output_size=1):
        super().__init__()
        self.hidden_layer_size = hidden_layer_size
        self.num_layers=1
        self.lstm = nn.LSTM(input_size, hidden_layer_size, num_layers=self.num_layers)
        self.linear = nn.Linear(hidden_layer_size, output_size)
        self.hidden_cell = (torch.zeros(self.num_layers,1,self.hidden_layer_size).cuda

    def forward(self, input_seq):
        lstm_out, self.hidden_cell = self.lstm(input_seq.view(len(input_seq) ,1, -1),
        self.hidden_cell[0].detach_()
        self.hidden_cell[1].detach_()
        predictions = self.linear(lstm_out.view(len(input_seq), -1))
        return predictions

columns=[x+str(y+1) for x in ["bid","ask","bidSize","askSize","oir"] for y in range(6
nnsoftmax = nn.Softmax(dim=1)
lstmmodel = LSTM(input_size=len(columns),output_size=lstmclassnum)
lstmmodel.cuda()
print(lstmmodel)

"""#### Train Network

torch.autograd.set_detect_anomaly(True)
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lstmmodel.parameters(), lr=0.01)
losslist=[]
losslistx=[]
loss=np.inf
startframe=0
lstmdatalength=1000
epoch_count=0
lrs=torch.optim.lr_scheduler.ExponentialLR(optimizer, 0.995)
while loss>0.1 and epoch_count<500:
  startframe+=100
```

```
    lstmdatalength+=200
    lstminput=all_data_copy.iloc[startframe:startframe+lstmdatalength,:][columns].to_nun
    lstminput=torch.tensor(lstminput).view(lstmdatalength,1,len(columns)).float().cuda(
    lstmtarget=all_data_copy.iloc[startframe+1:startframe+lstmdatalength+1,:]["midprice
    lstmtarget=np.rint(np.clip(lstmtarget,-lstmrange,+lstmrange)*4)+lstmrange*4
    lstmtarget=np.array(lstmtarget,dtype=np.int)
    lstmtarget=torch.tensor(lstmtarget).view(lstmdatalength,1).cuda().view(-1)
    #print(lstmtarget.size())
    epoch_count+=1
    lstmoutput=lstmmodel(lstminput)
    #print(lstmoutput.size())
    optimizer.zero_grad()
    loss_v=loss_function(lstmoutput,lstmtarget)
    loss=loss_v.item()
    loss_v.backward()
    losslist.append(loss_v.item())
    if loss_v.item()<=min(losslist):
        torch.save(lstmmodel.state_dict(), 'lstmmodel.pth')
        #print("Model Saved")
    losslistx.append(epoch_count)
    optimizer.step()
    lrs.step()
    plt.figure()
    if epoch_count%10==0:
        clear_output(wait=True)
        plt.scatter(x=losslistx[-50:],y=losslist[-50:])
        plt.show()
        sns.distplot(torch.argmax(lstmoutput.detach().cpu(),axis=1))
        sns.distplot(lstmtarget.detach().cpu())
        plt.ylim((0,4))
        plt.xlim((8,17))
        plt.show()
    #print("Epoch: {:2d}, lr: {:2f}, Loss: {:.4f}".format(epoch_count,optimizer.param_g

torch.save(lstmmodel.state_dict(), 'lstmmodel.pth')
```

```
files.download('lstmmodel.pth')

#### Load Network From Drive
"""

downloaded = drive.CreateFile({'id':"1cbShBw1hcusAIiUjVpdcaJuPnEIT8KBt"})
downloaded.GetContentFile('lstmmodel.pth')

state_dict = torch.load('lstmmodel.pth')
lstmmodel.load_state_dict(state_dict)

"""
lstmdatalength=all_data_copy.shape[0]-1
lstminput=all_data_copy.iloc[0:lstmdatalength,:][columns].to_numpy(dtype=np.float64)
lstminput=torch.tensor(lstminput).view(lstmdatalength,1,len(columns)).float().cuda()
lstmtarget=all_data_copy.iloc[1:lstmdatalength+1,:]["midprice_-1"].to_numpy(dtype=np.
lstmtarget=np.rint(np.clip(lstmtarget,-lstmrange,+lstmrange)*4)+lstmrange*4
lstmtarget=np.array(lstmtarget,dtype=np.int)
lstmtarget=torch.tensor(lstmtarget).view(lstmdatalength,1).cuda().view(-1)
lstmoutput=lstmmodel(lstminput)
lstmoutputsign=torch.sign(torch.argmax(nnsoftmax(lstmoutput),axis=1).detach().cpu()-1
lstmtargetsign=torch.sign(lstmtarget.detach().cpu()-12).numpy()
np.sum(lstmoutputsign==lstmtargetsign)/lstmtargetsign.shape[0]
"""

lstminput=all_data_copy[columns].to_numpy(dtype=np.float64)
lstminput=torch.tensor(lstminput).view(all_data_copy.shape[0],1,len(columns)).float()
lstmoutput=lstmmodel(lstminput)
lstmoutput=nnsoftmax(lstmoutput).detach().cpu().numpy()
for _ in range(100):
  plt.plot(lstmoutput[np.random.randint(lstmoutput.shape[0])],alpha=0.7)

for i in range(lstmoutput.shape[1]):
  all_data_copy["LSTM_"+str(i)]=lstmoutput[:,i]
```

```
for index in all_data_copy.columns:
  try:
    print(index,":␣Avg:␣",np.mean(all_data_copy[index][:10000]),"+−",np.std(all_data_c
  except:
    print(index,":␣Min:␣␣",np.min(all_data_copy[index][:10000]),"Max:",np.max(all_data

"""## 2. Market Making Environment"""

class StocksEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self,data,randomdata=False):
       self.data=data
       self.datatotallength=data.shape[0]
       self.tickstolookback=params["ticks_to_look_back"]
       self.orderbookdepth=params["order_book_depth"]
       self.randomdata=randomdata
       self.outputplot=True
       self.datastartframe=0
       self.datalength=5000
       self.select_data()
       self.action_space=gym.spaces.Box(low=np.zeros((4,)),high=np.array([10,10,1000000
       obs_space_dim=self.data.shape[1]+3
       self.observation_space=gym.spaces.Box(low=np.zeros((obs_space_dim,)),high=((np.
       self.seed()
       self.envcount=0

    def set_datalength(self,datalength):
       self.datalength=datalength
       #pass

    def select_data(self):
       if self.randomdata:
          self.datastartframe=np.random.randint(low=0,high=self.datatotallength−self.da
          #self.datastartframe+=2000
```

```
      #self.datalength+=1000
    self.dataslice=self.data.iloc[self.datastartframe:(self.datastartframe+self.data

  def reset(self):
    if self.envcount==0:
      self.select_data()
    self.positionpenalty=params["position_penalty"]
    self.envcount=(self.envcount+1)%(params["ENV_TRAIN_COUNT"])
    self.timesincelastclearpos=0
    self.timestamp=0
    self.value=0
    self.lastvalue=0
    self.cash=0
    self.position=0
    self.valuelist=[0]
    self.positionlist=[]
    self.rewardlist=[]
    self.datatick=self.dataslice.iloc[self.timestamp,:]
    self.lasttime=self.datatick["time"]
    self.midprice=self.datatick["midprice"]
    self.deltatime=self.datatick["deltatime"]
    return pd.concat([pd.Series({"deltatime":self.deltatime,"midprice":self.midprice

  def step(self,action):
    BidSpread, AskSpread, BidVolume, AskVolume=action
    self.timestamp+=1
    if self.timestamp>=self.datalength:
      raise RuntimeError("Done")
    if self.timestamp>=self.dataslice.shape[0]-10:
      return self.calcposition(-100,-100,-min(self.position*self.midprice,0),max(se
    else:
      return self.calcposition(BidSpread,AskSpread,BidVolume, AskVolume,False)

  def calcposition(self,BidSpread,AskSpread,BidVolume, AskVolume,done):
    BidPrice=self.midprice-BidSpread
```

```
AskPrice=self.midprice+AskSpread
self.datatick=self.dataslice.iloc[self.timestamp,:]
datatick=self.datatick
self.midprice=datatick["midprice"]
#print(BidPrice,AskPrice)
MarketBidPrices=self.midprice+datatick[["bid"+str(x+1) for x in range(self.order
MarketAskPrices=datatick[["ask"+str(x+1) for x in range(self.orderbookdepth)]]+
MarketBidVolumes=datatick[["bidSize"+str(x+1) for x in range(self.orderbookdepth
MarketAskVolumes=datatick[["askSize"+str(x+1) for x in range(self.orderbookdepth
#print(MarketBidPrices)
#print(MarketAskPrices)
BidCount=(MarketAskPrices<=BidPrice).sum()
#print(BidCount)
for i in range(BidCount):
  BidVolume-=self.Deal("Bid",MarketAskPrices[i],min(BidVolume,MarketAskVolumes[
AskCount=(MarketBidPrices>=AskPrice).sum()
#print(AskCount)
for i in range(AskCount):
  AskVolume-=self.Deal("Ask",MarketBidPrices[i],min(AskVolume,MarketBidVolumes[
unrealizedlastvalue=self.value
self.value=self.cash+self.position*self.midprice
self.valuelist.append(self.value)
self.positionlist.append(self.position)
if self.timesincelastclearpos>100 and abs(self.position)<0.01:
  self.reward=(self.value-self.lastvalue)-max((self.value-self.lastvalue),0)*pa
  self.timesincelastclearpos=0
  self.lastvalue=self.value
else:
  self.reward=0
  self.timesincelastclearpos+=1
self.reward=self.value-unrealizedlastvalue-max((self.value-unrealizedlastvalue)
if np.isnan(self.reward):
  self.reward=0
  print("Nan Reward")
self.rewardlist.append(self.reward)
```

```
    stateseries=pd.Series({"deltatime":datatick["deltatime"],"midprice":self.midpric
    obs=pd.concat([stateseries,datatick[[c for c in datatick.index if not c in ["tim
    return obs,self.reward,done,{}

  def position_penalty(self,position):
    return self.positionpenalty*np.log(1+np.absolute(position))

  def Deal(self,direction,dealprice,dealvolume):
    if direction=="Bid" and dealvolume>0:
      #print("Deal: Bid",dealvolume,"At",dealprice)
      self.position+=dealvolume/dealprice
      self.cash-=dealvolume
    if direction=="Ask" and dealvolume>0:
      #print("Deal: Ask",dealvolume,"At",dealprice)
      self.position-=dealvolume/dealprice
      self.cash+=dealvolume
    return dealvolume



  def render(self, mode='human', close=False):
    pass



  def seed(self, seed=None):
    self.np_random, seed1 = seeding.np_random(seed)
    seed2 = seeding.hash_seed(seed1 + 1) % 2 ** 31
    return [seed1, seed2]



  def plotvalues(self):
    if self.outputplot:
      try:
        print("Final Value:",self.value," Sharpe Value:",self.value/np.std(self.val
      except:
        pass
```

```
        fig ,ax=plt . subplots ()
        lns1=ax . plot ( self . valuelist , label="Value" )
        lns2=ax . plot (np . array ( self . positionlist )*100 , label="Position" )
        lns3=ax . plot (np . array ( self . rewardlist )*10 , label="Reward" )
        plt . legend ( loc="upperleft" )
        ax2=ax . twinx ()
        lns4=ax2 . plot ( self . dataslice [ "midprice" ] . to_numpy () , "r−" , label="Price" )
        lns = lns1+lns2+lns3+lns4
        labs = [ l . get_label () for l in lns ]
        ax . legend ( lns , labs )
        plt . show ()

#envcolumns=["midprice"]+["time","deltatime"]+["midprice_−"+str(x+1) for x in range(t
envcolumns=all_data_copy . columns . drop ( [ "index" , "date" ,"sym" ]+[ "LSTM_"+str ( j ) for j in
#envcolumns=["midprice"]+["time","deltatime"]+["midprice_−"+str(x+1) for x in range(t

params = HYPERPARAMS[ 'stockenv ' ]
env=StocksEnv ( data=all_data_copy [ envcolumns ] )
env . reset ()

env . step ( ( 0.5 ,0.5 ,100 ,100) ) [ 0 ]

env . observation_space

class ChangeObsWarpper (gym . Wrapper ) :
  def __init__ ( self , env=None ) :
    super () . __init__ ( env )
    self . singleobsshape=self . observation_space . shape [ 0 ] −1
    self . observation_space=gym . spaces . Box( low=np . zeros ( ( self . singleobsshape*params [ "t
  def updateobs ( self , obs ) :
    obs . drop ( [ "midprice" ] , inplace=True )
    obs [ [ x for x in obs . index if "Size" in x ] ]/=100000
    obs [ [ x for x in obs . index if "RSI" in x ] ]−=50
    return np . array ( obs , dtype=np . float64 )
```

```
  def reset(self):
    self.obsarray=np.zeros((params["ticks_to_look_back"],self.singleobsshape))
    obs=self.updateobs(self.env.reset())
    self.obsarray=np.concatenate((obs.reshape(1,-1),self.obsarray[:-1]))
    return self.obsarray.reshape(-1)
  def step(self,action):
    obs,reward,done,info=self.env.step(action)
    obs=self.updateobs(obs)
    self.obsarray=np.concatenate((obs.reshape(1,-1),self.obsarray[:-1]))
    return self.obsarray.reshape(-1),reward,done,info

class DiscreteActionWrapper(gym.Wrapper):
  def __init__(self,env=None):
    super().__init__(env)
    self.action_space=gym.spaces.Discrete(7)

  def reset(self):
    obs=self.env.reset()
    self.actionlist=[]
    return obs

  def step(self,action):
    action=np.clip(action,0,6)
    self.actionlist.append(action)
    if action==0:
      obs,reward,done,info= self.env.step((0.5,0.5,100,100))
    elif action==1:
      obs,reward,done,info= self.env.step((1,1,100,100))
    elif action==2:
      obs,reward,done,info= self.env.step((2,2,100,100))
    elif action==3:
      obs,reward,done,info= self.env.step((0.5,2,100,100))
    elif action==4:
      obs,reward,done,info= self.env.step((2,0.5,100,100))
    elif action==5:
```

```
        obs,reward,done,info= self.env.step((1,1.5,100,100))
    elif action==6:
        obs,reward,done,info= self.env.step((1.5,1,100,100))

    #elif action==9:
    #   obs,reward,done,info= self.env.step((-100,-100,-min(self.position*self.midprice
    if done:
        self.plotvalues()
    return obs,reward,done,info
  def plotvalues(self):
    if self.env.outputplot:
        self.env.plotvalues()
        plt.scatter(range(len(self.actionlist)),self.actionlist ,alpha=0.5,marker="*")
    plt.show()

params = HYPERPARAMS['stockenv']
env=StocksEnv(data=all_data_copy[envcolumns])
env=DiscreteActionWrapper(env)
env=ChangeObsWarpper(env)
env.set_datalength(1000)
obs=env.reset()
done=False
ct=time.time()
obslist=[obs]
while not done:
  obs,reward,done,_=env.step(np.random.randint(env.action_space.n))
  obslist.append(obs)
print("Speed: ",env.datalength/(time.time()-ct)," F/s")

obstransformer=StandardScaler()
obstransformer.fit(np.array(obslist))

class ObsTransformWarpper(gym.Wrapper):
  def __init__(self,env=None,obstransformer=None):
    super().__init__(env)
```

```
    self.obstransformer=obstransformer
  def updateobs(self,obs):
    return obstransformer.transform([obslist[0]])[0]
  def reset(self):
    return self.updateobs(self.env.reset())
  def step(self,action):
    obs,reward,done,info=self.env.step(action)
    return self.updateobs(obs),reward,done,info

env=ObsTransformWarpper(env,obstransformer)

env.observation_space

try:
  del env
  del all_data
except:
  pass

"""#3. DQN Agent"""

class NoisyLinear(nn.Linear):

    def __init__(self, in_features, out_features, sigma_init=0.017, bias=True):
        super(NoisyLinear, self).__init__(in_features, out_features, bias=bias)
        self.sigma_weight = nn.Parameter(torch.full((out_features, in_features), sigm
        self.register_buffer("epsilon_weight", torch.zeros(out_features, in_features))
        if bias:
            self.sigma_bias = nn.Parameter(torch.full((out_features,), sigma_init))
            self.register_buffer("epsilon_bias", torch.zeros(out_features))
        self.reset_parameters()

    def reset_parameters(self):
        std = math.sqrt(3 / self.in_features)
        self.weight.data.uniform_(-std, std)
```

```
        self.bias.data.uniform_(-std, std)

    def forward(self, input):
        self.epsilon_weight.normal_()
        bias = self.bias
        if bias is not None:
            self.epsilon_bias.normal_()
            bias = bias + self.sigma_bias * self.epsilon_bias.data
        return F.linear(input, self.weight + self.sigma_weight * self.epsilon_weight.d

class A2CNetwork(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(A2CNetwork, self).__init__()

        linearlayer=nn.Linear

        self.common = nn.Sequential(
            linearlayer(input_shape[0], 256),
            nn.LeakyReLU(),
        )

        self.policy = nn.Sequential(
            linearlayer(256, 128),
            nn.LeakyReLU(),
            linearlayer(128, n_actions)
        )

        self.value = nn.Sequential(
            linearlayer(256, 64),
            nn.LeakyReLU(),
            linearlayer(64, 1)
        )


    def forward(self, x):
```

```
        fx = torch.tensor(x.float()).to(torch.device(device))
        common_out = self.common(fx).view(fx.size()[0], -1)
        return self.policy(common_out), self.value(common_out)



def unpack_batch(batch, net, device=device):
    states = []
    actions = []
    rewards = []
    not_done_idx = []
    last_states = []
    for idx, exp in enumerate(batch):
        states.append(np.array(exp.state, copy=False))
        actions.append(int(exp.action))
        rewards.append(exp.reward)
        if exp.last_state is not None:
            not_done_idx.append(idx)
            last_states.append(np.array(exp.last_state, copy=False))
    states_v = torch.FloatTensor(states).to(device)
    actions_t = torch.LongTensor(actions).to(device)

    # handle rewards
    rewards_np = np.array(rewards, dtype=np.float32)
    if not_done_idx:
        last_states_v = torch.FloatTensor(last_states).to(device)
        last_vals_v = net(last_states_v)[1]
        last_vals_np = last_vals_v.data.cpu().numpy()[:, 0]
        rewards_np[not_done_idx] += params["gamma"] ** params["REWARD_STEPS"] * last_v

    ref_vals_v = torch.FloatTensor(rewards_np).to(device)
    return states_v, actions_t, ref_vals_v

class LRSchedualr(object):
    def __init__(self, optimizer):
        self.optimizer = optimizer
```

```
        self.initial_lr=self.optimizer.param_groups[0]["lr"]
        self._lr=self.initial_lr
        self.framecount=0

    def step(self, loss, epoch=None):
        self.framecount+=1
        current = float(loss)
        self._lr=max(self._lr*params['learning_rate_decay'],params['min_learning_rate
        for j, param_group in enumerate(self.optimizer.param_groups):
          param_group['lr']=self._lr

    def get_lr(self):
      return [self._lr]

def default_states_preprocessor(states):
    if len(states) == 1:
        np_states = np.expand_dims(states[0], 0)
    else:
        np_states = np.array([np.array(s, copy=False) for s in states], copy=False)
    return torch.tensor(np_states)


class ProbabilityActionSelector(ptan.actions.ActionSelector):
    def __call__(self, probs,epsilon):
        assert isinstance(probs, np.ndarray)
        actions = []
        for prob in probs:
            prob=np.nan_to_num(prob)
            p=prob*(1-epsilon)+epsilon/(prob.shape[0])
            try:
              actions.append(np.random.choice(len(prob), p=p))
            except:
              actions.append(np.random.choice(len(prob)))
              #print("Bad Probability",prob)
        return np.array(actions)
```

```
class ActorCriticAgent(ptan.agent.BaseAgent):
    def __init__(self, model, action_selector=ProbabilityActionSelector(), device=devi
                apply_softmax=False, preprocessor=default_states_preprocessor):
        self.model = model
        self.action_selector = action_selector
        self.device = device
        self.apply_softmax = apply_softmax
        self.preprocessor = preprocessor
        self.probslist=[]
        self.epsilon=params["epsilon_start"]
    @torch.no_grad()
    def __call__(self, states, agent_states=None):

        if self.preprocessor is not None:
            states = self.preprocessor(states)
            if torch.is_tensor(states):
                states = states.to(self.device)
        probs_v, values_v = self.model(states)
        if self.apply_softmax:
            #probs_v = F.softmax(torch.clamp(probs_v,-10,20) ,dim=1)
            probs_v = F.softmax(probs_v ,dim=1)
        probs = probs_v.data.cpu().numpy()
        self.probslist.append(probs)
        actions = self.action_selector(probs,self.epsilon)
        agent_states = values_v.data.squeeze().cpu().numpy().tolist()
        return np.array(actions), agent_states

    def updateeps(self,step_idx):
       self.epsilon=np.interp(step_idx,[0]+params["epsilon_final_frame"],[params["epsi

    def plotprobs(self):
       self.probslist=np.array(self.probslist)
       for i in range(self.probslist.shape[2]):
```

```
        for j in range(params["NUM_ENVS"]):
          if j==params["NUM_ENVS"]-1:
            plt.plot(self.probslist[:,j,i],alpha=0.9,label="Env"+str(j)+" , "+"Action
          else:
            plt.plot(self.probslist[:,j,i],alpha=0.9)
        plt.legend()
        plt.show()
      self.probslist=[]


class TBMeanTracker:
    """
    TensorBoard value tracker: allows to batch fixed amount of historical values and
    Designed and tested with pytorch-tensorboard in mind
    """
    def __init__(self, writer, batch_size):
        """
        :param writer: writer with close() and add_scalar() methods
        :param batch_size: integer size of batch to track
        """
        assert isinstance(batch_size, int)
        assert writer is not None
        self.writer = writer
        self.batch_size = batch_size


    def __enter__(self):
        self._batches = collections.defaultdict(list)
        return self


    def __exit__(self, exc_type, exc_val, exc_tb):
        self.writer.close()


    @staticmethod
    def _as_float(value):
        assert isinstance(value, (float, int, np.ndarray, np.generic, torch.autograd.
        tensor_val = None
```

```
        if isinstance(value, torch.autograd.Variable):
            tensor_val = value.data
        elif torch.is_tensor(value):
            tensor_val = value

        if tensor_val is not None:
            return tensor_val.float().mean().item()
        elif isinstance(value, np.ndarray):
            return float(np.mean(value))
        else:
            return float(value)

    def track(self, param_name, value, iter_index):
        assert isinstance(param_name, str)
        assert isinstance(iter_index, int)

        data = self._batches[param_name]
        data.append(self._as_float(value))

        if len(data) >= self.batch_size:
            self.writer.add_scalar(param_name, np.mean(data), iter_index)
            data.clear()



class RewardTracker:
    def __init__(self, stop_reward):
        self.stop_reward = stop_reward

    def __enter__(self):
        self.ts = time.time()
        self.ts_frame = 0
        self.total_rewards = []
        return self
```

```
    def __exit__(self, *args):
        pass

    def reward(self, reward, frame, lr,eps):
        self.total_rewards.append(reward)
        speed = (frame - self.ts_frame) / (time.time() - self.ts)
        self.ts_frame = frame
        self.ts = time.time()
        mean_reward = np.mean(self.total_rewards[-25:])
        print("%d: done %d epochs, last reward %.3f, mean reward %.3f, learning rate %
            frame, len(self.total_rewards), self.total_rewards[-1], mean_reward,lr,eps
        ))
        if mean_reward > self.stop_reward:
            print("Solved in %d frames!" % frame)
            return True
        return False




def set_seed(seed, envs=None, cuda=False):
    print("Seed: ",seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if cuda:
        torch.cuda.manual_seed(seed)

    if envs:
        for idx, env in enumerate(envs):
            env.seed(seed + idx)

try:
  del Trainer
  print("Deleted Last Trainer")
except:
```

```
    pass

def make_env(outputplot=False):
    env=StocksEnv(data=all_data_copy[envcolumns],randomdata=True)
    env.outputplot=outputplot
    env=DiscreteActionWrapper(env)
    env=ChangeObsWarpper(env)
    #env=ObsTransformWarpper(env,obstransformer)
    env.set_datalength(3000)
    #env=gym.make("CartPole-v1")
    return env

class dqnAgentTrainer:
    def __init__(self):
        self.envs = [make_env() for _ in range(params["NUM_ENVS"]-1)]+[make_env(True)]
        set_seed(int(time.time()*17)%1013, self.envs, cuda=True)

        self.dotest=False
        self.net = A2CNetwork(self.envs[0].observation_space.shape, self.envs[0].action_sp
        print(self.net)
        self.agent= ActorCriticAgent(self.net, apply_softmax=True, device=device)
        self.exp_source = ptan.experience.ExperienceSourceRollouts(self.envs, self.agent,
        self.optimizer = optim.Adam(self.net.parameters(), lr=params['learning_rate'])
        self.lrs=LRSchedualr(self.optimizer)
        self.losslist=[]

    def train(self):
        print("Traing Start Time:",time.ctime())
        print("------------------------")
        step_idx = 0
        self.writer = SummaryWriter(comment="-stockenv-v7-")
        with TBMeanTracker(self.writer, batch_size=10) as tb_tracker:
            with RewardTracker(params['stop_reward']) as tracker:
                for mb_states, mb_rewards, mb_actions, mb_values in self.exp_source:
                    #print(mb_states, mb_rewards, mb_actions, mb_values)
```

```
new_rewards = self.exp_source.pop_total_rewards()
if new_rewards:
  self.total_rewards=tracker.total_rewards
  #if len(self.total_rewards)%10==0:
  #  display.clear_output()
  if len(self.total_rewards)%params["DATALENGTH_INCREASE_FREQ"]==0:
    for env in self.envs:
      env.set_datalength(env.datalength+1000)
  print("Total Loss: {:.4f}, Policy Loss: {:.4f}, Entropy Loss: {:.4f} * 
  self.plotloss()
  try:
    print("Mean PnL:",np.mean(envvaluelist),"Std PnL:",np.std(envvaluelist
    #self.agent.plotprobs()

  except:
    pass
  self.agent.probslist=[]
  if tracker.reward(np.mean(new_rewards), step_idx, self.lrs._lr,self.agen
    break
  print("------------------------")

else:
  try:
    envvaluelist=[env.value for env in self.envs]
  except:
    pass

self.optimizer.zero_grad()
if np.any(np.isnan(mb_rewards)) or np.any(np.isnan(mb_values)):
  continue
with autograd.detect_anomaly():
  states_v = torch.FloatTensor(mb_states).to(device)
  mb_adv = mb_rewards - mb_values
  adv_v = torch.FloatTensor(mb_adv).to(device)
  actions_t = torch.LongTensor(mb_actions).to(device)
```

```
    vals_ref_v = torch.FloatTensor(mb_rewards).to(device)

    logits_v, value_v = self.net(states_v)
    loss_value_v = F.mse_loss(value_v.squeeze(-1), vals_ref_v)

    log_prob_v = F.log_softmax(logits_v, dim=1)
    log_prob_actions_v = adv_v * log_prob_v[range(len(mb_states)), actions_t
    loss_policy_v = -log_prob_actions_v.mean()

    prob_v = F.softmax(logits_v, dim=1)
    entropy_loss_v = (prob_v * log_prob_v).sum(dim=1).mean()

    # apply entropy and value gradients
    entropy_beta=np.interp(step_idx,params["ENTROPY_BETA_FRAME"],params["EN
    loss_v = loss_policy_v + entropy_beta * entropy_loss_v + loss_value_v
    loss_v.backward()
    nn_utils.clip_grad_norm_(self.net.parameters(), params["CLIP_GRAD"])
    self.optimizer.step()
    self.lrs.step(loss_v.item())
  self.losslist.append([loss_v.item(), loss_policy_v.item(),entropy_beta * e
  step_idx += params["NUM_ENVS"] * params["REWARD_STEPS"]
  self.agent.updateeps(step_idx)

  grads = np.concatenate([p.grad.data.cpu().numpy().flatten()
              for p in self.net.parameters()
              if p.grad is not None])

  tb_tracker.track("advantage",       adv_v, step_idx)
  tb_tracker.track("values",          value_v, step_idx)
  tb_tracker.track("batch_rewards",   vals_ref_v, step_idx)
  tb_tracker.track("loss_entropy",    entropy_loss_v, step_idx)
  tb_tracker.track("loss_policy",     loss_policy_v, step_idx)
  tb_tracker.track("loss_value",      loss_value_v, step_idx)
  tb_tracker.track("loss_total",      loss_v, step_idx)
  tb_tracker.track("grad_l2",         np.sqrt(np.mean(np.square(grads))), st
```

```
            tb_tracker.track("grad_max",          np.max(np.abs(grads)), step_idx)
            tb_tracker.track("grad_var",          np.var(grads), step_idx)

        print(time.ctime())

    def plotloss(self):
        losslist=np.array(self.losslist[1:])
        self.losslist=[]
        fig,ax=plt.subplots()
        lns1=ax.plot(losslist[:,1],label="Policy Loss (left)")
        lns2=ax.plot(losslist[:,2],label="Entropy Loss (left)")
        quantiles=np.quantile(losslist ,[0.1,0.9],axis=0)
        #print(quantiles)
        ax.set_ylim((min(quantiles[0,1],quantiles[0,2])-0.1,max(quantiles[1,1],quantiles[1
        ax2=ax.twinx()
        lns3=ax2.plot(losslist[:,3],"r-",label="Value Loss (right)")
        ax2.set_ylim((0,quantiles[1,3]))
        lns = lns1+lns2+lns3
        labs = [l.get_label() for l in lns]
        ax.legend(lns, labs)
        plt.show()

params = HYPERPARAMS['stockenv']
try:
    del Trainer
    print("Deleted Last Trainer")
except:
    pass
Trainer=dqnAgentTrainer()
Trainer.dotest=False
print(params)

#downloaded = drive.CreateFile({'id':"1KH9Oulw_MyR4bT74uymA0mqQvAdVxN5t"})
#downloaded.GetContentFile('TrainedNet.pth')
```

```
uploaded = files.upload()

state_dict = torch.load(list(uploaded.keys())[0])
Trainer.net.load_state_dict(state_dict)

# Commented out IPython magic to ensure Python compatibility.
!rm -rf ./runs/
!kill 6481
os.makedirs(logs_base_dir, exist_ok=True)
# %tensorboard --logdir {logs_base_dir}

params = HYPERPARAMS['stockenv']
Trainer.train()

params

probslist=[]
valuelist=[]
actionselector=ProbabilityActionSelector()
for envcount in range(100):
  print("Env No.",envcount)
  env=make_env(True)
  env.set_datalength(10000)
  obs=env.reset()
  done=False
  while not done:
    probs,values=Trainer.net(torch.FloatTensor([obs]))
    probs=scipy.special.softmax(probs.detach().cpu().numpy())
    probslist.append(probs)
    action=actionselector(probs,0)
    obs,reward,done,info=env.step(action)
  valuelist.append(env.value)
  for i in range(probslist[0].shape[1]):
    plt.plot(scipy.special.softmax(np.array(probslist),axis=2)[:,0,i],label="Action"+s
  plt.legend()
```

```
    plt.show()
    probslist=[]
    del env

sns.distplot(valuelist)
print("Mean PnL:",np.mean(valuelist),", Std PnL:",np.std(valuelist))

torch.save(Trainer.net.state_dict(), 'TrainedNet.pth')
files.download('TrainedNet.pth')

scipy.special.softmax(probs.detach().cpu().numpy())[0]

probs.shape[0]
```

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[2] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[3] Richard S. Sutton. Learning to predict by the methods of temporal differences. 10.1007/BF00115009.

[4] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[5] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration, 2017.

[6] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2015.

[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.

[8] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning, 2017.

[9] Merton Miller Sanford Grossman. Liquidity and market structure. The Journal of Finance, Vol 43, No. 3.

[10] Jose Penalva Alvaro Cartea, Sebastian Jaimungal. *Algorithmic and High-Frequency Trading*. Cambridge University Press.

[11] Olivier Gueant. *The Financial Mathematics of Market Liquidity*. CRC Press.

[12] Rahul Savani Andreas Koukorinis Thomas Spooner, John Fearnley. Market making via reinforcement learning. Preprint, arXiv:1804.04216.

[13] Jonathan Sadighian. Deep reinforcement learning in cryptocurrency market making. Preprint, arXiv:1911.08647, 2019.

[14] Jianfeng Gao Xiaodong He Jianshu Chen Li Deng Ji He Xiujun Li1, Lihong Li. Recurrent reinforcement learning: A hybrid approach. Preprint, arXiv:1509.03044.

[15] Maxim Lapan. *Deep Reinforcement Learning Hands-On.* Packt Publishing.

[16] Fernando Doglio. *Mastering Python High Performance.* Packt Publishing.