Equity Day-trade Agent with Reinforcement Learning

Yang Fan, Christopher Lazarus, Aman Sawhney
Stanford University

Abstract

Optimal trading agents must extract information from the market, manage risk, and manage execution costs. Only through the balance of all three objectives, a optimal trading agents is able to maintain consistent alpha. Given the multifaceted objective, most quantitative strategies have to be created in stages. However, this may lead to inconsistencies within a trading system that can throw the balance of the three objectives off. Given the issues surrounds traditional trading systems, we explore an alternative method of model creation that can balance all objectives at the same time: Reinforcement Learning. In this paper we demonstrate that the problem of optimal trading is a Markov Decision Process, provide an overview of different reinforcement learning techniques, discuss different methods of data augmentation, find relevant features, and create profitable trading agents.

1 Introduction

Optimal trading agents must learn to incorporate stock market features to extract an informational edge on the overall market. Using asset features, either through insider data or improved market information extraction optimal trading agents must discover an edge over the broader market. Before the digital age, most trading agents learn through real world experience. In general, before mathematical finance, most academics could not out perform practitioners in the stock market. Instead, learned experience dominated the market. Furthermore, most human traders today rely very little on academic knowledge. Successful traders learn by trading and accessing the quality of their trades.

Given the history of stock market practitioners learning trading through experience, we attempt to mimic this learning technique using reinforcement learning. Reinforcement learning is a subset of machine learning that attempts to make use of temporal return signals for the training of optimal actions. Besides the similarities with previous agent trading methods, exploring reinforcement learning for the creation of optimal trading agents is also interesting because unlike the majority of quantitative trading systems, reinforcement learning agents can solve the problem of optimal trading using only one model. Normally, given an alpha signal a quantitative trader would have to construct a trading system using this alpha signal and manage the risk of the trading system. Given the non-interpretable nature of modern statistical method, implementers rarely understand the underlying economic reason behind their alpha signal's informational edge. Hence, it is impossible to implement many alpha signals in an optimal fashion. However, in the reinforcement learning setting, the model learns more than an alpha signal. The model learns in an end to end fashion both alpha signals, the implementation details of those signals, and risk management. Given this advantage of reinforcement learning models for optimal trading, we naturally apply such models for the optimal trading problem. Specifically, we demonstrate that the problem of optimal trading is a Markov Decision Process, provide an overview of different reinforcement learning techniques, discuss different methods of data augmentation, find relevant features, and create profitable trading agents.

2 Reinforcement Learning

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a framework to model sequential processes that involve actions, uncertainty and outcomes. MDPs are formalized as a tuple comprised by states, actions, transitions and rewards. At every step, an agent in state s takes action a which with transition probability $T(s, a)$ leads to state s' and yields reward $R(s, a, s')$.

MDPs are naturally fit for modelling financial problems where at a given time an agent must select different actions...

Rewards depend on the actions taken by the agent, different actions at different times can lead to radically different results. Solving an MDP involves identifying a policy $\pi^*(s) s \in S \rightarrow a \in A$ which is a function that prescribes an action for every state such that the expected reward is maximized.

Dynamic programming-like algorithms can be used to solve MDPs when the transition function is known T. The value iteration or policy iteration algorithms, which basically consist of a sequential application of the bellman operator have theoretical optimality guarantees. However, when the transition function is not known (or when the state and action spaces are large or continuous) these algorithms cannot practically solve an MDP. This situation, which occurs naturally in many real-world problem, such as trading can be addressed using Reinforcement Learning (RL).

2.2 Reinforcement Learning

Reinforcement Learning is a branch of machine-learning that is comprised by a set of techniques and algorithms designed to solve MDPs when the transition function is not known. As the name suggests, is involves learning by trial and error. RL algorithms prescribe the actions that an agent undertakes as he interacts with an environment while attempting to balance exploration and exploitation with the aim of producing a policy $\pi(s)$ which maps every state to an action and ideally maximizes the total expected reward.

The canonical reinforcement learning set-up consists of an environment in which an agent observes a set of states, $\mathcal{S}$, and can interact with the environment through a set of actions $\mathcal{A}$. For each step, the agent takes in observations from the environment (i.e. the state), and decides an action in accordance to a policy, π. The policy function, $\pi(a|s)$, denotes a probability distribution over all possible actions given a state s. Additionally, at each step, the agent receives a scalar reward. The total return of the policy is a function of the accumulated reward of the agent across all time steps. Total return can be represented as a state-value function $V(s)$, which denotes the total expected reward from a given state, or an action-value function $Q(s, a)$, which denotes the expected return of taking action a when in state s. The objective is thus to find a policy that will result in the greatest possible return from any of the starting states in the environment.

At the highest level, RL algorithms can be broadly categorized in two groups: model based and model free. Model based algorithms reconstruct the MDP and then use MDP solvers and are limited, among other things, by the memory and computational cost of exactly solving and MDP, which limits their applicability to problems with small state and action spaces. Model free algorithms do not reconstruct the environment's transition function but rather craft a policy in a roundabout way. In the context of financial instruments a transition function would typically correspond to a model that governs the price of an instrument. Clearly, for a real application, such models would be highly complex which leads us to focus our attention on model-free methods.

RL algorithms can also be classified by the type of mathematical framework the use to solve the problem. In this regard, there are two main families of methods: Dynamic Programming methods such as Policy Iteration, Value Iteration and Q-Learning which perform Bellman back-ups to compute value functions and state-action value functions and from there construct a policy. And, on the other hand, Policy Optimization methods which aim to maximize the expected sum of rewards.

Our MDP formulation for this trading scenario combines characteristics of problems best suited for both families of methods. The action space has been heavily discretized and consists of only 3 actions: buy, sell and hold. The state space is a vector that lives in a continuous space. We will use both value-based and policy search methods to train our agent.

2.3 Value-based Method: Deep Q-Learning

Deep Q-Learning is a version of Q-Learning that uses a Deep Neural Network to approximate the Q-function.

One approach to RL involves constructing a Q-function that estimates the value of performing each possible action at each possible state. The Q-function has the following structure: $Q : S \times A \rightarrow \mathbb{R}$, and represents the quality of a state-action combination:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s' \mid s, a) U(s') \tag{1}$$

When the MDP is known, the Q-function can be computed using dynamic programming as shown in the Bellman Equation below:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a') \tag{2}$$

Optimal policies can be extracted after performing the above computation and taking an action

$$a^* = \arg\max_a Q(s,a) \tag{3}$$

that maximizes the Q-function for a given state.

The Q-function can also be estimated using simulators or historic data as in the popular Q-learning algorithm [1]. Q-learning is a model-free reinforcement learning algorithm, and involves applying incremental updates to estimations of the Bellman Equation (2):

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \tag{4}$$

$$= R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a} Q(s',a') \tag{5}$$

The key observation is that instead of using the transition function T and the reward function R, we use the observed state s, next state s' and reward r obtained after performing action a. The algorithm is outlined below.

Q-learning consists of initializing the values in the Q-table randomly and then at step t with state s_t, select an action a_t based on the table and an exploration strategy. Then, upon observing the a new state and reward

$$s_{t+1} \sim T(\cdot \mid s_t, a_t), \qquad r_t = R(s, a_t) \tag{6}$$

the table is updated as follows:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)) \tag{7}$$

where α is the learning rate.

Q-learning as described above enables the estimation of tabular Q-functions which are useful for small discrete problems. However, our setting is better described with a continuous state space. The problem with applying tabular Q-learning to these larger state spaces is not only that they would require a large state-action table but also that a vast amount of experience would be required to accurately estimate the values. An alternative approach to handle continuous state spaces is to use Q-function approximation where the state-action value function is approximated which enables the agent to generalize from limited experience to states that have not been visited before and additionally avoids storing a gigantic table.

There exist a variety of approaches to approximate the Q-function. Some rely on local information and use a notion of distance to interpolate between previously visited states, while others perform global approximation. DNNs have been used successfully as value function approximators [2, 3].

Learning a policy with value function approximation consists of specifying a family of functions that depend on parameters and then finding the parameters that approximate the optimal Q-function by interacting with the environment. This idea has made it possible to use state-action value function algorithms in higher dimensional problems and even problems with continuous states such as ours. After its introduction in [2] for RL tasks, a common approach is to use DNNs as function approximators.

We approximate the state-action function Q by a deep neural network, and we refer to the learned state-action function as $\hat{Q}$. The parameters of the network are learned using the experience tuples (s, a, r, s'), by following the policy extracted from the current Q-network.

We follow a similar ϵ-greedy strategy as [2] to encourage exploration during the learning phase, in the sense that the policy we follow during the learning process is

$$\pi(s) = \begin{cases} \arg\max_{a \in A} \hat{Q}(s,a) & \text{with probability } 1 - \varepsilon \\ \mathcal{U}(A) & \text{with probability } \varepsilon, \end{cases}$$

where $\mathcal{U}(A)$ denotes a uniformly sampled action from the action set A.

As outlined in [2] deep neural networks used to approximate the state-action function in a reinforcement learning problem suffer from stability issues. A source of instability is the non-stationary distribution of samples, as the distribution

of samples depends on actions dictated by the policy extracted from the network which is constantly updated in the learning procedure. Additionally, samples are intrinsically correlated with the Q-network which increases the variance in the loss estimation. To address these issues, we used the target network scheme introduced in [2].

The parameters of the target network are updated less frequently than the parameters of the value network. In addition, we used a memory buffer to implement experience replay, again analogous to the one introduced in [2]. The memory buffer enhances the learning procedure by breaking the dependency between samples in a trajectory and acting as a regularization of the policy over time by keeping samples drawn from different versions of the policy.

After learning $\hat{Q}$, we extract a policy at a given state s by maximizing $\hat{Q}$ over the action space:

$$\hat{\pi}(s) = \arg\max_{a \in A} \hat{Q}(s, a)$$

2.4 Policy Optimization Methods

Policy Optimization methods which aim to maximize the expected sum of rewards

$$\max_{\theta} U(\theta) = \max_{\theta} E\left[\sum_{t=0}^{H} R(s_t) \mid \pi_\theta\right]$$

by considering U as a black box and sampling over a population of possible policies which are parametrized by θ. There are three main motivations to use Policy Optimization methods when compared to Dynamic Programming. First, it is often the case that the policy π is simpler than $V(s)$ or $Q(s, a)$. Second, in order to use $V(s)$ to prescribe an action a dynamics model must be known and computing one Bellman back-up is required to select the optimal action. Third, even if Q is known there are often no efficient ways to solve $\arg\max_u Q_\theta(s, u)$, particularly in high dimensional or continuous action spaces.

A natural first approach to this problem would be to use zero-order methods such as the cross-entropy method to find the policy π that maximizes U by ignoring all other information collected during one episode. This apparently naive method (CEM) has shown promising results [4]. However, CEM and other zero-order methods do not scale well with the dimensionality of θ such as when the model is represented by a complex model like a neural network. On the other hand, computing gradients can be very hard if the environment is stochastic because the noise is likely to dominate.

A way to address these problems is to compute the likelihood ratio policy gradient. Let τ denote the state-action sequence $s_0, u_0, \ldots, s_H, u_H$. Let $R(\tau) = \sum_{t=0}^{H} R(s_t, u_t)$. Then,

$$U(\theta) = E\left[\sum_{t=0}^{H} R_{\pi_\theta}(s_t, u_t)\right]$$

and the goal becomes to find θ such that $\max_\theta U(\theta) = \max_\theta \sum_\tau P(\tau \mid \theta) R(\tau)$. Then the expected sum of rewards $U(\theta)$ can be expressed as the sum of all possible trajectories τ weighted by their corresponding reward and the probability of following that trajectory. Luckily the gradient of $U(\theta)$ can be computed as follows:

$$\begin{aligned}
\nabla_\theta U(\theta) &= \nabla_\theta \sum_\tau P(\tau \mid \theta) R(\tau) \\
&= \sum_\tau \nabla_\theta P(\tau \mid \theta) R(\tau) \\
&= \sum_\tau \frac{P(\tau \mid \theta)}{P(\tau \mid \theta)} \nabla_\theta P(\tau \mid \theta) R(\tau) \\
&= \sum_\tau P(\tau \mid \theta) \frac{\nabla_\theta P(\tau \mid \theta)}{P(\tau \mid \theta)} R(\tau) \\
&= \sum_\tau P(\tau \mid \theta) \nabla_\theta \log P(\tau \mid \theta) R(\tau)
\end{aligned}$$

Which means that $\nabla_\theta U(\theta)$ can be expressed as the expectation under the current policy of $\nabla_\theta \log P(\tau \mid \theta) R(\tau)$ which can be empirically estimated from rollouts under π_θ as:

$$\nabla_\theta U(\theta) \approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)} \mid \theta) R(\tau^{(i)})$$

Which yields the very surprising result that the gradient of the policy can be estimated directly from rollouts. This result is valid even if the reward function is discontinuous because by considering a stochastic policy $U(\theta)$ is smoothed out. Another important observation is that a dynamics model is not needed in this computation. The idea is then to start with a policy π_θ and do rollouts to estimate $\nabla_\theta U(\theta)$ and update the policy using the gradient. The intuition is that updating the policy with the gradient will increase the probability of trajectories with positive R and decrease the probability of trajectories with negative R. The bigger the reward, the more the probability of a trajectory is increased, this increase is encoded in the new values of θ which parametrize the model that define the policy. A key idea here is that this updates only modify the probabilities of experiences paths, it does not try to change the policy to go to unexplored paths. This derivation yields an unbiased yet noisy estimator and the few modifications exists to reduce the variance and increase the sample efficiency. Some modifications incorporate ideas from value-based methods such as the actor-critic algorithm.

3 Data Augmentation

Even though RL lends itself naturally to model financial decision making, its use is complicated because exploration is highly costly. Additionally, there is only one history of financial time series and RL algorithms are very data hungry. Often, one would have access to a simulator that would enable the agent to interact with the environment freely and collect a vast amount of experience tuples. This is not the case for our setting. In order to provide our agent with more experience tuples we decided to augment our dataset. Recent advances in Deep Learning have shown that it is possible to train models that learn distributions of complex objects using a Generative Adversarial Network (GAN).

Producing synthetic realistic time-series data poses multiple challenges beyond those encountered for other typical GAN tasks such as in the image domain. On top to the distribution of variables at specific location, a time-series model also needs to learn the temporal dynamics that govern a sequence.

A novel framework, TimeGAN [5], aims to account for temporal structure by combining supervised and unsupervised training. The framework is trained to embed time-series by optimizing both supervised and adversarial losses that promote adherence to the observed dynamics in the historical data. This model combines the autoregressive structure of time-series by optimizing two losses. This structure rewards the model for learning a distribution for transitions over time. As GANs typically do, TimeGAN has an embedding component that maps the time-series to a low dimensional latent space to simplify the adversarial space. In this architecture, both the generator and the auto-encoder are trained to minimize the supervised loss that measures how well the model learns the temporal dynamics.

We based our implementation on [6]. The architecture is shown in Figure 1. The networks is composed of four components:

- Embedding (auto-encoder)
- Receovery (auto-encoder)
- Generator (adversarial)
- Discriminator (adversarial)

The auto-encoder components map the input space into the smaller latent space designed to facilitate the learning of the temporal dynamics by the adversarial components. The auto-encoder components are implemented using recurrent neural networks (RNNs).

The adversarial component composed of the generator and discriminator operate on sequential data, the synthetic features are generated in the latent space that model learns.

The reconstruction loss evaluates how well the reconstruction of encoded data works and drives the geometry of the latent space. The unsupervised loss reflects the adversarial interaction between the generator and the discriminator: the generator attempts to minimize the probability that the discriminator classifies its output as false and the discriminator aims to maximize the correct classification of the output. Finally, the supervised loss represents how well the generator estimates the next step in the latent space when receiving a prior sequence.

Combining the three losses produces a model that simultaneously learns to encode features, generate representations and capture temporal dynamics [6].

Training is done in three stages: the reconstruction loss is minimized, the supervised loss is minimized and then both are minimized at the same time.

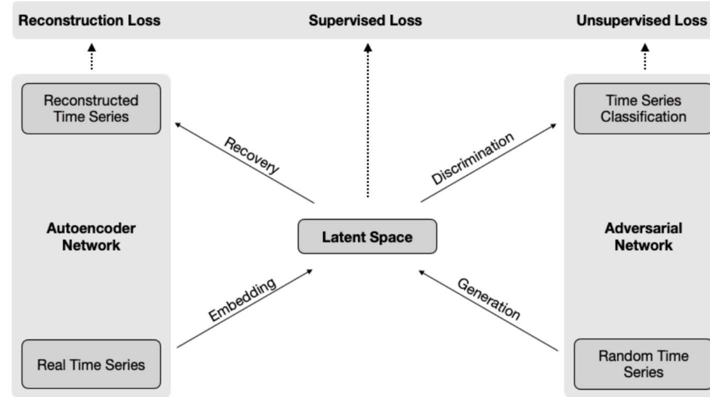

Figure 1: GAN Schematic. Source: [6]

We trained a model using the implementation from [6] on our dataset. Figure 2 shows sample trajectories along with their synthetic counterparts to illsutrate qualitatively the fit of the model.

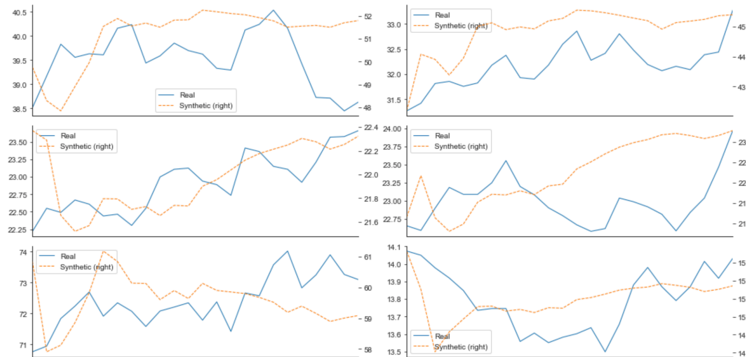

Figure 2: Random sample of trajectories for real and synthetic data.

To further asses the quality of our model we performed a qualitative analysis as suggested in [6]. We projected the real and synthetic data to two different two dimensional spaces. We can observe in Figure 3 that the synthetic data approximately lies in the same regions as the original data which strongly suggests that the model learned a somewhat reasonable distribution.

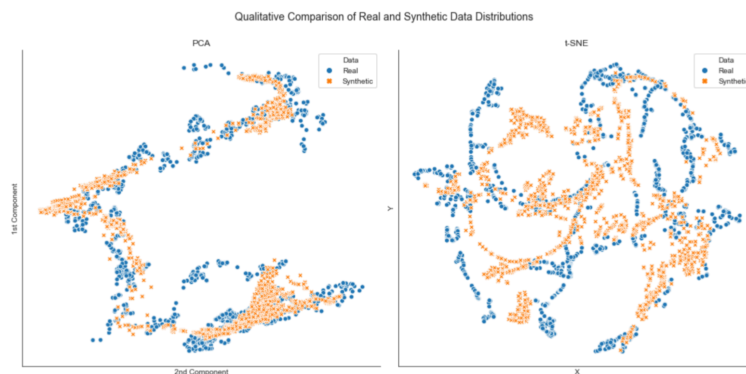

Figure 3: Qualitative Analysis of Results

To continue assessing the validity of our synthetic data generator we performed the quantitative analysis suggested in [6]. For that purpose we trained classification model on both synthetic and real data aimed at distinguishing both sources. Figure 3 shows that the model was trained relatively well as its accuracy and AUC were high for the training set but low for the test sets. This suggests that the real and synthetic data are hard to distinguish.

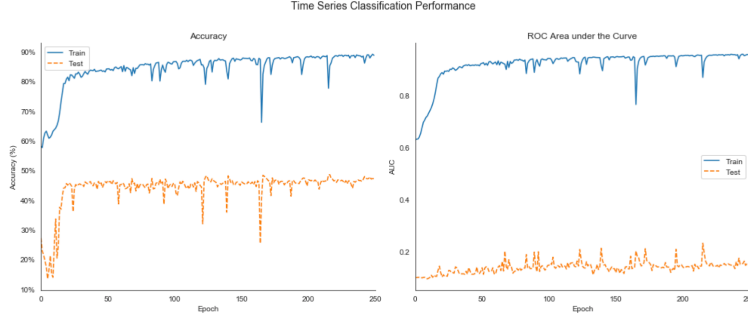

Figure 4: Quantitative Analysis of Results

Unfortunately, due to time constraints we were unable to incorporate our synthetic data to the training procedure of the RL agents. We leave this for future work.

4 Trading Schemes and Reward Functions

4.1 Environment

Since HFT strategies rely on taking and providing liquidity when it is appropriate, we make the modeling assumption that order book dynamics are a markov process. Hence we may formulate a HFT strategy as a Markov Descion Process in the following manner:

- We will assume discrete time intervals which will be determined by our time-scale, T.
- For each time step $0 \leq t \leq T$ we have
  - $s_t := (O_t, q_t)$ where O_t is the order book history at time t over a look back period steps and q_t is the amount of the asset the agent currently holds.
  - $a_t \in \{T, P, N\}$ where T is the act of taking liquidity, P is the act of providing liquidity, and N is the act of doing nothing.
  - r_{t+1} is an appropriate reward function.

This framework generalizes to a multidimensional asset space.

Given this formulation we developed a historical trading simulator using OpenAI's gym interface. We view an episode a random day's price action from 9:30:02 AM to 10:30:02 AM for the particular symbol of interest. Through the length of the episode the agent must place buy and sell orders for the symbol. Given a buy order, the agent will be able to ex cute a buy at the ask for the minimum of the ask size and 100 lots of the security. Given a sell order, the agent will be able to execute a sell at the bid for the minimum of the bid size and 100 lots of a security. Hence, for every transaction the agent must pay the entire bid-ask spread. On top of these parameters we allocate the agent an initial capital of the equivalent to the cash value of 3000 shares at the opening price and limit the agents loss to 30%. Through trial and error we developed the following reward signal

$$r_t = \alpha * 1_{t \neq T \bigcup c_t = 1} \left(r^{a_{t'},t} - \frac{(t - t')}{\gamma} \right)$$

Where c_t is a Boolean representing the presence of a closed position at time t, $(r^{a_{t_1},t_2}$ is the return associated with the closed position, t' is the time in which the position was opened, t is the time in which the position is closed, and α and γ are tunable hyperparmeters. We may view this reward function as the return of an open and closed position adjusted for the holding risk of that action over the time horizon of the position. Hence the agent is incentivized to find actions that result in a high return over a short time horizon.

4.2 Data

We pulled top of the book data from MayStreet aggregated by second, from 9:30AM to 10:30AM for AMD stock from July of 2020 to May of 2021. We reserved the last 45 days of our dataset as a training set. This amounted to a massive data set with well over 100 million rows. Given the data density provide by MayStreet, we were able to avoid using the data augmentation strategies described in the previous section. However, those strategies would be useful for research over longer horizons with less data density.

4.3 Simulator

To train our RL agent we built an environment using Open AI's gym interface. Each episode of training loads a random day and random symbol's data from a train or test time dataframe. Compared to other RL projects, such as FinRl and other RL environments, our environment has the following advantage:

- Homogeneous Trading Horizons: By sampling from the same time of day, as compared to randomly sampling along a stock path, our episodes contain very similar market micro structure
- Data Density: Most other academic projects use daily data. Since we use data aggregated on the second level, the data is far more dense.
- Realism: In the real world, high frequency trading models should be adapt at providing sustainable profit across a variety of symbols. Hence the variety of symbols allows train a more realistic model.

5 Feature Engineering

As the policy function adopts a neural net structure, to facilitate learning, we conduct featuring engineering based on the order book data. This is achieved through adding technical indicators and performing feature reduction through Random Forests.

5.1 Feature selection pool

To be precise, the following order book data is included in at every time step: level 1 - 3 bid and ask price, size, number of providers and adjusted volume, of a total length of $3 * 7 = 21$. The following technical indicators are included:

1. Simple Moving Average (SMA)
2. Exponential Moving Average (EMA)
3. Relative Strength Index (RSI)
4. Price Rate of Change (ROC)
5. Triple Exponential Average (TRIX)
6. Percentage Price Oscillator (PPO)
7. Percentage Volume Oscillator (PVO)
8. Aroon Indicator
9. Detrended Price Oscillator (DPO)
10. Moving average convergence divergence (MACD)
11. Ulcer Index (UI)
12. Close-to-Close Volatility

The listed indicators are only based on the closing price, estimated by the mid price in our case, and adjusted volume. For each of the technical indicators, we varied the time window sizes from 20 seconds to 120 seconds and kept a copy for each. To deal with the issue of exploding gradients, we normalize all the features in both the training and test sets by the following formula, for any feature f:

$$f_{\text{normalized}} = \frac{f - \min(f_{train})}{\max(f_{train}) - \min(f_{train})}$$

In this way, we can clip features in the training set between 0 and 1 range, and roughly set features in the test set to be around that range.

5.2 Feature reduction

To improve on the learning result of the policy network, we performed feature reduction using a Random Forest. We trained a Random Forest Regressor using our train dataset to predict the one minute return using the features listed above. After training the regressor we used MDI to select the most relevant features. The following top 10 features are what we used for the actual training of the model: PVO(120), Close-to-Close Volatility(60/120), AROON(120), KST(60/120), MACD(120), UI(60/120), TRIX(120).

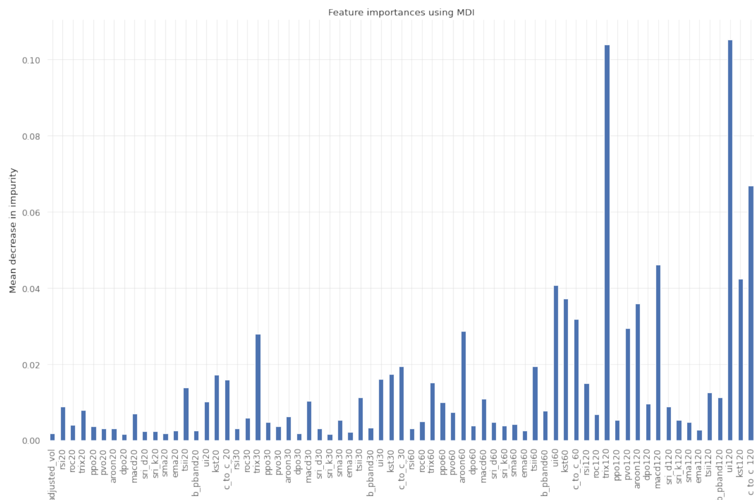

Figure 5: Feature importance from the random forest model

The observation matrix is an $F \times L$ matrix, where F is the total number of features at every time step and L is the number of time steps we allow the agent to look back, including the current step. When training on real stock data, we also attach three vectors of size F, containing information on the agent's past actions, execution prices, and time of the past trades.

6 Experimental Runs

To test the efficacy of our custom made trading environment, which contains the integral part of our trading strategy, we conducted experimental training and trading on a regularly oscillating pattern. To achieve that, we create a counterfactual order book with oscillating linear patterns ranging from 10 to 20, with 0 gap between the bid and ask price. The bid and ask size are set to be 1000, a constant for all entries. The number of bid and ask providers are both set to match the prices. We added technical indicators based on the counterfactual order book data, as defined previously. The maximum holding periods for the agent is set to be 5 seconds in this setting, to adapt to the frequency of price changes. After verifying the validity of our model, we experimented with custom feature extraction networks, based on Long short-term memory (LSTM) and Transformer layers. We define actions to be {0: short, 1: buy, 2: hold}.

6.1 Multilayer Perceptron (MLP) networks

When the agent exhibited stable trading behavior the feature extractor is build with two layers of linear layers. If we zero out reward for non-terminal steps, the agent multiplied the portfolio value by roughly 24 times by making considerable long and short moves.

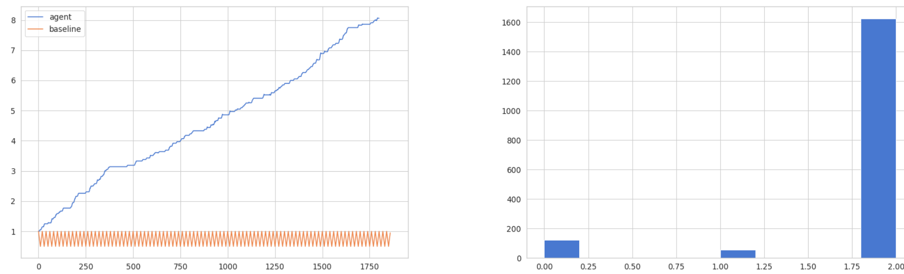

Figure 6: Portfolio Value and Actions for MLP when reward is defined at every step

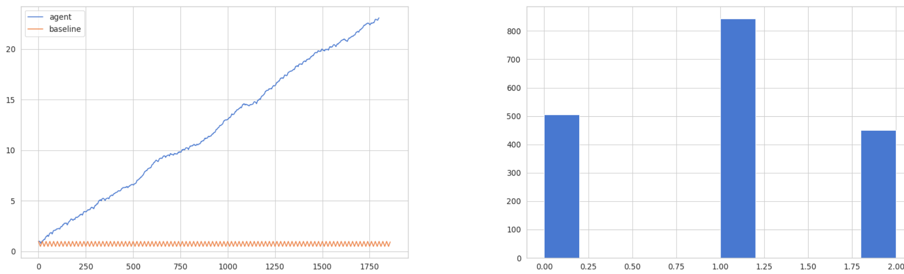

Figure 7: Portfolio Value and Actions for MLP when reward is defined at terminal steps

6.2 LSTM + MLP

We experimented the feature extractor with a LSTM layer connected by two more linear layers before feeding the outputs to the policy networks. The agent exhibits unstable behavior when the reward function is defined at every step, but performed well when we only have rewards defined at terminal states.

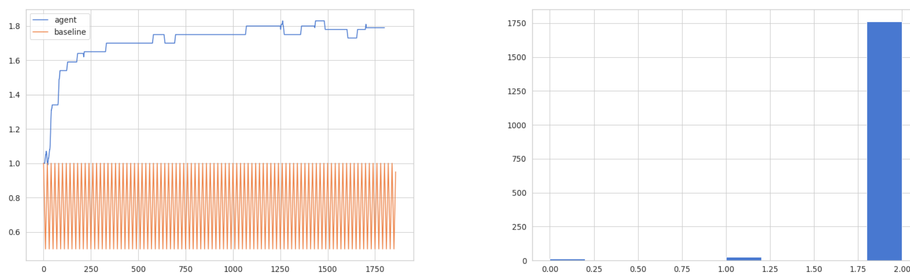

Figure 8: Portfolio Value and Actions for LSTM when reward is defined at every step

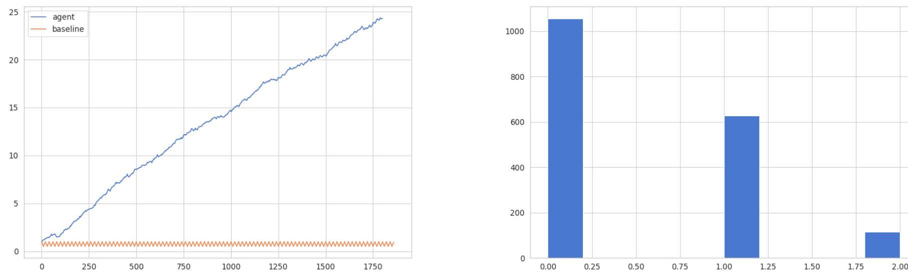

Figure 9: Portfolio Value and Actions for LSTM when reward is defined only at terminal steps

6.3 Transformer + MLP

Inspired by success at predicting returns using transformers [7], we experimented with the configuration of a transformer layer plus two linear layers connected to train the agent on the ZigZag pattern, with both inputs to the encoder and decoder to be the same observation matrix. The transformer module also exhibits unstable behaviour when the reward function is set to be defined for only terminal states, but performed well in the other case.

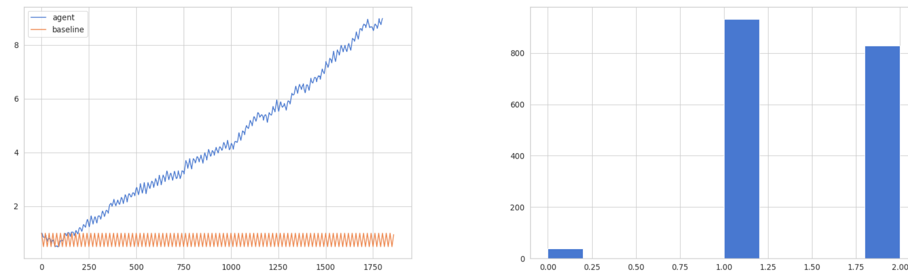

Figure 10: Portfolio Value and Actions for Transformer when reward is defined only at terminal steps

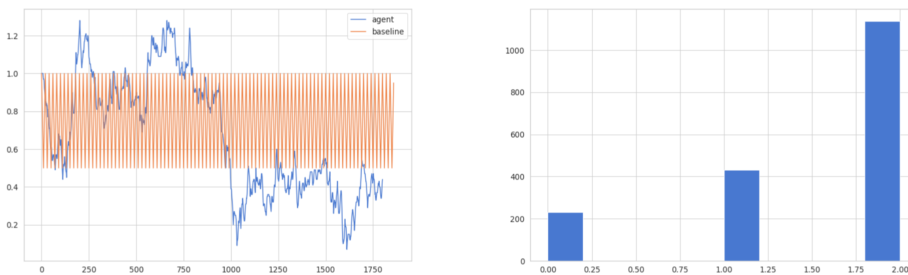

Figure 11: Portfolio Value and Actions for Transformer when reward is defined only at terminal steps

6.4 Takeaways

We made the following deductions from the above experimental runs:

- MLP might be sufficient as compared to more complicated networks, with the added benefit of being more regularized and stable. If needed can always add non-linearity in the feature engineering step.
- There is room to try different reward functions, and it might be the key to the agent's performance.

7 Model performance on Stock data

7.1 Training Hyperparmeters and Algorithm Selection

Through experimentation we discovered that value based reinforcement learning methods performed better than policy based estimates. We suspect this is a result of the discrete nature of our action space, along with the action based definition of the reward function resulted in the value. Hence the results that follow are trained using deep-q learning. For each model we train a multilayer perceptron network with four layers, have the model explore on 60% of both the training and testing time steps, update gradients every 5 timesteps, and let $\gamma = 1$.

7.2 Trading Performance of Various Agents

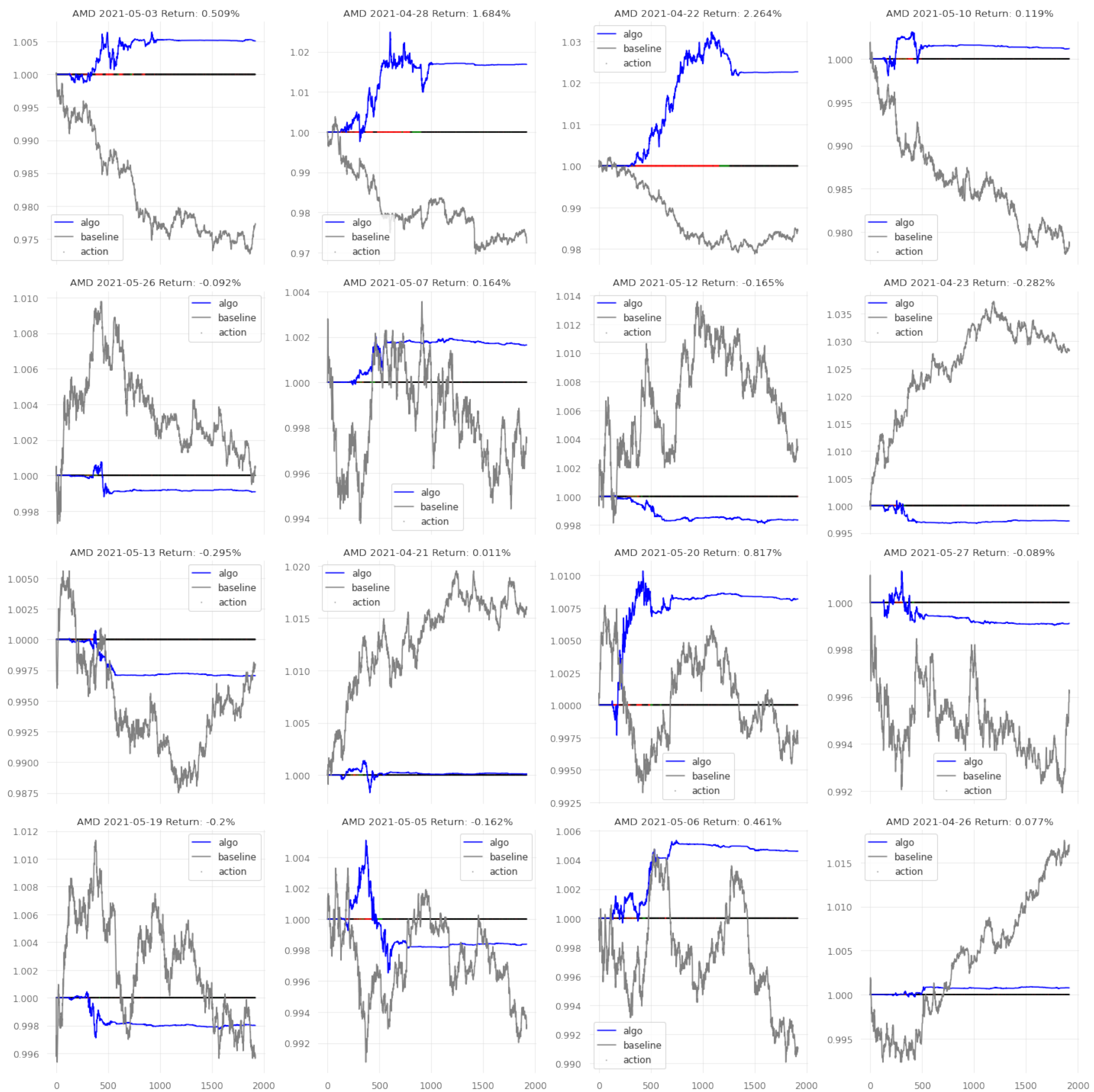

Figure 12: Initial Trading Results For AMD

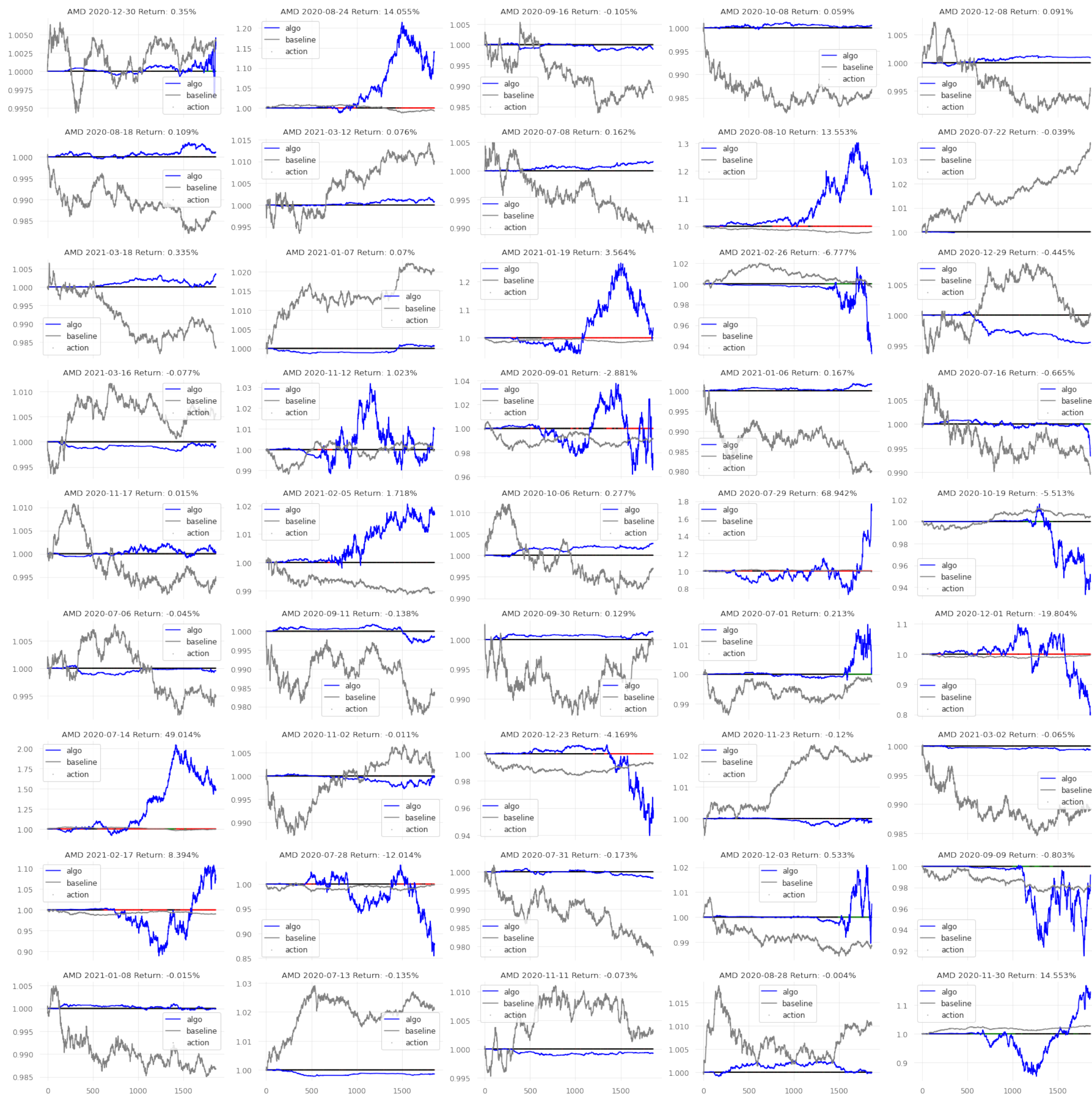

Figure 13: Trading Results For AMD Over 45 Days Of Out of Sample Data

Given our hyperparamters above we trained a deep-q network on AMD using all the features(i.e. we did not use the features subselected by the random forest) and consider the agents actions on a test set of 16 days. On this test set we obtain a Sharpe ratio for the 1 second time period of 0.0131. Annualized, assuming you can only trade half an hour a day, that is a Sharpe ratio of 8.82. Moreover, we can analyze the behavior of the trading agent to gain more insight into its strategy. From Figure 12 it is clear that the agent seems to trade for a short period and then seems to stop either when a small loss is realized or a gain is realized. Through the reinforcement learning process, it is clear the agent has learned to be loss adverse when trading AMD. Given that the agent only trading in short bursts, the resulting Sharpe ratio is high while the realized returns by the agent are not substantially high. Rather only the risk adjusted returns are impressive.

Given the rather remarkable result in figure 13, we trained another deep-q network to trade AMD. However, this time we trained for longer and used the features we sub-selected in section 5. On top of that, we also turned the exploration rate to be 0 during the testing time. See figure 14 for the results of this agent. This agent achieves a Sharpe of 5.86

over 45 days of out of sample data. However, unlike the previous agent, this agent generates substantial return. This agent takes on large risk when the market conditions are appropriate, i.e. less volatile. This results in an agent that often stays out the market. However, when the agent decides to trade actively for certain days, the resulting gains are massive. Given this model, we re-sampled the returns over a daily time period and compared it to the underlying security. With the re-sample our Sharpe is reduced as re-sampling does not capture the same standard deviation. With the re-sample we achieve a Sharpe of 4.55. See figure 16 for metric comparisons with the underlying security and figure 14 for the cumulative daily returns and draw-down. Over the 45 day trading period the strategy achieved a return in excess of 100%. Additionally, the strategy only traded in the first half hour of the training session. Thus if we could generalized the strategy to a longer trading time we could realize much greater returns. Currently we do not enforce the margin requirement for the agent, so it is possible for the agent to keep its leveraged position even when the underlying position is going down, which is possibly unrealistic. In the future, we will also investigate how to incorporate the margin requirement to the agent, while maintaining an outstanding performance.

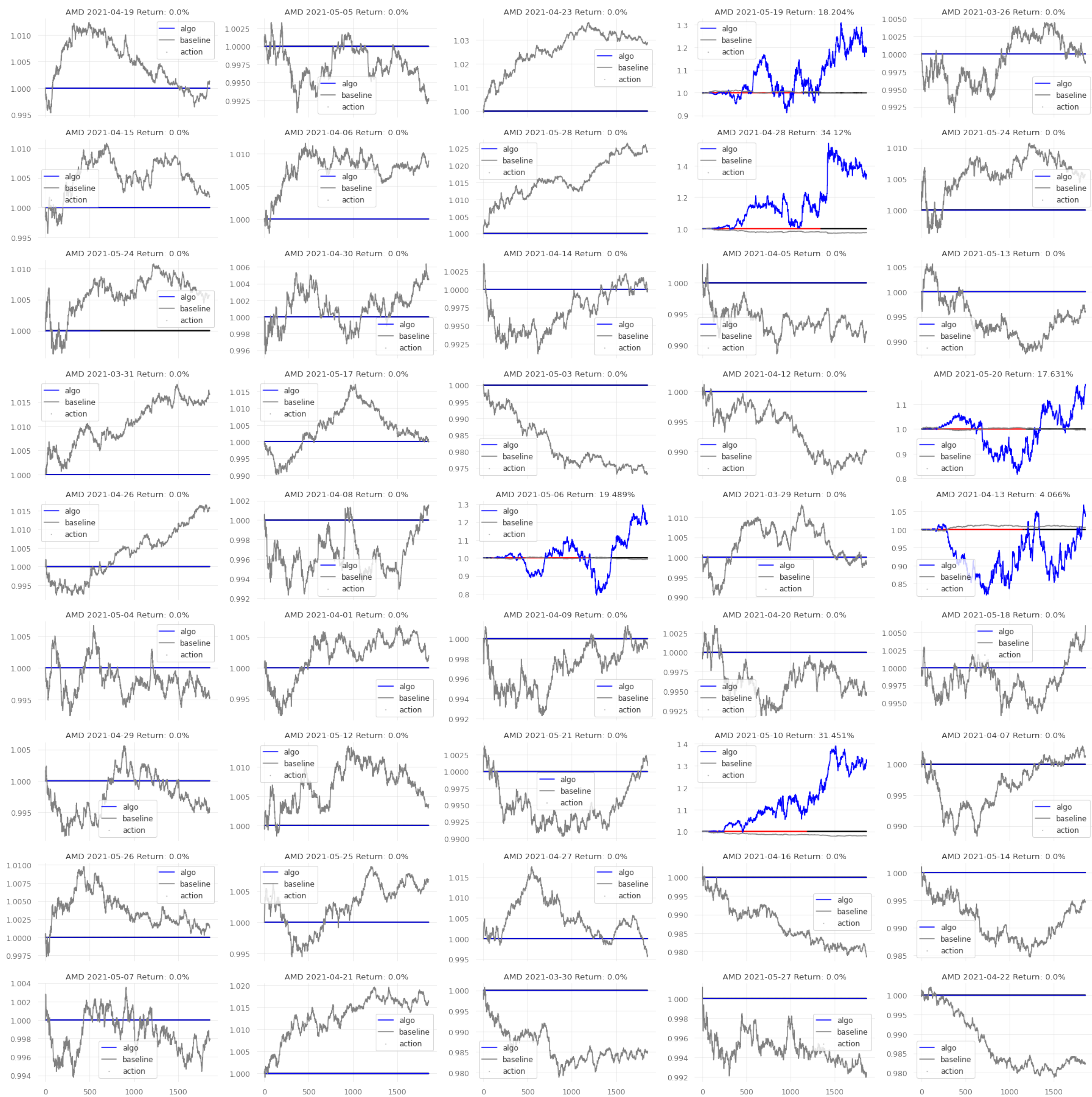

Figure 14: Trading Results For AMD Over 45 Days Of Out of Sample Data

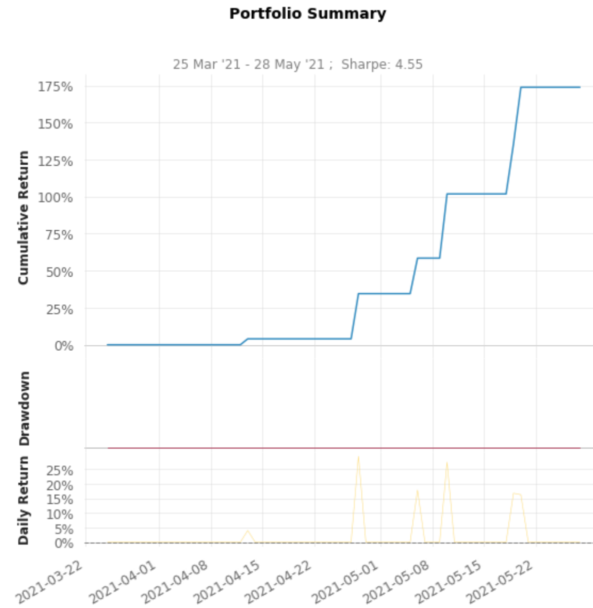

Figure 15: AMD Return and Drawdown Results

Performance Metrics

	Strategy	Benchmark
Start Period	2021-03-26	2021-03-26
End Period	2021-05-28	2021-05-28
Risk-Free Rate	0.0%	0.0%
Time in Market	10.0%	71.0%
Cumulative Return	173.79%	5.06%
CAGR%	34120.18%	33.14%
Sharpe	4.58	0.79
Sortino	inf	1.2
Sortino/√2	inf	0.85
Max Drawdown	nan%	-14.42%
Longest DD Days	-	-
Gain/Pain Ratio	inf	0.17
Gain/Pain (1M)	inf	3.86
Payoff Ratio	nan	nan
Profit Factor	inf	1.17
Common Sense Ratio	inf	1.24
CPC Index	nan	nan
Tail Ratio	inf	1.06
Outlier Win Ratio	9.51	17.12
Outlier Loss Ratio	nan	1.16
MTD	103.55%	-1.89%
3M	173.79%	5.06%
6M	173.79%	5.06%
YTD	173.79%	5.06%
1Y	173.79%	5.06%
3Y (ann.)	34120.18%	33.14%
5Y (ann.)	34120.18%	33.14%
10Y (ann.)	34120.18%	33.14%
All-time (ann.)	34120.18%	33.14%
Avg. Drawdown	nan%	-7.33%
Avg. Drawdown Days	-	-
Recovery Factor	inf	0.35
Ulcer Index	0.8	1.08

Figure 16: Agent's Performance Metrics vs AMD stock

8 Future Research

We plan to continue the project in the following directions:

1. Using the synthetic data generated by GAN, or transfer the encoder network for the feature extractor and see if either of those improves the agent's performance.
2. Train the agent on other stocks and assets in general to let it be an expert at making day-trades.
3. Experiment with other reinforcement learning algorithms, and perform systematic hyperparameter tuning.

9 Conclusion

It is clear from our results that Reinforcement Learning can be used to effectively create profitable trading agents. In this paper we provided the mathematical basis of using reinforcement learning to solve the problem of optimal trading, provided an overview of different reinforcement learning techniques, discuss different methods of data augmentation, find relevant features, and create profitable trading agents. Moreover, we can additionally view reinforcement learning agents as the natural successors of human traders. From this view the possibilities of reinforcement learning as a method of creating trading agents is more clear. Like humans, reinforcement learning agents balance the three objectives of optimal trading agents: information extraction, risk management, and cost management. Thus as markets become more efficient and alpha less readily available, we expect that the quantitative trading community will continue to facilitate this transfer from man to machine.

References

[1] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[3] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[4] Shie Mannor, Reuven Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pages 512–519. AAAI Press, 2003.

[5] Jinsung Yoon, Daniel Jarrett, and Mihaela van der Schaar. Time-series generative adversarial networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[6] Stefan Jansen. https://github.com/stefan-jansen/synthetic-data-for-finance. *Deep RL Workshop, NeurIPS 2020*, 2020.

[7] J. Wallbridge. Transformers for limit order books. *arXiv:2003.00130*, 2020.