**"به نام خدا"**



گزارش پروژه

**تهیه کنندگان: هلیا وفایی، جواد فرجی، علی خوانین زاده، پارسا احمدی، امیر ایزدی**

استارتاپ استودیو نیوتون

**Q1:**

The current state representation includes six recent log-returns (r\_{k-5} to r\_k), along with normalized time remaining (m\_k) and inventory remaining (i\_k). This is a well-balanced choice:

* The log-returns capture short-term price trends and volatility, helping the agent identify good or bad times to trade.
* The m\_k and i\_k values inform the agent about time pressure and workload, which are essential for strategic planning.
* The state is small enough to be learnable by a neural network, yet informative enough to support complex decision-making.

Adjusting the number of past log-returns (D) could influence performance. Reducing D simplifies the state and may speed up training, but risks making the agent short-sighted.

Increasing D could give better insight into longer trends, but may also introduce noise or slow learning.

The optimal D depends on the volatility and memory patterns in the price data.

Shrinking D (e.g. use 3 returns instead of 6):

✅ Pros:

Smaller state = simpler model = faster learning

Less chance of overfitting

❌ Cons:

May miss price trends or volatility patterns

Agent becomes short-sighted

Expanding D (e.g. use 10 returns):

✅ Pros:

More info on market trends or momentum

Might help agent see longer-term behavior

❌ Cons:

Larger state = more complex = slower learning

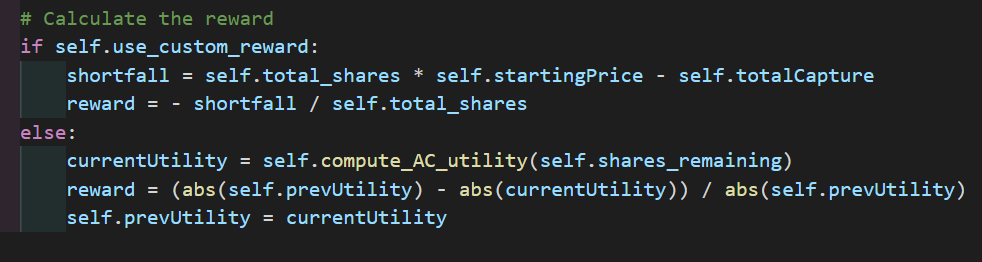
Could confuse the agent with irrelevant noise

**Q2:**

To evaluate the impact of different reward functions in training a reinforcement learning agent for optimal portfolio liquidation, several modifications were made to the existing codebase. The primary goal was to enable the comparison between the original Almgren–Chriss (AC) utility-based reward and a custom reward defined as the negative of the normalized implementation shortfall.

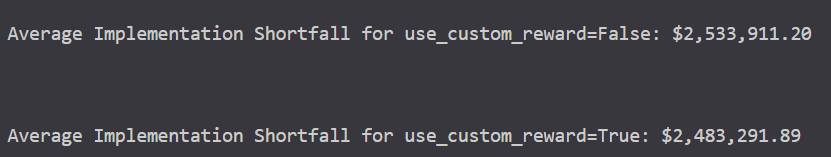
In the “syntheticChrissAlmgren.py” file, a new parameter named “use\_custom\_reward” was introduced in the “MarketEnvironment” class constructor. This boolean flag allows the environment to dynamically switch between the default and custom reward mechanisms. Additionally, the “reset()” method was updated to accept and retain the value of “use\_custom\_reward” during environment resets, ensuring consistent reward logic throughout training and evaluation.

The reward calculation logic inside the “step()” method was modified to respect this flag. If “use\_custom\_reward” is set to “True”, the reward is computed as the negative of the current shortfall divided by the total number of shares—directly encouraging the agent to minimize realized loss. Otherwise, the original reward function based on the change in AC utility is used, which penalizes variance as well as expected cost.



In the “4-DRL.ipynb notebook” file, a testing function called “test\_agent(use\_custom\_reward)” was added. This function instantiates the environment with the appropriate reward setting and evaluates agent performance over multiple episodes. By running this function with “use\_custom\_reward” set to both True and False, it becomes possible to compare the behavior and effectiveness of agents trained under different reward formulations.

These changes make the training and evaluation pipeline more flexible and enable experimentation with alternative reward signals without modifying the core environment logic. As a result of this setup, we were able to compare the agent’s performance under two different reward functions. When using the default Almgren–Chriss utility-based reward (“use\_custom\_reward=False”), the agent achieved an average implementation shortfall of **$2,533,911.20**. In contrast, when trained with the custom reward function based on normalized shortfall (“use\_custom\_reward=True”), the agent achieved a lower average implementation shortfall of **$2,483,291.89**. This suggests that the custom reward function may provide clearer incentives aligned with the financial objective of minimizing execution costs.



**Q3:**

To investigate the effect of sparse rewards on agent performance, we modified the environment to support two modes of reward distribution: a **dense reward** setting where the agent receives feedback after every action, and a **sparse reward** setting where the agent receives a reward only at the end of the episode. This change was implemented by adding a “sparse\_reward” flag to the environment. When enabled, intermediate rewards are suppressed and only the final reward (based on total implementation shortfall) is returned. In our experiments, the agent trained with dense rewards achieved an average implementation shortfall of **$2,483,291.89**, while the same agent under sparse reward conditions resulted in a higher shortfall of **$2,534,694.13**. This suggests that dense rewards provide better learning signals, allowing the agent to more effectively minimize trading costs. Sparse rewards, while potentially useful in some reinforcement learning scenarios, may hinder convergence unless paired with additional exploration strategies or more advanced algorithms.

