Risk-Averse Trading Agent Using Dueling DDQN with Penalized Reward

Jalil Douglas, Dailin Ji, Chenxi Liu

New York University, Tandon School of Engineering

I. Introduction

Machine Learning has been widely applied in modern finance, with one of the most popular areas being model-enabled autonomous trading. In this approach, models are trained on historical data to predict future prices and make trading decisions, without the need for human intervention. Among various types of models, Reinforcement Learning (RL) has a unique advantage due to its ability to make sequential decisions. However, RL agents often struggle to find a risk-neutral strategy in the high volatility of financial markets. In this paper, we experiment a heuristics method to address the issue. Specifically, we adjust the reward function to penalize investment actions with negative returns and high risk. By tuning the penalization factor, we train investment agents that are more conservative in making investment decisions. The effectiveness of our proposed method is validated through extensive experiments, demonstrating its potential in improving the performance of RL-based autonomous trading systems. The purpose of this study is to demonstrate the ability to interfere DQN agent by adjusting rewards, instead of to make a profitable trading strategy. The code are available at https://github.com/cl6530/DLfinalproject.

The structure of the paper is organized as follows: The subsequent section provides a concise overview of the existing literature, exploring the various applications of reinforcement learning (RL) in the field of trading. Section III primarily focuses on presenting the problem formulation and our methodology, encompassing the dueling network structure and the penalization factor. In Section IV, we demonstrate the effectiveness of trading simulators in facilitating our experiments, and present the numerical results obtained from our research. Finally, we conclude the project by summarizing our findings and suggesting potential future directions for further exploration.

II. Literature Review

Reinforcement Learning and DQN

Reinforcement learning (RL) is a subdomain of machine learning where an agent learns to make decisions by interacting with an environment to achieve a goal. As being reviewed by Li (2018), The agent selects actions based on its current state. The environment responds by providing a reward signal.The agent then chooses actions based on its present condition. Learning an ideal strategy that maximizes the cumulative reward over time is the goal of RL. The RL problem is formalized using a Markov Decision Process (MDP), which is characterized by states, actions, rewards, and transition probabilities that represent the dynamics of the environment.

Deep Q-Networks (DQN), introduced by Mnih et al. (2015), combine traditional Q-learning with deep neural networks. This combination makes it possible to learn more complex patterns and decision-making criterias, which will be particularly useful when dealing with high-dimensional state spaces and large-scale action sets. DQNs use a neural network to approximate the Q-function, which estimates the future rewards expectations of taking a specific action in a given state. By continuously updating the Q-function approximation, the agent learns to select actions that aim to maximize the cumulative rewards.

Reinforcement Learning in Trading

As concluded by Pricope (2021), being inspired by the excellent performance of Deep Reinforcement Learning (DRL) in complex games, DRL has shown great potential in stock trading that it could compete with experienced traders.

Under a financial context, RL may be used to simulate and optimize different elements of trading and investing, including portfolio management, algorithmic trading, and risk management. For instance, the states may be market data, financial indicators, or other pertinent information, and the agent could stand in for a trader or an algorithm. The actions could involve buying, selling, or holding, and the rewards can be based on the profit or loss resulting from these actions.

There are many studies involving DRL in stock trading studies. For critic-only deep reinforcement learning, Chen et al. (2019) tested a Deep Recurrent Q-Network (DRQN) variant, using a recurrent neural network at the base of DQN, improving temporal sequence processing, and a secondary target network for stability. The S&P500 ETF price history dataset is utilized for training and testing the agent against benchmark strategies like buy-and-hold and random action-selected DQN trading. The DRQN-based agent out-

performs the baselines, achieving an annual expected return of approximately 22-23%. For actor-only deep reinforcement learning, LSTM, Direct Deep Reinforcement (DDR) system by Deng et al. (2017) and Wu et al. (2019) and many other methods are commonly used. There is also actor-critic Deep Reinforcement Learning that aims to train the two models at the same time but it is currently being relatively unexplored though this approach is believed to be an even more powerful method. Deng et al. (2017) employed recurrent deep neural network (RDNN) to learn optimal trading strategies in the Chinese stock market and observed significant improvements in risk-adjusted returns compared to traditional methods. The latter two are a bit out of the scope of discussion from this paper since we focus on the DQN main structure.

Several studies have demonstrated the effectiveness of DQN in algorithmic trading. For example, Théate, et al. (2021) presents the Trading Deep Q-Network (TDQN) algorithm, a deep reinforcement learning (DRL) approach for algorithmic trading, aiming to optimize trading positions in stock markets. It maximizes the Sharpe ratio performance indicator, using artificial trajectories generated from stock market historical data for the training process. The work further assesses the performance of the TDQN algorithm, it highlights how the TDQN algorithm performs well on the Apple stock, where it can detect and benefit from major trends and exhibit both reactive and proactive behaviors. However, there are challenges to overcome, including the poor observability of the trading environment and high variability of the algorithms.

Zejnullahu, et al. (2022) used Double Deep Qlearning (DDQN) algorithm for trading single assets. The study provides a comprehensive analysis of the application of DDQN in financial trading, particularly in algorithmic trading of the S&P 500. It demonstrates that DDQN models, despite their complexity and susceptibility to overfitting, can exhibit robust insample performance and adapt to various market conditions. With a simple state representation, their work demonstrated DDQN's ability in financial trading, even under a realistic environment where trading cost is incurred.

III. Methodology

This paper builds upon the framework proposed by Zejnullahu et al. (2022), incorporating two significant modifications. Firstly, we conducted experiments using a dueling structured network in place of the traditional fully-connected network, comparing its performance with and without the dueling structure while keeping the penalty factors constant. Additionally, we employed a varying penalty strategy, gradually adjusting the penalty factors, to train a more conservative agent that penalizes negative rewards.

Dueling Structure

Dueling network architectures, introduced by Wang et al. (2015), expand on traditional Deep Reinforcement Learning (DRL) models by decomposing the Q-function into two distinct functions: the state value function (V) and the action advantage function (A). This decomposition enhances the model's ability to independently learn the value of a state and the relative advantage of each action, resulting in more precise Q-value estimations and improved overall performance.

In a financial context, this approach is particularly intuitive. While we may not explicitly formulate the problem as partially observed, it is often the case that the true state of the financial market remains unobserved. Therefore, we can only approximate the state using the available observations. Amidst significant noise, the same state-action pair may yield different Q-values at different times. Consequently, the dueling structure enables us to estimate the state and action separately, and the advantage estimation assists in selecting the most favorable investment decision, even when the state value is not accurately estimated.

Please refer to Section IV for further details.

Penalization Factor

In the context of Deep Reinforcement Learning (DRL), our objective is to update our network to obtain the optimal policy. Consequently, the update rule plays a crucial role. Similar to other Deep Learning methods, we utilize backpropagation to update the network parameters by computing the gradient of a loss function. However, the loss function employed in the Deep Q-Network (DQN) algorithm differs slightly. Let's consider a classical Bellman Q-update formulation:

$Q(s,a) \leftarrow Q(s,a) + \alpha[\delta]$

where $\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$, is known of the temporal difference, and γ is the discount factor. Without any modifications, the Bellman equation naturally updates the network to find the state-action pairs that yield the highest reward and expected next state value. However, as mentioned in the introduction, our goal is to train an agent that exhibits a relatively more conservative behavior, meaning it is more inclined to choose not to invest more frequently.

To accomplish this, we can manually increase the impact of negative rewards. We introduce an additional constant, β ($\beta >= 1$), and define a loss threshold, θ ($\theta <= 0$). If the immediate reward, r, is less than θ, we modify it by multiplying it with β, such that r = βr.

By doing this, we are making the temporal difference more conservative. For example, assume the agent took an action and got a strong negative reward, there is a still a possibility that the temporal difference is positive, since $\max_{a'} Q(s',a')$ could be high. Like stated above, we are relatively pessimistic about the state-representation under financial setting, thus we want to penalize this kind of situation by adding more weight to immediate negative return, while remains the same with normal or positive rewards.

Therefore, choosing the correct value for β is an important task is this study. In the next section, we performed experiments wit different penalization factors and compare their performance.

IV. Numerical Experiment

In this section, we conducted a numerical experiment to validate our model and algorithms. We adopted a similar approach to the study conducted by Zejnullahu et al. (2022).

Dataset

We collected E-mini S&P 500 continuous futures contract's daily price data from 2010 to 2023, and the 'Close' column will be used as the price, excluding the holidays and weekends, there are totally 3352 data points; the last 500 days will be used for out-of-sample evaluation and excluded from training.

During training process, every epoch/episode contains 252 data points, equivalent to the number of trading days in a year.

MDP and Trading Simulator

Basically all RL problems are firstly formualted as Markov Decision Process(MDP), containing a state representation, an action space, a reward function and a state transition probability distribution. Now we define these properties for our problem.

- **State**: 1-day and 5-day asset return.
- **Action Space**: 0: short, 1: stay-out-of-the-market, 2: long.
- **Reward**: net portfolio value percentage change, which includes a immediate return, plus any transcation cost if the position is changed.
- **State Transition**: move t and update state as the next 1-day and 5-day return.

To cooperate with this MDP, we built an environment, named StockTradingEnv, inheriting Open AI's Gym class, and works as follow.

init:

- data = series of input returns
- position = 0 (initially not holding any asset)
- portfolio value = input constant (starting capital)
- step = 0
- done = False

simulation step:

- get input action from ϵ greedy policy
- – if need to adjust position: new portfolio value = portfolio value + immediate return - transaction cost
  - else: new portfolio value = portfolio value + immediate return
- set reward = (new portfolio value - portfolio value) / portfolio value
- set step = step + 1
- set done = step == len(data)
- return state, action, reward, next state, done

In each iteration, environment is given an action, and the transaction cost will only incurs if the action instruct a change in position. For instance, if the action is 0, meaning to short this asset, and current position is -1, i.e. already in short position, then no transaction will occur and the reward will be the net immediate return.

The step function described above returns a transition tuple, which will be stored in a replay buffer. During training, transitions will be sampled from replay buffer and used to update the network.

Model and Environment Hyper-Parameters

Normal-DQN:

- Input: The input layer takes the n_observations-dimensional state vector as input.
- Layer 1: A fully connected (Linear) layer with n_observations input nodes and 128 output nodes, followed by a ReLU activation function.
- Layer 2: A fully connected (Linear) layer with 128 input nodes and 128 output nodes, followed by a ReLU activation function.
- Layer 3: A fully connected (Linear) layer with 128 input nodes and n_actions output nodes. The output represents the Q-values for each action in the given state.
- Output: The final output is the tensor of Q-values for each action in the given state.

Dueling-DQN:

- Input: The input layer takes the input_dim-dimensional state vector as input.
- Feature Layer: A fully connected (Linear) layer with input_dim input nodes and 128 output nodes, followed by a ReLU activation function.
- Advantage Layer:
  - a. A fully connected (Linear) layer with 128 input nodes and 128 output nodes, followed by a ReLU activation function.
  - b. A second fully connected (Linear) layer with 128 input nodes and output_dim output nodes, representing the advantage values for each action.
- Value Layer:
  - a. A fully connected (Linear) layer with 128 input nodes and 128 output nodes, followed by a ReLU activation function.
  - b. A second fully connected (Linear) layer with 128 input nodes and 1 output node, representing the value of the state.
- Output: The final output is obtained by combining the advantage and value streams. The advantage values are adjusted by subtracting their mean, and this adjusted advantage is added to the value. The result is the Q-values for each action in the given state.

Environment Hyper-Parameters: As shown in Table 1, the hyperparameters we used for training process are as follows: N_EPISODES: The number of episodes for which the training process will run. In this case, 100 episodes. N_EXPLORATION: The number of episodes during which exploration probability (epsilon) linearly decays from 1 to 0. In this case, it's set to 20 episodes. TAU: A parameter used in soft update of the target network, which blends the weights of the policy network and the target network. A value of 0.1 means that 10% of the policy network's weights are blended with 90% of the target network's weights. LR: The learning rate for the optimizer (ADAM in this case), which determines the step size for weight updates during training. It's set to 1e-3 (0.001) in this case. batch_size: The number of

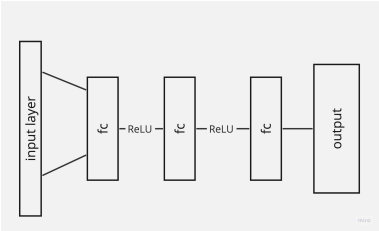

Figure 1: Normal fully connected DQN structure

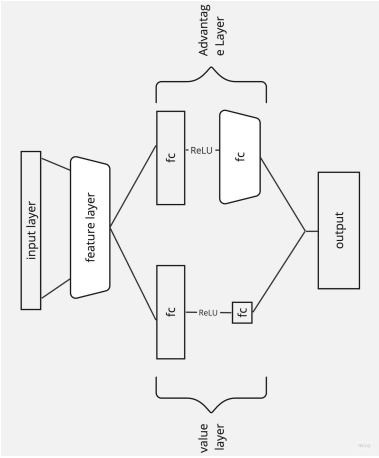

Figure 2: Dueling DQN structure

samples used for each update during training. In this case, it's set to 1024. optimizer: The optimization algorithm used for training the network. In this case, it's the ADAM optimizer. GAMMA: The discount factor for future rewards in the Q-learning algorithm, which determines the importance of future rewards compared to immediate rewards. It's set to 0.9 in this case. transaction_cost: The cost incurred for each transaction (buying or selling), which is set to 0.0005. penalty_threshold: The threshold for the penalty, set to -0.01. If the reward is below this threshold, a penalty will be applied. penalize_factor: A factor by which the penalty is multiplied. This factor varies in the code, and different values are tested during training.

Experiment steps

The purpose of this experiment is to see the impact of penalty factor on trained policies by looking at the change in portfolio values and counts of actions. For every pre-selected penalty factor, we trained a policy using the same environment and hyper-parameters. Then we observe policies' performance on the test dataset, which is also the same for all experiments.

During testing, we set the initial portfolio value, i.e. starting capitalas $1000 and observe its changes. Furthermore, we record the actions taken at each step under each policy, and we do an aggregation count in the end to measure the effectiveness of penalty factor.

Finally, we evaluate the difference between dueling and normal DQN network structure.

Hyper Parameters	Value
N_EPISODES	100
N_EXPLORATION	20
TAU	0.1
LR	1.00E-03
batch_size	1024
optimizer	ADAM
GAMMA	0.9
transaction_cost	0.0005
penalty_threshold (θ)	-0.01
penalize_factor (β)	varies

Table 1: Hyperparameters used for training process.

Experiment Results

As depicted in Figure 3, we observe that as the penalty factor increases, the trend of the portfolio value becomes "flat." This behavior can be attributed to the fact that a higher penalty factor reduces the negative immediate reward, leading to lower Q-values for actions associated with penalties. It's important to note that only actions 0 (short) and 2 (long) generate rewards, while action 1 (stay) yields a reward of 0. Consequently, actions 0 and 2 are penalized. Therefore, if we set the penalty factor excessively high, the best action under all states becomes action 1 (stay), as the Q-values for the other two actions are greatly diminished due to excessive penalization. In our experiment, a penalty factor of 4 or 5 causes the agent to practically take no action, indicating an extremely conservative approach. We refer to this as over-penalization.

Figure 4 provides additional support for our results. When the penalty factor is set to 1, indicating no penalty, we observe that the agent takes 282 long actions, 161 stay actions, and 72 short actions. As we progressively increase the penalty factor, the agent becomes inclined towards action 1 (stay). For penalty factors greater than 2, the agent predominantly chooses to stay out of the market, aligning with the "flat lines" depicted in Figure 3.

Furthermore, we evaluate the effectiveness of the dueling network structure. Figure 5 illustrates the difference in net portfolio value between the dueling and non-dueling structures. Although the y-axis represents the differences, we observe similar trends to those seen in Figure 3. Notably, the differences are generally small in the first half and gradually increase. One possible interpretation is that the dueling structure is more sensitive. While it is challenging to definitively determine which structure is superior, the discrepancies in the actions taken are evident.

V. Outlook

Expanding on the potential applications of our work, we propose the following possibilities:

Risk Control: Although our approach in this paper is simplistic, there is potential to apply it in investment risk control by collaborating with more informative state representations and sophisticated datasets. For instance, if other models indicate a trading decision, the decision-maker can utilize a risk-averse trading agent to re-evaluate and minimize risk.

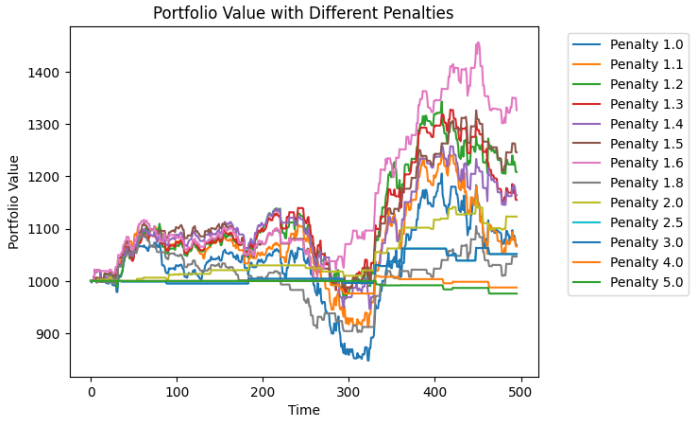

Figure 3: Portfolio profile based on various penalty value

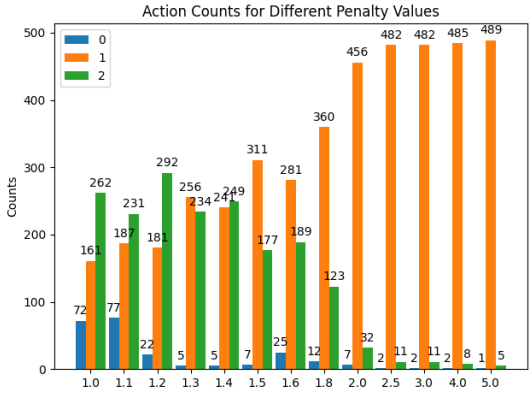

Figure 4: Action counts (0: short, 1: stay-out-of-the-market, 2: long) for each penalty value

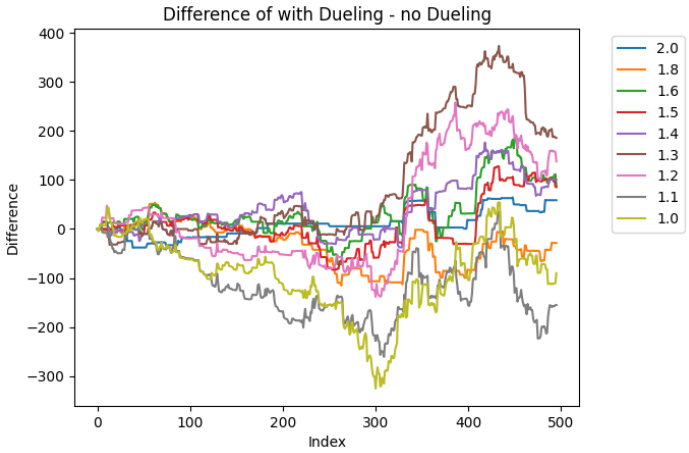

Figure 5: Portfolio profile difference between with-dueling and no-dueling structure under various penalty factors

Behavior Cloning: In this paper, we adjusted the reward by implementing a basic heuristic. By modifying the penalty factor to a similarity index, which evaluates the probability that an expert would take a particular action given a certain state, we can potentially update the network to mimic the expert's policy using a gradient-based approach.

Modelling Exogenous Effects: In certain scenarios, incorporating multiple factors into the reward function poses challenges in formulating the Markov Decision Process (MDP). However, by demonstrating the possibility of influencing policy through transparent modifications to the reward, we can exclude certain exogenous factors from the MDP and simply reintroduce them into the reward before updating the network. With careful design, this can simplify the problem while retaining crucial information.

VI. Conclusion

In this study, we examined the use of Double Deep Q-Network (DDQN) in financial trading to train risk-averse autonomous trading agents. In our original proposal, we intended to reproduce the simple DDQN for trading single assets in finance and extend the model with additional Dueling structure under various market conditions to test their performance. In our final study, we did implement the both architectures with further extended study on impact of penalized rewards with a more clear purpose of training a risk-averse trading agent. By adjusting the reward function to penalize negative returns and high risk, we found that agents became more conservative in their trading decisions. However, excessive penalization rendered the agents practically inactive. The study also highlighted the sensitivity of the dueling network structure compared to the non-dueling structure. The work has implications for risk control methods, behavior cloning, and simplifying problem formulation in autonomous trading. Finding the optimal balance between risk aversion and profitability remains a challenge. Overall, this research demonstrates the potential of reinforcement learning in modern finance, as well as more possibilities with self-designed reward functions. Again, our source code are available at https://github.com/cl6530/DLfinalproject

References

Chen, L.; and Gao, Q. 2019. Application of deep reinforcement learning on automated stock trading. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 2019-October: 29–33.

Deng, Y.; Bao, F.; Kong, Y.; Ren, Z.; and Dai, Q. 2017. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3): 653–664.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Mousavi, S. S.; Schukat, M.; and Howley, E. 2018. Deep Reinforcement Learning: An Overview. [Online]. Available: http://arxiv.org/abs/1806.08894, arXiv:1806.08894.

Pricope, T.-V. 2021. Deep Reinforcement Learning in Quantitative Algorithmic Trading: A Review. [Online]. Available: http://arxiv.org/abs/2106.00123, arXiv:2106.00123.

Th'eate, T.; and Ernst, D. 2021. An application of deep reinforcement learning to algorithmic trading. *Expert Systems with Applications*, 173(April 2020): 114632.

Wang, Z.; Schaul, T.; Hessel, M.; Van Hasselt, H.; Lanctot, M.; and De Frcitas, N. 2016. Dueling Network Architectures for Deep Reinforcement Learning. In *33rd International Conference on Machine Learning (ICML 2016)*, volume 4, 2939–2947.

Wu, J.; Wang, C.; Xiong, L.; and Sun, H. 2019. Quantitative Trading on Stock Market Based on Deep Reinforcement Learning. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2019-July, 1–8. IEEE. ISBN 9781728119854.

Zejnullahu, F.; Moser, M.; and Osterrieder, J. 2022. Applications of Reinforcement Learning in Finance – Trading with a Double Deep Q-Network. [Online]. Available: http://arxiv.org/abs/2206.14267, arXiv:2206.14267.