MATH 527: Final Project

Market Model Deep Q-Learning Project

Letting the Computer Do the Thinking

Ben Barber
bbarber@hawk.iit.edu
A20478568

Ivan Gvozdanovic
@hawk.iit.edu
A20465207

Noel Nathan
nnathan2@hawk.iit.edu
A20450675

Farukh Sarkulov
fsarkulov@hawk.iit.edu
A20474316

September 3, 2023

Abstract

This project applies deep Q-learning to a specific market model. The market model has specific features as we'll outline later. The market model is the environment in which we will train an agent to take the optimal action. This project is done in three parts: market model implementation and analysis, deep Q learning network construction and analysis, and deriving the optimal policy from our deep Q learning results.

Contents

1 Background and Theory

1.1 Market Model

The base of our market model is $d > 0$ stock assets, without the typical presence of an interest-bearing banking account. Each asset has an associated price history $s^i = (s_t^i)_{t=0}^T; s_t^i \in \mathbb{R}_{\geq 0} \forall t \in \{0, 1, \ldots, T\}$ (and thus the state of the market can be represented by the time series $\vec{\mathbf{s}} = (\vec{\mathbf{s}}_t)_{t=0}^T; \vec{\mathbf{s}}_t \in \mathbb{R}_{\geq 0}^d \forall t \in \{0, 1, \ldots, T\}$), and at time 0, all stock prices are equal. Our portfolio between time t and $t+1$ is represented by $\vec{\mathbf{w}}_t \in \Delta^d := \{x \in \mathbb{R}^d : x^i \geq 0 \text{ and } \sum_{i=1}^d x^i = 1\} \forall t \in \{0, 1, \ldots, T-1\}$, thus forming time series $w = (\vec{\mathbf{w}}_t)_{t=0}^{T-1}$. At time t, we rebalance from strategy $\vec{\mathbf{w}}_{t-1}$ to $\vec{\mathbf{w}}_t$ subject to the self-financing constraint $0 = \sum_i s_t^i (w_t^i - w_{t-1}^i)$. We relate our weights in each each stock at time t, w_t^i, to the position in each stock at time t, N_t^i, via $w_t^i P_t = N_t^i S_t^i$, where P_t is the portfolio value at time t.

Our stock prices follow

$$\vec{\mathbf{S}}_{t+1} - \vec{\mathbf{S}}_t = \vec{\boldsymbol{\mu}} + \vec{\boldsymbol{\kappa}}(\Delta \vec{\mathbf{N}}_t) + \mathbf{M}\vec{\boldsymbol{\xi}}_t \tag{1}$$

where $\vec{\mathbf{S}}_t = \ln(\vec{\mathbf{s}}_t)$, μ^i is the drift of the ith asset, $\vec{\boldsymbol{\kappa}}$ is the market impact with form $\kappa_i(\vec{\mathbf{x}}) = c_i \text{sgn}(x_i)\sqrt{x_i}$, $\mathbf{M}$ is a low-rank matrix which can either be thought of as a factor-impact matrix (especially in the case where $\mathbf{M}$ is not square) or as related to the covariance matrix of the returns (only in the case $\mathbf{M}$ is square), and $\vec{\boldsymbol{\xi}}_t$ is the driving noise of the dynamical behavior. We require that the ξ_t^is be I.I.D., with $\mathbb{E}[\xi_t^i] = 0$ and $\mathbb{V}[\xi_t^i] = 1$. We define the asset-wise return $r_t^i = \frac{s_t^i - s_{t-1}^i}{s_{t-1}^i}$, and define the one-period portfolio return as

$$R_t = 1 + \vec{\mathbf{r}}_t \cdot \vec{\mathbf{w}}_{t-1} - \eta \left(\left| \Delta \vec{\mathbf{N}}_{t-1} \right| \cdot \mathbb{1} \right) \tag{2}$$

, where η is the market friction of trading costs, and

$$R_t^{\text{cum}} = \prod_{s=1}^{t} R_s \tag{3}$$

, and $P_t = R_t^{\text{cum}} P_0$.

1.2 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning technique that uses interactions between an agent and the environment to gain maximum rewards over a period of time [4]. This ability to learn from feedback opens up the possibility to overcome the shortcomings of supervised and unsupervised learning. Combined with deep learning neural networks, RL continues to grow into a widely applicable science in numerous domains, such as robotics, gaming, healthcare, and finance, to name a few. The optimization achieved through RL and deep learning has made it possible for an artificial agent to interact similarly to a biological agent in many sequential decision-making problems [3].

We will first highlight how RL is different from supervised and unsupervised learning in a few aspects, namely in the dynamic wealth management (DWM) context. Similar to supervised and unsupervised learning, there is a teacher present in RL. However, the teacher in RL only provides a partial feedback, in the form of rewards [2]. The teacher does not explicitly lead the agent toward the actions that produce the highest rewards, which in this case is the cumulative portfolio return.

This leads to the next difference, which is the presence of a feedback loop, where the state of the environment is dependent on the agent's action. This action-reward relationship in RL means that the agent has to learn good behaviors incrementally, which requires planning [2]. In DWM, we seek to maximize the long-term cumulative portfolio returns (reward) by choosing the optimal portfolio weights (action) for each state by solving the Bellman optimality equation.

Another difference is the presence of the exploration-exploitation dilemma. As mentioned earlier, the agent is not explicitly directed at the maximum reward. The agent is then tasked with exploring a variety of actions, and then exploiting the chosen actions [2]. In exploration, the agent takes random actions with a probability of ϵ to try and find a path towards greater rewards. But with probability $1 - \epsilon$ the agent can choose to take the action they know provides a substantial reward thereby "exploiting" their knowledge of the environment, state, and action space.

For instance, in DWM, an agent may have gone through and invested through the portfolio's holding period and increased the overall cumulative portfolio return based on different state-action pairs. However, the cumulative return is not guaranteed to be the maximal possible cumulative portfolio return and thus, there is incentive to run the simulation again in hopes of a larger cumulative return. As such, the agent must find equilibrium between

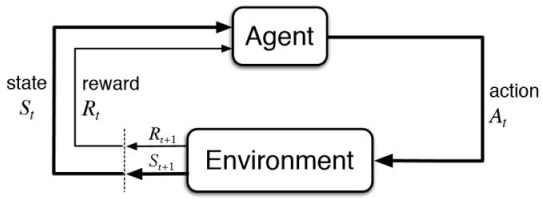

Figure 1: An image of a RL framework ([1])

exploration and exploitation.

The RL framework consists of:

- An environment - data evolves as RL progresses and is characterized by its state at time t, S_t
- A policy $\pi_t(S_t)$ - ruleset that dictates how the agent will act given the environment state S_t
- Value Function $V_\pi(S_t)$ - quantifies the "attractiveness" of a given state S_t for the agent
- Rewards $r_t(S_t, A_t)$ - determines the goal of the RL problem

1.3 Bellman Equation

We know that an agent learns good behaviors through a feedback loop and thereby obtaining the highest reward possible. In order to proceed with the next action, the agent must be equipped with a memory of what it has previously learned. This is where the Bellman Equation becomes an integral part of RL.

The Bellman Equation decomposes value functions into two parts, the immediate reward and the discounted future value function [7]. Value function here represents the evaluation of a given state of the environment, i.e. how good the state is for the agent.

The Bellman Equation is as follows:

$$V_t^\pi(s) := E_t^\pi[R_t(s, a, s^{'})] + \gamma E_t^\pi[V_{t+1}^\pi(s^{'})] \tag{4}$$

1.4 Markov Decision Process (MDP)

The Markov property implies that the probability of the future is independent of the past, given the present [3]. This means that the future of the process in RL is solely dependent on the current observation and not the full history. The MDP is chosen for RL problems because it represents the interaction between agent and environment by passing messages for three signals: a signal of actions for the agent, a signal for the state of the environment, and a signal defining the reward, i.e the goal of the problem [2].

The MDP describes control variables that help direct an agent toward the maximum reward.

There are five components of MDPs, which are:

- Set of States : S_t ,t
- Set of Actions : A_t
- Transition Probability Function :

$$T(s'|s,a) := P[S_{t+1} = s'|S_t = s, A_t = a] \tag{5}$$

- Discount Factor for Future Rewards: γ
- Reward Function:

$$r(s,a) = E[R_{t+1}|S_t = s, A_t = a] \tag{6}$$

$$R_{t+1} = R(S_{t+i}, a_{t+i}, S_{t+i+1}) \tag{7}$$

We can denote discounted rewards for convenience as:

$$G_t = \sum_{i=0}^{T-t-1} \gamma^i R_{t+1} \tag{8}$$

Using our discounted rewards we define our state value function which measures the "attractiveness" of a state to the agent based on potential rewards:

$$V_\pi(s) := E_\pi[G_t|S_t = s] \tag{9}$$

$V_\pi(s)$ is our expected return of discounted future rewards G_t under a stochastic policy defined

as:

$$\pi(a|s) := P[A_t = a|S_t = s] \tag{10}$$

1.5 Q-Learning and Deep Q-Learning

Q-learning is a value-based method of RL implementation where an agent learns over time by using rewards to decide on the best action in a given environment. Q-learning works to inform an agent which action to take by using a Q-table, which is a data structure that calculates the maximum expected rewards of every action at every step. The rewards generated from the sequence of actions are recorded as Q-values and they are iteratively updated, using the Bellman Equation, as the agent explores and exploits the environment, until it reaches convergence.

We define Q-values as:

$$Q_t(s, a) = r(s, a) + \gamma \max_a Q(s^{'}, a) \tag{11}$$

The Bellman Equation is integral for this purpose because it calculates the value functions required for the Q-learning method. The Bellman Equation in the Q-learning Update context is as follows:

$$Q_t(s, a) \leftarrow Q_t(s, a) + \alpha[R_t(s, a, s^{'}) + \gamma \max_a Q_{t+1}(s^{'}, a^{'}) - Q_t(s, a)] \tag{12}$$

In this equation,α determines the learning rate that controls the convergent speed of the updating process while adjusting γ increases or decreases the contribution of future rewards

Deep Q-learning is an extension of Q-learning. It is used to handle environments that include continuous state spaces, where the space can be of a large dimension and occur over a continuous time interval. In Deep Q-learning, the Q-values are approximated using a neural network, the deep Q-network (DQN):

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{13}$$

The following are the steps involved in implementing DQNs (Liu, 2020):

- Initialize two neural networks randomly: Target Network $Q(s^{'}, a^{'}; \theta^{'})$ and Prediction Network $Q(s, a; \theta)$ using $\theta = \theta^{'}$ and build an empty replay buffer

- Select an action using the ϵ-greedy method: with probability ϵ, select a random action. Otherwise, select the greedy action a: $\arg\max_a Q(s, a; \theta)$.
- After the chosen action is performed, the agent will receive reward r and move to a new state $s^{'}$.
- This transition is stored in the replay buffer as $< s, a, r, s' >$
- We then sample a random transition from buffer and calculate loss $\mathcal{L}$:

$$\mathcal{L} = (r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}; \theta^{'}) - Q(s, a; \theta))^2 \tag{14}$$

- Next, we perform stochastic gradient descent with respect to the neural network parameters to minimize $\mathcal{L}$
- After every nth epoch, the target network is overwritten by the current state of the prediction network
- Repeat for N episodes, tapering ϵ as the number of episodes increases

2 Method

2.1 Market Model

We implemented the market model as a class in Python using the `Numpy` library. The `MarketModel` class is iniatilized with `Numpy` arrays representing $\boldsymbol{\mu}$, $\mathbf{c}$ (the market impact factors), and $\mathbf{M}$, and scalars representing T (the maximum time duration of a single epsiode) and η. The `MarketModel` advances the state of the market one step in time when a trading strategy $\vec{\mathbf{w}}_t$ is supplied as a `Numpy` array.

2.2 Agent

The trading strategy is implemented by an object of the `agent` class. The *textttagent* is initialized with a policy, represented by a function that takes the current market state as an input. The `agent` is assumed to be following an ϵ-greedy policy, and thus the `agent` generates an action when supplied with both ϵ and the market state s. The agent generates

a random action[1] with probability ϵ and follows the current policy with probability $1-\epsilon$. The `agent` allows for updating the policy to change the trading strategy between episodes.

2.3 Deep Q-Learning

To help debug our training regime, we use a highly constrained market model, consisting of two stocks: stock 1: positive drift and low variance, and stock 2: 0 drift and high variance. This is represented by choosing $\vec{\boldsymbol{\mu}} = \begin{pmatrix} 0.1 \\ 0 \end{pmatrix}$ and $\mathbf{M} = \begin{pmatrix} 0.01, 0.01 \\ 0.1, 0.1 \end{pmatrix}$. We also set $\eta = 0$ and $\vec{\mathbf{c}} = \vec{\mathbf{0}}$ to represent a friction-free market without market impact. This second condition allows us to benchmark our trading strategy against the known efficiency frontier of the Markowitz portfolio strategy [6]:

$$\begin{aligned} \mathbb{E}[\vec{\mathbf{a}}_t] &= w^1\mu^1 + w^2\mu^2 \\ &= \mu^2 + w^1(\mu^1 - \mu^2) \qquad (15) \\ \mathbb{V}[\vec{\mathbf{a}}] &= w^1\sigma_1^2 + w^2\sigma_2^2 + 2w^1w^2\mathrm{Cov}(1,2) \qquad (16) \end{aligned}$$

Combining our two constraints allows us to further define the efficiency frontier, as $\mathrm{Cov}(1,2) = \sigma_1\sigma_2$ in our model (the returns are completely covarying), and thus

$$\mathbb{E}[\vec{\mathbf{a}}] = w^1\mu^1 \qquad (17)$$

$$\mathbb{V}[\vec{\mathbf{a}}] = (w^1\sigma_1 + (1-w^1)\sigma_2)^2 = \sigma_1^2\left(w^1 + \frac{\sigma_2}{\sigma_1}(1-w^1)\right)^2 \qquad (18)$$

The actual Q-learning was implemented using networks generated by the `Keras` library. We initialized the target and prediction networks, then populated the replay buffer with transitions generated by running multiple episodes of the market. The agent in this initial run followed a flat-market strategy ($w_t^i = w_t^j \forall i,j \in \{1,\ldots,d\}$). Initially, $\epsilon = 0.5$. After populating the buffer, we selected a batch of experiences with which to train the prediction network. Periodically during model training, we update the target network with the current weights and biases of the prediction network. After the target number of training epochs has been reached, we update the policy of the agent using the prediction network, and generate new transitions that overwrite existing experiences in the replay buffer. This process of generating new experiences, training the prediction network, and updating the agent policy

[1]The random action is determined by creating an $\mathbb{R}^d$ vector by sampling $|\mathcal{N}(0,1)|$ d times, and then scaling that vector to conform to our condition that $\sum_i w_t^i = 1$

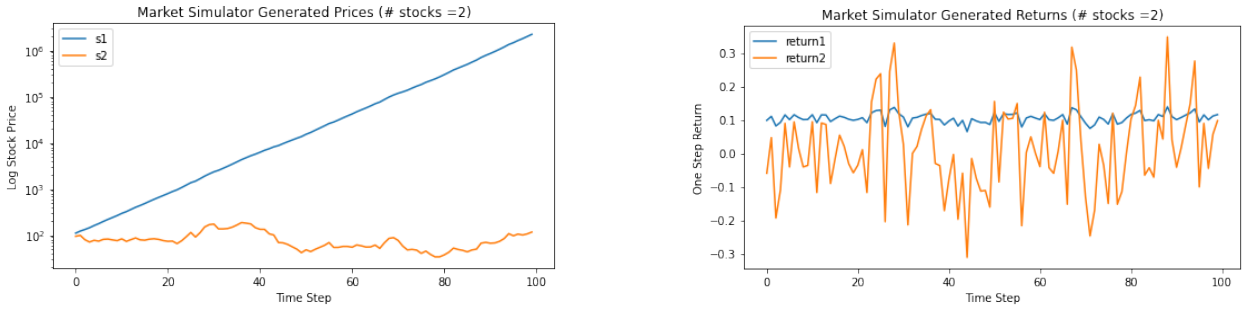

Figure 2: Stock Price Progression and Stock Returns

is continued until prediction network has converged to the desired tolerance.

3 Results

3.1 Market Model Output Analysis

The plots in 2 show the outputs of our Market Model under the training constraints. Based on the plots, we see the expected behavior: stock 1 has consistent, steady returns, while stock 2 has flat returns with high variance. This shows that our market model is at least performing as anticipated, under the exaggerated conditions of the training constraints. Table 1 shows the mean and variance for the returns of stock 1 and stock 2.

From the output of the Market Model we can get an idea of what we anticipate our deep Q learning favoring when we derive the optimal policy. Given that stock 1 increases exponentially, we would want to allocate most if not all of our portfolio to stock 1. If we define the rewards as cumulative returns then the deep Q agent should favor allocating more of the portfolio to stock 1. In the implementation of the Deep Q Learning networks we would hope that the final output would value portfolios that allocate more to stock 1 than stock 2, that is a portfolio with a majority allocated to stock 1 will have a more positive q value than a portfolio that favors stock 2.

	r_1	r_1
μ	0.10543	0.00999
σ	0.01439	0.13098

Table 1: Table of Mean and Variance for Returns of Stock 1 and Stock 2

3.2 Deep Q Learning Analysis

As expected, the convergence of the Q network to the optimal Q state-value function is highly unstable. In our experiments, we modified parameters such as episode number, training time (determines after which episode to begin training the network), epoch number, batch sizes, etc. The following is a list of parameters that worked very well on the test problem of only 2 stocks with one "good" and one "bad" stock:

$$\begin{aligned}
\text{training-time} &= 10 \\
\text{copy-weights-time} &= 20 \\
\text{exploration-cutoff-time} &= 30 \\
\text{episode-num} &= 100 \\
\text{episode-length} &= 50 - 70 \\
\text{epoch-num} &= 40 \\
\text{batch-size} &= 50 \\
\text{stock-num} &= 2
\end{aligned}$$

With these parameters, we obtain the following figure of the reward over the episodes. Al-

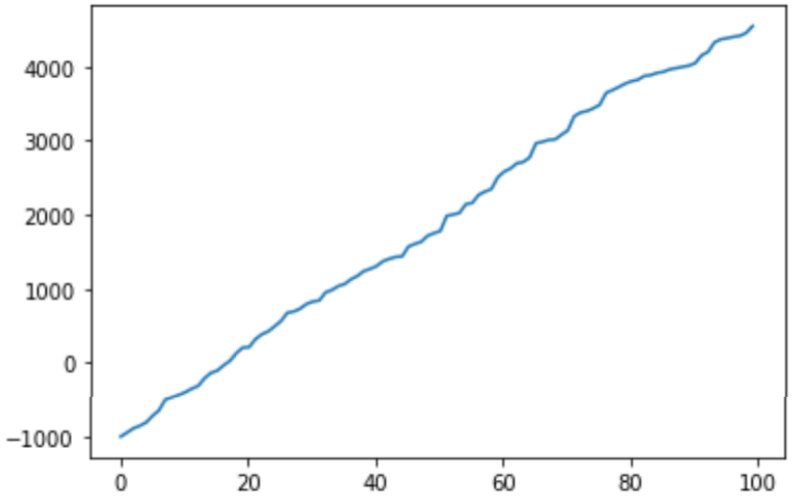

Figure 3: Reward over episodes

though the graph does not explicitly show the standard exploration-exploitation behaviour, it is indeed there. In order to test the convergence of the Q network, we used a "good" and a "bad" stock as mentioned above. However, the good stock, on average, positively outweighs the bad stocks in terms of the returns (rewards) and hence we see this almost monotonically

increasing pattern of the rewards.

However, this does not affect our findings. On the contrary, after applying **optimalAction** function on the trained network Q, we found that almost every time, the action that maximizes Q is the action (1,0); meaning the optimal action is to allocate all funds into the "good" stock. Moreover, the way the **optimalAction** works is that it takes gradients w.r.t to actions (Using TenserFlow's GradientTape) and keeps the state fixed. Testing the surface on multiple (s, a) pairs (keeping s fixed), the algorithm was almost always able to find the global maximum and minimum w.r.t action which are $(1, 0)$ and $(0, 1)$ respectfully. This implies that the surface learned by Q is smooth enough for our approach to work.

4 Conclusions and Further Work

In this project, we applied Deep Q-learning to the Market Model to derive Q values for given combinations of stock prices and allocations. We did this by using the derived Q values to determine an optimal action. In our market model, we chose to only have two stocks composed of one very well performing stock and one poorly performing stock. Our intuition was that the agent in the Deep Q Network (DQN) would accurately learn to favor the good stock at each choice of action. The results of the optimal action analysis verified and validated our confidence in the DQN. We fed a simple market model which assumes no transaction cost and market impact into the DQN to ensure that the DQN performed as we expected. In order to increase the practicality of this project in portfolio allocation, one useful approach is to increase the replay buffer size to improve memory retention. Another way is to incorporate other inventive architectures like actor-critic. We can also improve DQN by using Double DQN or Dueling DQN to help learning. In further applications, we plan to implement a market impact factor and transaction cost into the model to add complexity to the simulated market. We would also adjust the number of stocks to see how well the model reacts to a larger state-action space. Last but not least, we would need to run multiple simulations to choose optimal hyperparameters for the model.

References

[1] Bhatt, Shweta. "5 Things You Need to Know about Reinforcement Learning." KDnuggets. https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html.

[2] Dixon, Matthew F., et al. "Machine Learning in Finance: From Theory to Practice." Springer, 2020.

[3] François-Lavet, Vincent, et al. "An Introduction to Deep Reinforcement Learning." Foundations and Trends in Machine Learning, vol. 11, no. 3-4, 3 Dec. 2018, pp. 219–354., https://doi.org/10.1561/2200000071.

[4] Gourav, Kumar, and Dr. Amanpreet Kaur. "A Study of Reinforcement Learning Applications & Its Algorithms." International Journal of Scientific & Technology Research, vol. 9, no. 3, Mar. 2020, pp. 4223–4228.

[5] Liu, Chenyu. "Deep Reinforcement Learning and Electronic Market Making". Imperial College London, 2020, https://www.imperial.ac.uk/media/imperial-college/faculty-of-natural-sciences/department-of-mathematics/math-finance/Chenyu-Liu.pdf Accessed 26 Nov. 2021.

[6] Milken Institute. "Portfolio Optimization (Markowitz)". https://www.5minutefinance.org/concepts/mean-variance-portfolio-optimization. Accessed December 2, 2021.

[7] Tatsat, Hariom, et al. "Machine Learning and Data Science Blueprints for Finance: From Building Trading Strategies to Robo-Advisors Using Python." O'Reilly Media, Inc., 2020.