Reinforcement Learning Project Report: Reinforcement Learning in Quantitative Finance

Ribeiro, Bernardo	Lendering, Camile	Cianci, Giuseppe
u234623	u234753	u234127

March 7, 2024

Contents

1 Introduction **3**

2 Stock Trading Environment **3**
- 2.1 Environment Configuration . . . 3
- 2.2 State and Action Space . . . 4
- 2.3 Environment Dynamics . . . 4
- 2.4 Prophetic Actions . . . 4
- 2.5 Utility Functions . . . 5

3 Imitative Recurrent Deterministic Policy Gradient (iRDPG) **5**
- 3.1 Theoretical Framework . . . 5
  - 3.1.1 Deep Deterministic Policy Gradient (DDPG) Foundation . . . 5
  - 3.1.2 Recurrent Deterministic Policy Gradient (RDPG) . . . 5
  - 3.1.3 Imitative Recurrent Deterministic Policy Gradient (iRDPG) . . . 6
- 3.2 RDPG with the Continuous MountainCar Environment . . . 6
  - 3.2.1 Problem 1: Not Enough Exploration . . . 7
  - 3.2.2 Problem 2: Not Enough Learning . . . 7
- 3.3 iRDPG with the Stock Trading environment . . . 8

4 Transformer Actor-Critic with Regularization (TACR) **10**
- 4.1 Theoretical Framework . . . 10
  - 4.1.1 Transformer Actor . . . 10
  - 4.1.2 Critic with Regularization and Behaviour Cloning . . . 11
  - 4.1.3 TACR Algorithm . . . 11
- 4.2 TACR with the Cart Pole Environment . . . 12
- 4.3 TACR with the Stock Trading Environment . . . 13

5 Conclusion **16**

6 Repository for the Project **16**

List of Figures

1 Overview of the iRDPG Framework . . . 6
2 Reward per Episode with High Noise . . . 7
3 Reward per Episode with Noise Decay . . . 7
4 RDPG: Reward for Train Dataset per Iteration . . . 8
5 RDPG: Reward for Test Dataset per Iteration . . . 9
6 TACR Framework for Portfolio Allocation . . . 10

7 Train Actor Loss per Timestep 12
8 Train Critic Loss per Timestep 13
9 Average Reward per Testing Episode 13
10 Reward for Train Dataset per Iteration 14
11 Reward for Test Dataset per Iteration 14
12 Accumulated Returns for the Testing Dataset 15
13 Shares Held for the Testing Dataset 15
14 Trade Returns for the Testing Dataset 16

1 Introduction

In this project, our objective was to conduct an empirical evaluation of Reinforcement Learning (RL) methodologies, with a specific focus on their application in quantitative finance. RL presents a compelling approach in this domain, primarily due to its capability to train an end-to-end agent. Such an agent can directly execute profitable actions, circumventing the need for multiple models to address various sub-tasks.

However, the application of standard RL techniques in finance is challenging. These challenges stem from several factors: the poor handling of out-of-distribution states by RL methods, the vastness of the state space (characterized by numerous stocks with continuous price points and other continuous financial variables), the noisiness of financial data, the non-observability of actual market states (only having access to historical volume and price data), and the general non-impact of an agent's actions on the environment (for instance, stock prices remain unaffected unless the agent possesses substantial financial leverage).

During our literature review, we encountered FinRL, a tailored RL environment designed for automated stock trading [Liu et al., 2022]. We also explored innovative RL methods like Imitative Recurrent Deterministic Policy Gradient (iRDPG) [Liu et al., 2020], a batch DRL method utilizing a replay buffer and Imitation Learning for intraday trading on one-minute bar intervals. Additionally, we studied Transformer Actor-Critic with Regularization (TACR) [Lee and Moon, 2023], an offline DRL method based on transformers for Portfolio Optimization. Motivated by these findings, we set out to develop our own implementations of iRDPG and TACR for automated stock trading using daily stock data. This involved creating a custom stock trading environment and assessing its performance against common benchmarks.

2 Stock Trading Environment

In order to train our agents, we designed the `SimpleOneStockStockTradingBaseEnv`, a novel gym environment tailored for single-stock trading on a daily timeframe. This environment provides a simplified realistic simulation of stock trading activities, where an agent can execute discrete actions (buy, hold, sell) based on daily stock data and technical indicators. The key features of this environment are outlined as follows:

2.1 Environment Configuration

The environment is initialized with various parameters:

- **Initial Account Value:** Set at $1,000,000 by default, this parameter represents the starting capital for trading activities.
- **Discount Factor:** A discount factor of 0.999 is used to calculate future rewards.
- **Trading Constraints:** The maximum percentage of account value permissible per trade is capped at 10 %, coupled with transaction fee rates for both buying (0.025 %) and selling (0.025 %). To simplify even further, the agent is only able to go long or short on 10 % of account value, thus effectively blocking position sizing (either long or short on 10 % of account value, or not positioned). This is done to prevent the account value from going to negative values while shorting, which would cause the reward function to break, since it will use a logarithm for relative performance which is undefined for negative values.
- **Market Data:** Arrays for open, high, low, and close prices are provided alongside an array of technical indicators.
- **Termination:** An episode is terminated when the last trading day of the dataset begins.

2.2 State and Action Space

The state space includes a representation of the current position (long, short, or hold) and an array of technical indicators for the current day, providing a rich set of features for decision-making. Open, High, Low, Close (OHLC) data is not part of the state representation, since it can generate out-of-distribution effects on learning (e.g. stock prices may reach values never before seen during the training phase). Thus, OHLC data is indirectly represented in the technical indicators that will be used, since all technical indicators are just calculations using only OHLC and volume data, but often presented using a relative scale. This prevents out-of-distribution problems.
The action space is discrete with three possible actions: -1 (sell), 0 (hold), and 1 (buy). For simplification purposes, if the agent is currently long and a sell action is sent, then it closes the previous position (sell all shares it currently has, but doesn't go short). The same applies to the opposite case.

2.3 Environment Dynamics

At each step, the environment processes the agent's action, updates the portfolio accordingly (shares held, cash in hand, account value), and computes the reward. The reward is calculated based on the log ratio of the current and previous account values, since it causes the reward to be linearly scaled due to the compounding effect of returns which is desirable for RL algorithms, scaled by the reward scaling factor, and taking into consideration transaction fees, using the following formulas:

- **Buy Action (action = 1):**

$$\text{shares\_to\_move} = \begin{cases} -\text{shares\_held}, & \text{if shares\_held} < 0 \\ 0, & \text{if shares\_held} > 0 \\ \left\lfloor \frac{\text{percentage\_acc\_value\_per\_trade} \times \text{account\_value}}{\text{close\_price}} \right\rfloor, & \text{otherwise} \end{cases} \tag{1}$$

$$\text{total\_volume} = \text{shares\_to\_move} \times \text{close\_price} \times (1 + \text{buy\_transaction\_fee\_rate}) \tag{2}$$

$$\text{cash\_in\_hand} = \text{cash\_in\_hand} - |\text{total\_volume}| \tag{3}$$

- **Sell Action (action = -1):**

$$\text{shares\_to\_move} = \begin{cases} \text{shares\_held}, & \text{if shares\_held} > 0 \\ 0, & \text{if shares\_held} < 0 \\ \left\lfloor -\frac{\text{percentage\_acc\_value\_per\_trade} \times \text{account\_value}}{\text{close\_price}} \right\rfloor, & \text{otherwise} \end{cases} \tag{4}$$

$$\text{total\_volume} = \text{shares\_to\_move} \times \text{close\_price} \times (1 - \text{sell\_transaction\_fee\_rate}) \tag{5}$$

$$\text{cash\_in\_hand} = \text{cash\_in\_hand} + |\text{total\_volume}| \tag{6}$$

- **Account Value Update:**

$$\text{account\_value} = \text{cash\_in\_hand} + \text{shares\_held} \times \text{close\_price} \tag{7}$$

- **Reward Calculation:**

$$\text{reward} = \ln\left(\frac{\text{account\_value}}{\text{previous\_account\_value}}\right) \times \text{reward\_scaling} \tag{8}$$

2.4 Prophetic Actions

A unique feature of this environment is the generation of prophetic actions. These actions are determined based on a window of stock prices, where the agent aims to maximize gains by taking the best action (buy or sell) at the start of the window and closing the position at the optimal point within the window. This approach simulates an ideal trading strategy based on hindsight, providing a benchmark for Imitative Learning modules, which are used in both implementations.

2.5 Utility Functions

The `YHFinanceProcessor` class encapsulates several utility functions which were introduced to help set up the environment with the following functionalities:

- **download_data**: From a list of stock tickers and a start date and end date, download historical OHLC + volume daily data for all the stocks, using Yahoo Finance.
- **clean_data**: Fill missing values of OHLC data with values from previous rows and volume data with 0.
- **add_technical_indicators**: Add technical indicators to the dataset using the libraries pandas_ta and stockstats.

3 Imitative Recurrent Deterministic Policy Gradient (iRDPG)

iRDPG is an adaptive quantitative trading model proposed by [Liu et al., 2020] to address two challenges: representing the noisy high frequency financial data and balancing exploration and exploitation of the agent. It can be split into two components: one part which is responsible for the imitation learning (i.e. demonstration buffer and behavior cloning) and the other part which is standard RDPG. RDPG extends the DDPG algorithm (Deep Deterministic Policy Gradient) [Lillicrap et al., 2019], by using recurrent neural networks (RNNs) to handle partially observable environments or environments with long-term dependencies.

3.1 Theoretical Framework

3.1.1 Deep Deterministic Policy Gradient (DDPG) Foundation

DDPG is a model-free, off-policy actor-critic algorithm that can learn policies in continuous action spaces. It uses two neural networks: the actor, which proposes actions given the current state, and the critic, which evaluates the proposed action.
DDPG draws inspiration from the Deep Q Network (DQN) algorithm, notably in its use of a target Q network, which helps to stabilize learning by providing consistent targets in temporal difference updates. During training, episodes are randomly sampled from a replay buffer to reduce sample correlations.
The actor network is updated by adjusting its weights in the direction that maximizes the expected return, using the policy gradient method. This involves computing the gradient of the critic's Q-value with respect to the actor's actions.
The critic network is updated by adjusting its weights to minimize the difference between its estimated Q-values and the target Q-values.
The target networks are updated by slowly combining the learned networks parameters, using soft updates: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. This means that the target values change slowly during training, improving the stability of the algorithm.
In DDPG, exploration of the environment is handled by adding noise sampled from some noise process to the actor policy. For example, in environments that simulate physical control problems with inertia, an Ornstein-Uhlenbeck process can be used to generate temporally correlated exploration noise [Lillicrap et al., 2019].

3.1.2 Recurrent Deterministic Policy Gradient (RDPG)

RDPG is a direct extension of DDPG for Partially Observable Markov Decision Processes (POMDP), incorporating recurrent neural networks into the actor-critic architecture to handle environments with incomplete information and temporal dependencies, effectively replacing the initial dense layers in the actor and critic network with LSTM (Long Short-Term Memory) layers. This enables the algorithm to process sequences of observations, retaining relevant information over time [Heess et al., 2015].

3.1.3 Imitative Recurrent Deterministic Policy Gradient (iRDPG)

iRDPG combines ideas from imitation learning and RDPG to effectively solve the quantitative trading problem defined within a POMDP framework. Using a demonstration buffer and behaviour cloning for initial strategy development and leveraging RDPG to handle the uncertainties and temporal aspects of financial data.

Initially, the demonstration buffer is filled with episodes from the Dual Thrust trading strategy. The Dual Thrust strategy is based on 'breakout trading', where positions are taken based on the price of stock breaking out of a predefined range, where the range is determined by recent highs and lows. When the price breaks through the upper threshold, a long position is taken (i.e. the stock is bought). Conversely, if the price drops below the lower threshold, a short position is taken (i.e. the stock is sold).

The agent undergoes pre-training using these demonstrations, allowing it to learn a basic trading strategy. Training involves sampling minibatches that include both demonstration and agent-generated episodes.

Behaviour cloning in this context involves using a "prophetic trading expert", which takes long positions at the lowest prices and short positions at the highest, to set goals for each trade.

If and only if the critic indicates the expert actions are superior, the agent's actions are compared to these expert actions, using the Q-filter technique. This comparison yields a behavior cloning loss (BC loss), which can be seen below: [Liu et al., 2020]

$$L' = -\mathbb{E}\left[\left\|\mu^{\theta}(h_t) - \bar{a}_t^i\right\|^2 1_{Q(h_t^i,\bar{a}_t^i)>Q(h_t^i,\mu^{\theta}(h_t))}\right]. \tag{9}$$

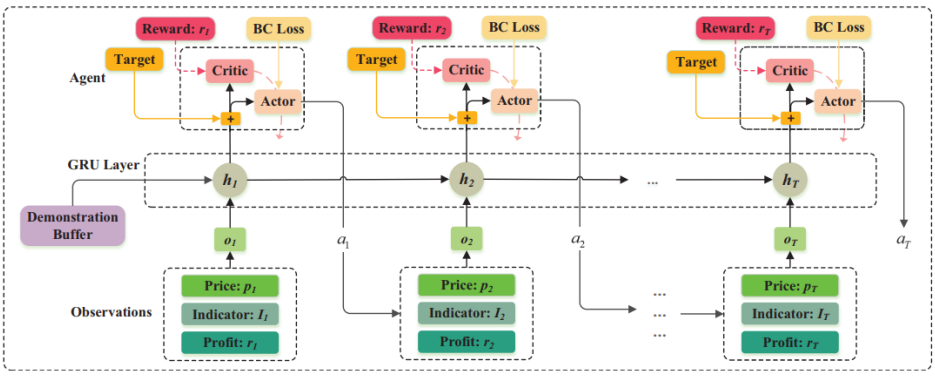
This behavior cloning loss is then used as an auxiliary loss to calculate a modified policy gradient that is applied to the actor, which helps eliminate inefficient exploration phases:

$$\nabla_{\theta}\bar{J} = \lambda\nabla_{\theta}J + (1-\lambda)\nabla_{\theta}L', \tag{10}$$

Where $\nabla_{\theta}\bar{J}$ is the policy gradient and λ is a hyperparameter that control the weights between the losses.

An overview of the iRDPG method can be seen below:

Figure 1: Overview of the iRDPG Framework



Source: Extracted from [Liu et al., 2020]

3.2 RDPG with the Continuous MountainCar Environment

As a first step, we implemented RDPG to solve the MountainCar-Continuous environment. While doing this, we faced the following problems.

3.2.1 Problem 1: Not Enough Exploration

Our first implementation wasn't able to solve the MountainCar environment because the agent only learned to drive with full thrust to the right, but it didn't yet learn to get more momentum by driving up the opposite side of the hill first. To add more exploration during the learning phase, we increasingly added random noise. However, this still wasn't enough. Only when we changed the completely random noise to Ornstein–Uhlenbeck process (OU) noise, the agent was able to solve the environment.

3.2.2 Problem 2: Not Enough Learning

Figure 2: Reward per Episode with High Noise

We noticed that the reward per episode had lots of noise and wasn't very consistent. This is due to the fact that the OU noise we added for exploration was too high. Around 20 % of the action came from the model while 80 % came from the OU noise. Basically, the OU noise is good enough to solve the environment sooner or later because it can behave similar to the momentum we want the agent to learn. We solved this problem by adding linear noise decay.

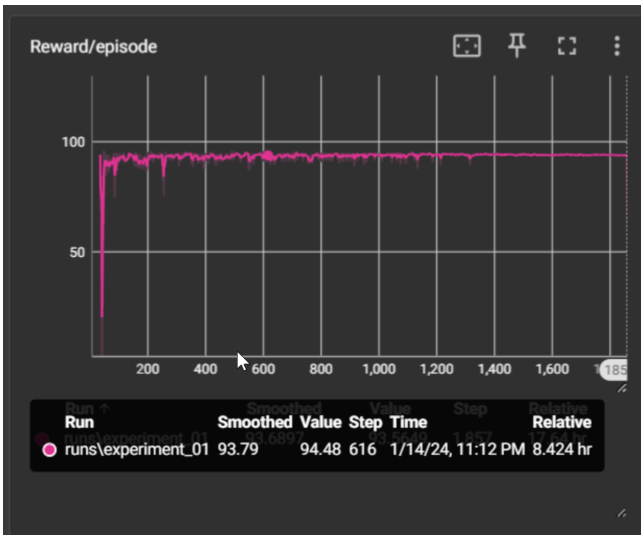


Figure 3: Reward per Episode with Noise Decay

Thanks to this change, the reward per episode is much more stable. Thus, we were able to solve the MountainCar problem (i.e. the agent gets a reward greater than 90 on 100 consecutive episodes).

3.3 iRDPG with the Stock Trading environment

After the successful test of RDPG with the MountainCar environment, we implemented the necessary changes to convert RDPG into iRDPG, by including the Behaviour Cloning Loss and generating the prophetic actions for comparison, and also creating a function to generate expert actions to fill up the replay buffer. An experiment using the Stock Trading environment was held, configured as follows:

- **Experiment Configuration:**
  - Stock: Intel (INTC)
  - Training Dataset: 01/01/2004 - 01/01/2018
  - Testing Dataset: 01/01/2018 - 01/01/2024
  - Discount Factor: 0.999
  - Lambda = 0.8
  - Critic Learning Rate = 1×10^{-5}
  - Actor Learning Rate = 1×10^{-5}
  - Batch Size = 20
  - Action Noise Decay Steps = 900,000
  - Max Timesteps: 1,000,000
  - Technical Indicators: PCT_RETURN(2), OBV_PCT_CHANGE(8), BINARY_SMA_RISING(24), RSI(14).
- **Expert Actions (Demonstration Buffer):**
  - For every day, returns 1 (Buy) if the Technical Indicator SMA (Simple Moving Average) over 24 days has a positive inclination, -1 (Sell) if it has a negative inclination. For both cases, if the agent already is long or short, the returned action is 0 (Hold).
- **Prophetic Actions (Behaviour Cloning):**
  - Employing a window size of 4 days, calculate the price change within the start of the window and for each day in the window until the end, and take the action that maximizes the gain for any of the days and closing the position on the day when the gain is maximum. Then continue from the day after the position is ended and not at the end of the window.

Using Tensorboard, the evolution of the average reward for the train dataset and the test dataset can be seen in the plots below:

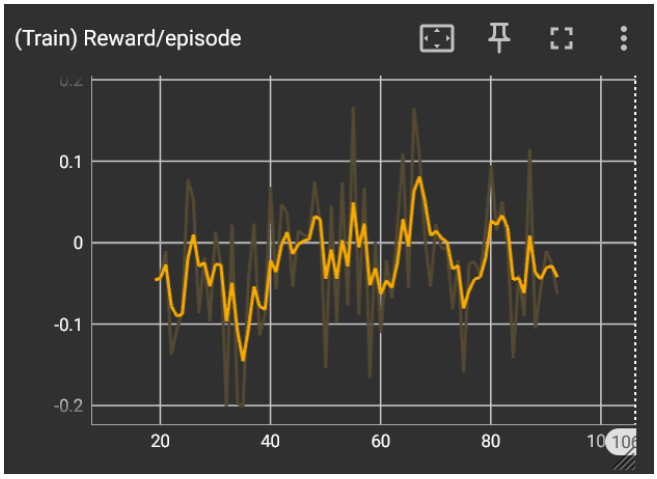


Figure 4: RDPG: Reward for Train Dataset per Iteration

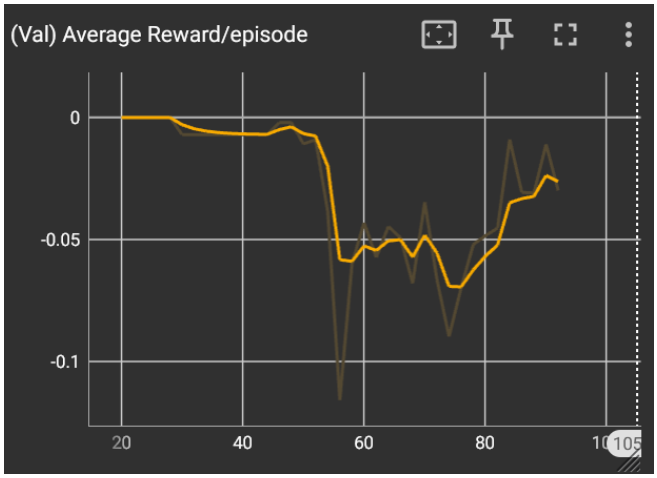


Figure 5: RDPG: Reward for Test Dataset per Iteration

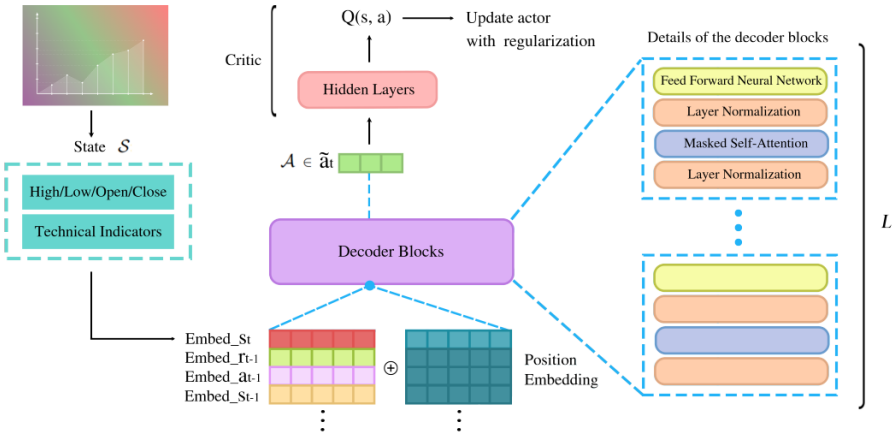
The agent did not succeed in learning a profitable policy, as evidenced by its inability to produce satisfactory results. The algorithm struggled with generalization, particularly with the Behavior Cloning (BC) loss and the employed prophetic actions, displaying a limited learning capacity in training without extending this to the validation dataset.
A significant issue was the prohibitive training duration, exceeding 12 hours and increasing with the number of technical indicators and the batch size. This extensive training time constrained the ability to conduct thorough testing, hyperparameter optimization, and exploration of various technical indicators.

4 Transformer Actor-Critic with Regularization (TACR)

4.1 Theoretical Framework

Transformer Actor-Critic with Regularization (TACR) is a Reinforcement Learning algorithm proposed in [Lee and Moon, 2023] that combines Transformers with Offline Learning to leverage previous stock information and the correlation between previous MDP elements using an Attention Network. It uses an Actor-Critic structure, where a Transformer network acts as an Actor whose actions are evaluated by a Multi-Layer Perceptron acting as a critic network. The entire framework of TACR is shown in figure 6.

Figure 6: TACR Framework for Portfolio Allocation



Source: Extracted from [Lee and Moon, 2023]

Its features can be described as follows:

4.1.1 Transformer Actor

The Transformer Actor in TACR, as elucidated in [Lee and Moon, 2023], is a pivotal component that distinguishes this approach from conventional Actor-Critic models. It maps previous MDP elements to an action, effectively encoding a policy π consisting of decoder blocks using the Attention Mechanism and Hidden Layers:

$$h_0 = MW_e + W_p \tag{11}$$

$$h_l = \text{Decoder\_block}(h_{l-1}),\ l = 1, \ldots, L \tag{12}$$

$$\tilde{a} = \text{Softmax}(h_L W_L + b_L) \tag{13}$$

Where $M = (r_{t-u}, x_{t-u}, a_{t-u}, ..., r_{t-1}, x_{t-1})$ is a vector with previous MDP elements of length u (lookback), indicating how many timesteps in the past are included in the inputs. The previous MDP elements are then multiplied by the embedding matrix of weights W_e to represent the hidden state as an input, and then added with a time embedding weights W_p to include temporal continuity information. The correlation of the embedding inputs is learned through L decoder blocks, and the action $\tilde{a} = \{\tilde{a}_0, ..., \tilde{a}_J\}^T$ is finally predicted through a linear transformation layer and a Softmax function, where each element is interpreted as the probability of a certain action being taken under the context of discrete actions.
For this project and just like in the original paper, the transformer network GPT-2 was used in the transformer actor architecture.

4.1.2 Critic with Regularization and Behaviour Cloning

Being an Offline Reinforcement Learning Algorithm, TACR does not interact with the environment and learn through trial and error, but rather imitating prepared suboptimal actions. Therefore, suboptimal trajectories must be created for every environment. Using pre-generated suboptimal datasets for learning has a big time-efficiency advantage over traditional off-policy reinforcement learning that is traditionally done in the stock field, allowing for much faster training since no further interaction with the environment is needed.
Due to the agent not interacting with the environment when updating the policy in Offline Reinforcement Learning, it tends to incorrectly estimate the value of actions of out-of-distribution states, which requires the usage of a regularization method, and in this algorithm the policy is regularized adding a behavior cloning regularization term:

$$\pi = \underset{\pi}{\operatorname{argmax}}\ \mathbb{E}_{(x,a,r)\sim\mathcal{D}}\left[\lambda Q(x, \pi(M)) - (\pi(M) - a)^2\right] \tag{14}$$

Where π is the encoded policy in the transformer actor and the sequence M is the stacked previous MDP elements. The term $(\pi(M) - a)^2$ is the behavior cloning regularization term that makes the transformer actor follow the distribution of actions in the suboptimal trajectories included in the dataset. The hyperparameter λ controls the maximization of the Q-value as well as the minimization of the Behavior Cloning term, representing the strength of the regularizer. Sampling N transitions (x_i, a_i) randomly, λ is defined as:

$$\lambda = \frac{\alpha}{\frac{1}{N}\sum_{(x_i,a_i)} |Q(x_i, a_i)|} \tag{15}$$

With α being a hyperparameter that controls the sensitivity of λ, and the denominator $\frac{1}{N}\sum_{(x_i,a_i)} |Q(x_i, a_i)|$ normalize its value using the sample mean of $|Q(x_i, a_i)|$

4.1.3 TACR Algorithm

Our implementation of TACR was heavily influenced by the TACR algorithm provided in [Lee and Moon, 2023], and also the repository provided with the paper, which can be seen below:

Algorithm 1 TACR algorithm

Randomly initialize critic weights θ^Q and transformer actor weights θ^π.
Initialize target weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\pi'} \leftarrow \theta^\pi$
Set the length of the sequence u and minibatch size n.
for each iteration **do**
  for each environment step **do**
    Sample a random minibatch of $n \times u$ transitions $(x_{i-u}, \ldots, x_i, a_{i-u}, \ldots, a_i, r_{i-u}, \ldots, r_i, x_{i+1-u}, \ldots, x_{i+1})$ from suboptimal trajectories.
    $M_i = (r_{i-u}, x_{i-u}, a_{i-u}, \ldots, r_i, x_i)$
    Predict action $\tilde{a} = \pi(M_i|\theta^\pi)$ through transformer actor
    Set $y_i = r + \gamma Q'(x_{i+1}, \pi'(M_{i+1}|\theta^{\pi'})|\theta^{Q'})$
    Apply mean squared error to update critic:
      $Loss = \frac{1}{N}\sum_i (y_i - Q(x_i, a_i|\theta^Q))^2$, where $N = n \times u$.
    Set hyperparameter α and $\lambda = \frac{\alpha}{\frac{1}{N}\sum_{(x_i,a_i)} |\mathcal{Q}(x_i,a_i)|}$
    Update the transformer actor network:
      $\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_{\tilde{a}}\{\lambda Q(s, \tilde{a}|\theta^Q) - (\tilde{a} - a)^2\}|_{x=x_i, \tilde{a}=\pi(M_i)} \nabla_{\theta^\pi}\pi(M_i|\theta^\pi)$
    Update the target weights:
      $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}$
  end for
end for

In TACR, both the critic and transformer actor networks are initialized randomly. The algorithm uses a sequence of previous Markov Decision Process (MDP) elements of a certain length, denoted by the

hyperparameter u (lookback), as inputs to the transformer actor. These inputs are designed to capture correlations and dependencies in the data over time.
The algorithm proceeds by sampling minibatches of $n \times u$ transitions to minimize data correlation. The transformer actor employs an attention network to learn the correlations among the embedded MDP elements, resulting in predicted actions for the current state.
For network updates, the critic's parameters are adjusted using a mean squared error method, focusing on minimizing the discrepancy between predicted and actual Q-values. Concurrently, the transformer actor is updated considering both the critic's Q-value estimations and a behavior cloning term, with the objective function incorporating regularization.
Stability in learning is achieved by updating the target network weights gradually (Polyak Averaging), using a small value of τ to ensure that changes are incremental.

4.2 TACR with the Cart Pole Environment

After implementing the TACR algorithm, a test was set up using the gym CartPole-v1 environment to verify that it was working correctly, using the following configuration:

- **Experiment Configuration:**
  - Discount Factor: 0.99
  - Alpha (BC Regularization Factor) = 0.9 (Optimized manually)
  - Critic Learning Rate = 1×10^{-4} (Optimized manually)
  - Actor Learning Rate = 1×10^{-4} (Optimized manually)
  - Batch Size = 32
  - Lookback = 20 steps
- **Suboptimal Trajectories:**
  - 200000 suboptimal trajectories generated using four different action generator sources: 1. random, 2. biased random, 3. heuristic (based on pole angle), 4. oscillating generator

The results for the learning process can be seen in the following plots:

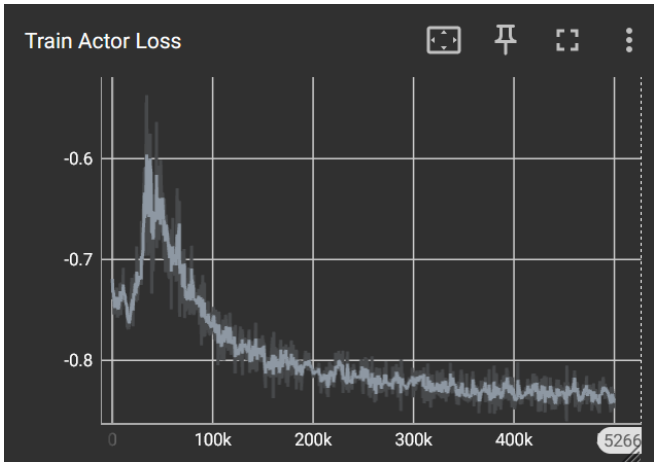


Figure 7: Train Actor Loss per Timestep

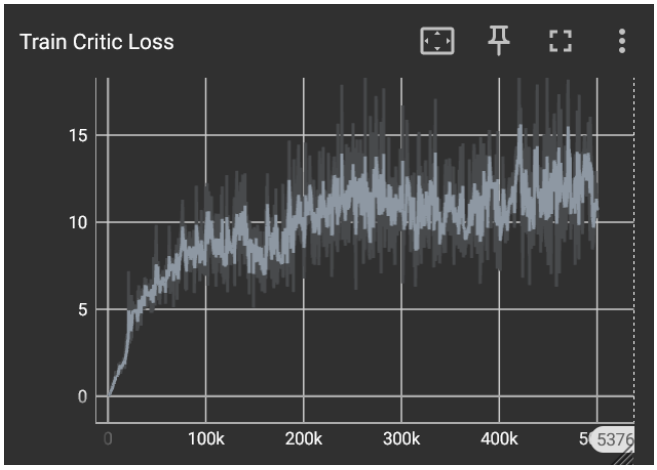


Figure 8: Train Critic Loss per Timestep

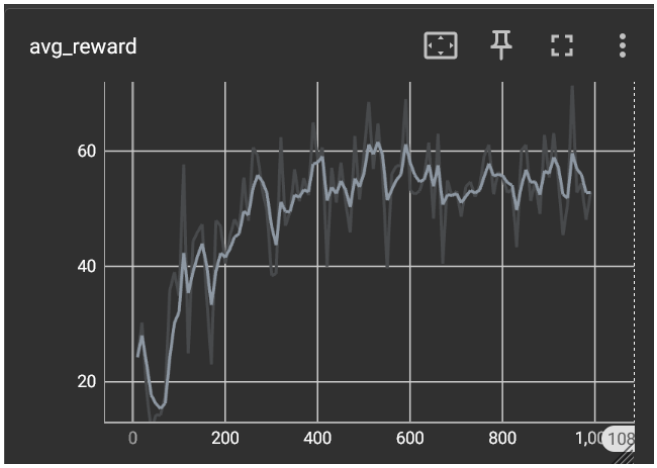


Figure 9: Average Reward per Testing Episode

The agent was thus able to learn from the suboptimal trajectories and stabilized, having an average reward of around 60.

4.3 TACR with the Stock Trading Environment

In the next phase of our project, we put our Transformer Actor-Critic with Regularization (TACR) algorithm to the test in our custom-designed simple stock trading environment. The experiment was configured as follows, incorporating both well-defined properties and finely-tuned hyperparameters:

- **Experiment Configuration:**
  - Stock: Intel (INTC)
  - Training Dataset: 01/01/2004 - 01/01/2018
  - Testing Dataset: 01/01/2018 - 01/01/2024
  - Discount Factor: 0.999
  - Alpha (BC Regularization Factor) = 0.9 (Optimized manually)
  - Critic Learning Rate = 1×10^{-5} (Optimized manually)
  - Actor Learning Rate = 1×10^{-5} (Optimized manually)
  - Batch Size = 64

- Lookback = 20 trading days
- Technical Indicators: PCT_RETURN(2), OBV_PCT_CHANGE(8), RVI_PCT_CHANGE(20,2), RSI(14).

- **Suboptimal Trajectories:**
  - Employing a range of window sizes from 2 to 20, and varying offsets from 0 to 16. Each trajectory was generated by prophetically taking actions that yielded positive returns for each window throughout the dataset, resulting in the creation of over a hundred distinct suboptimal trajectories.

Hyperparameters and Technical Indicators were manually adjusted, using a manual optimization process, and the best combinations found within the time constraints were selected as shown above. Training was run in iterations of 1000 timesteps, where each timestep represents a random sampling of batches of 64 transitions from the suboptimal trajectories.
Using Tensorboard, the evolution of the average reward for the train dataset and the test dataset can be seen in the plots below:

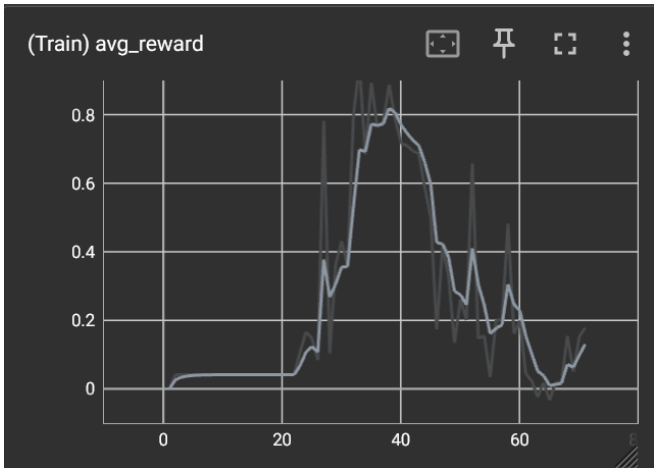


Figure 10: Reward for Train Dataset per Iteration

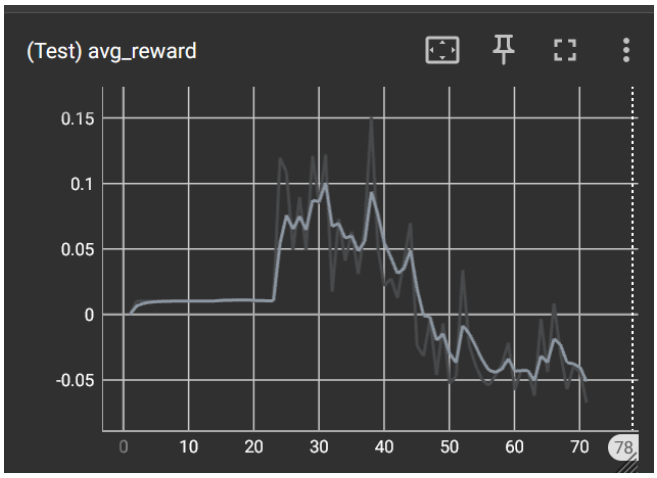


Figure 11: Reward for Test Dataset per Iteration

To assess the efficacy of our algorithm, we analyzed its performance in generating a profitable policy for the test dataset. This involved producing trading statistics and visualizations based on the policy iteration that achieved the highest average reward during validation.

One of the key visualizations is the accumulated returns plot displayed below. This plot reveals that our Transformer Actor-Critic with Regularization (TACR) strategy surpassed both the buy-and-hold and short-and-hold approaches over a six-year period with Intel stock. It is important to note that the returns for TACR, Buy and Hold, and Short and Hold were adjusted to reflect the use of 100 % of the capital, even though only 10 % was utilized during training. This adjustment was necessary to prevent errors in the logarithmic reward function.

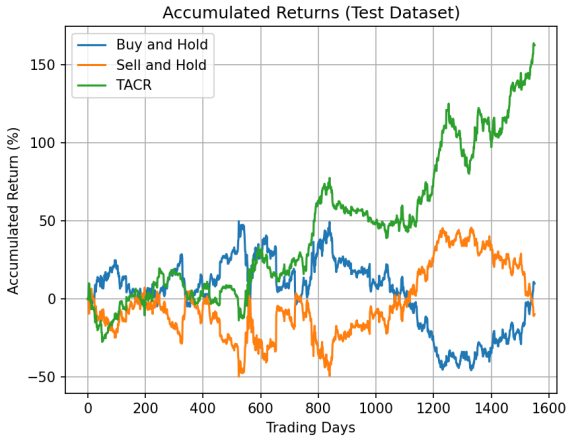


Figure 12: Accumulated Returns for the Testing Dataset

Furthermore, we examined the policy's behavior through the Shares Held Plot, shown below. This plot roughly represents the action signal multiplied by the number of stocks that could be bought or sold with 10 % of the account value. Notably, after initiating a buy, both actions (1) and (0) result in unchanged shares held figure, as the environment restricts purchases to a maximum of 10 % of the account's value (similar constraints apply to selling). The policy predominantly engages in short-term swing trading, achieving considerable success in this strategy.

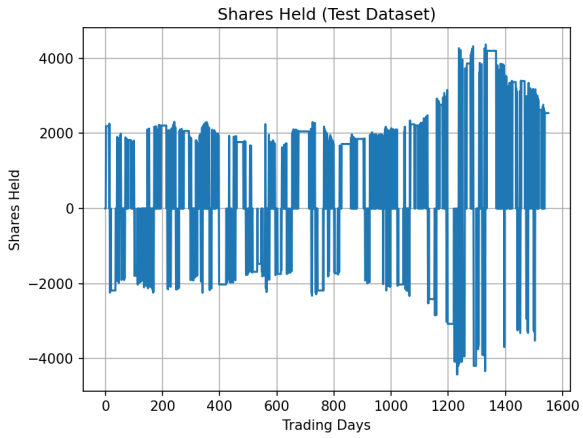


Figure 13: Shares Held for the Testing Dataset

Additionally, we observed the distribution of trade returns, as illustrated in the following plot. A Trade Return here refers to the percentage return realized between entering (buying or shorting) and closing a position.

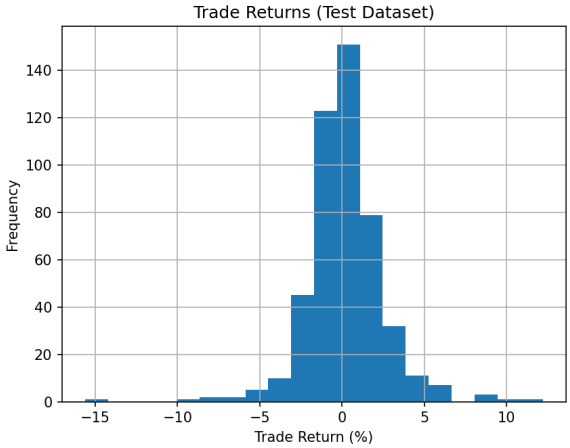


Figure 14: Trade Returns for the Testing Dataset

In conclusion, while the results are encouraging, further investigation and optimization of hyperparameters are required, which was limited by the project's time constraints. Additionally, our testing was confined to just two stocks—Occidental Petroleum and Intel. Expanding the test to a broader range of stocks is necessary to validate the general applicability of this method.

5 Conclusion

In our exploration of applying Reinforcement Learning (RL) to quantitative finance, we have encountered both promising possibilities and significant challenges. While RL has shown potential in specific areas of finance, its application is often hindered by complexities such as large state spaces, data efficiency issues, partial observability, and the inherent non-stationarity of financial markets. Furthermore, the effect of arbitrary actions on the environment can't be observed in the quantitative finance domain (e.g. the effect of how actions impact stock prices can't easily be simulated). Also, some counterfactual information which is observed (i.e. what would the reward have been had the agent taken another action) can't be fully utilized by RL methods. These challenges, still largely unresolved, limit the effectiveness of RL in this domain.

Our implementation of the iRDPG algorithm in stock trading did not yield significant results, a situation possibly attributed to our limited time for training. With training sessions extending beyond 12 hours, time constraints were a major factor, preventing us from exploring the full potential of the algorithm.

Conversely, our implementation of TACR showed more promise. Preliminary results from this approach are encouraging, and our team plans to continue this research. We aim to investigate the impact of varying technical indicators, different stocks, and hyperparameter tuning to fully assess TACR's capabilities in stock trading.

Additionally, our repository includes a custom environment developed for Portfolio Allocation, a task frequently targeted in RL competitions. Literature suggests successful applications of RL in this area, and we plan to test our approach in the coming weeks. While these tests won't be included in this report due to time constraints, they represent an exciting direction for future research in our project.

In conclusion, while challenges remain in applying RL to financial tasks, our study underscores its potential, particularly in nuanced applications like portfolio allocation. We are optimistic about the future of RL in finance, acknowledging the need for further research and experimentation.

6 Repository for the Project

The full code and custom implementations of the algorithms and environments can be seen in the following Open-Source GitHub repository:
https://github.com/Giuspepe/finance-reinforcement-learning-project

References

[Heess et al., 2015] Heess, N., Hunt, J. J., Lillicrap, T. P., and Silver, D. (2015). Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*.

[Lee and Moon, 2023] Lee, N. and Moon, J. (2023). Offline reinforcement learning for automated stock trading. *IEEE Access*, PP:1–1.

[Lillicrap et al., 2019] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.

[Liu et al., 2022] Liu, X.-Y., Yang, H., Chen, Q., Zhang, R., Yang, L., Xiao, B., and Wang, C. D. (2022). Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance.

[Liu et al., 2020] Liu, Y., Liu, Q., Zhao, H., Pan, Z., and Liu, C. (2020). Adaptive quantitative trading: An imitative deep reinforcement learning approach. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):2128–2135.