# A Comparison of Various Machine Learning Models in a Fully Autonomous Adversarial Stock Market Environment

Jared Pomerantz

November 30, 2025

**Abstract**

The increasing relevance of Machine Learning in stock trading amplifies the need for an adversarial stock market environment where models can trade with one another and be evaluated based on their performance against each other. An environment that provides a framework for this trading was created to perform this study, which examines the performance of different valuation regression models and stock feature encodings, general market trends based on included models, and feature importances [Pom]. It was found that a Multi-Layer Perceptron model that masks the current price tends to vastly outperform other agents without this price masking, and it has the effect of stabilizing growth and potentially preventing "bubbles" in the stock environment. Models that do not have price masking were shown to assign over 50% importance to current price in the valuation of a stock, indicating significant reactivity to the overall market's valuations.

## 1 Introduction

Machine learning has built a notable presence in the stock market as participants attempt to gain an edge. Starting in the 1990s and 2000s, Support Vector Machines and basic Neural Networks were beginning to be used to identify value in the market, and computational advancements in the 2010s increased the capabilities of Deep Learning, which has spurred tremendous interest into the technology [oB25]. Model selection, data feature selection, data preprocessing techniques, and algorithm design are decisions with significant effects on market performance in a stock market environment.

This study examines the performance of multiple Machine Learning and Deep Learning models, how the inclusion of different models affects overall market performance and stability, and feature importances generated by the relevant models.

## 2 Background and Literature Review

Many studies have been performed related to Machine Learning for stock market predictions. It was found that given a variety of advanced stock features, such as stock momentum and exponential moving average, Artificial Neural Networks tend to outperform machine learning models, such as Logistic Regression and Support Vector Machines (SVMs), in the binary classification task of predicting whether stocks will move up or down in real markets, such as the NYSE 100 and NIKKEI 225 [Isk24]. This study also showed that SVMs tend to perform the best among non-Neural Network Machine Learning models.

Additionally, it's been shown that Deep Reinforcement Learning can be used for Algorithmic Trading, which consists of any use of algorithms to automate the trading process [ea24a]. Encoding

the current state of the market such that the Deep Learning model can make the best possible action is an important engineering decision as well. For example, it is possible to encode a set of features as a time series to create a 2-dimensional tensor, and to infer on this input tensor with a Convolutional Neural Network [Mil21].

While it's not directly related to stock markets, it's worth noting the influence that *Playing Atari with Deep Reinforcement Learning* has had on decision-making for Deep Learning in Reinforcement Learning environments. In this study, a Deep Learning model was trained to predict the action-value function, *Q(s, a)*, for different paddle movement actions, *a*, in Atari given the environment state, *s*. Q-Learning consists of inferring with this model for the estimated value for each action, given the state and possible actions, to find the action associated with the optimal value, $Q^*$ [ea13]. Given a set of actions in an RL environment, this study and algorithm provides a framework for acting and learning with a Deep Learning policy.

# 3 Methods

## 3.1 Stock Market Environment

To perform this study, a stock market environment was built to allow the instantiation of stock market participant agents with the ability to trade stocks with one another [Pom]. Generally, the Markov Decision Process framework consists of each agent in the environment taking a single action per timestep, and the execution of each action is independent of the actions taken by other agents [ea25]. However, in a stock market environment, this is not applicable; participants in a real stock market can make multiple orders in a given time period, but those orders only execute if there's a complementary action performed by another agent, such as an agent willing to sell a stock that another agent wants to buy, where both agents are able to meet at a neutral price point. Therefore, in this environment, each agent is able to bid 10 buy orders and 10 sell orders at each timestep, consisting of the following data:

| **Data** | **Description** |
|---|---|
| Stock | The stock to transact. |
| Stock Quantity | The quantity of the stock to transact. |
| Stock Price | The bid or ask price for the order. |

Once these bids are made, the market resolves them by granting the highest priced buy orders and the lowest priced sell orders until all possible orders are resolved [Fer25]. For a buy order and sell order pair, the transaction price evaluates to the average of the buy order price (bid) and the sell order price (ask).

The stocks within the stock market are randomly generated before the simulation. Each stock is characterized by the following features when ingested by a model:

| Feature | Description |
|---|---|
| Price | The last traded price of the stock. |
| Cash | The available cash to invest into Earning Value of Assets. |
| Earning Value of Assets | The stock's capacity to add cash over time. |
| Latest Quarterly Earnings | The amount of cash gained at last earnings report. |
| 1-Day Percent Change | The stock price percent change over 1 day (1 timestep). |
| 5-Day Percent Change | The stock price percent change over 5 days (5 timesteps). |
| 10-Day Percent Change | The stock price percent change over 10 days (10 timesteps). |
| 1-Month Percent Change | The stock price percent change over 1 month (30 timesteps). |
| 3-Month Percent Change | The stock price percent change over 3 months (90 timesteps). |
| 6-Month Percent Change | The stock price percent change over 6 months (180 timesteps). |
| 1-Year Percent Change | The stock price percent change over 1 year (365 timesteps). |
| 3-Year Percent Change | The stock price percent change over 3 years (1095 timesteps). |
| 5-Year Percent Change | The stock price percent change over 5 years (1825 timesteps). |

Every 92 timesteps, each stock reports its quarterly earnings, which consists of sampling an amount based on the following distribution:

$$\mathcal{N}(E, 2000.0)$$

where $E$ is the Earning Value of Assets attribute for that stock. This value is meant to represent the stock's business's ability to earn cash based on their assets and business plan. However, this value depreciates at each timestep by the following value:

$$\mathcal{U}(0, E * 0.03) + 1.5$$

In response, the stock invests its cash to its Earning Value of Assets at each timestep. The amount of cash invested, $I$ is sampled from the following distribution:

$$\mathcal{U}(0, 1000 + max(0, C)/500$$

where $C$ represents the cash held by the stock at that timestep. This investment is converted to Earning Value of Assets by the following distribution:

$$\mathcal{N}(0.7 * L, 0.1) * I$$

where $L$ is the stock's Quality of Leadership, such that $L \in [0.0, 1.0]$. This value is meant to simulate a business's ability to convert cash into meaningful value. Since this value drives the stock's ability to grow, the task of market participants over the long-term is to implicitly choose stocks that are highest in this value.

Another market parameter is the risk-free rate of return. In the real world, this is approximately equal to the interest paid on a 10-year government Treasury note [Vipwn]. In this stock market environment, cash compounds at each timestep such that it grows by this percentage each year, or 365 timesteps. If $r$ is the risk-free rate of return, the cash compounds each timestep by the following equation:

$$C_{t+1} = C_t * (1 + r)^{(1/365)}$$

Additionally, random noise is injected to the stock prices at each timestep to initiate more trading volume. At each timestep, the amount of noise that is added to each stock is sampled from the following distribution:

$$\mathcal{N}(0, \nu_{stock} * P_{stock})$$

where $\nu_{stock}$ is the volatility parameter of the stock being effected, and $P_{stock}$ is the stock's current price.

Finally, when a stock is delisted, its price is reassigned to 0, and it can no longer be traded. Stocks will be delisted from the market if one of the following conditions is true in a given timestamp:

1. The stock price moves below 1.0. On the New York Stock Exchange, if a stock remains below $1.00 for 30 days, it is delisted from the Exchange [Sto21].

2. The stock's cash, $C$, is negative and its Latest Quarterly Earnings are less than the following threshold $T$:

   $$T = C * \frac{r}{4}$$

   $T$ represents the amount that the debt will grow in a quarter. If this value becomes larger than the stock's Latest Quarterly Earnings, it represents that the debt can no longer be serviced, leading to bankruptcy.

## 3.2 Models

The use of many independent machine learning models and neural networks in this RL environment creates the need for lightweight models if to avoid reliance on expensive hardware [Xu25]. Therefore, variants of Transformers and other large models will not be available. At a high level, the models in this study aim to predict the price of stocks in 30 timesteps. Therefore, regression models are used. The chosen models are described below.

### 3.2.1 Random Forest Regressor

A Random Forest is an ensemble of independent Decision Trees, each trained to optimize information gain at each Tree's node, given input features and random noise injection [ea22]. Each Decision Tree in the Random Forest has voting power and contributes to the final output value. In this implementation, which used Scikit-Learn's RandomForestRegressor function [SL25b], had the following parameters:

| Parameter | Value |
|---|---|
| n_estimators | 100 |
| criterion | "squared_error" |
| max_depth | None |
| min_samples_split | 2 |
| min_samples_leaf | 1 |
| random_state | 0 |

All other parameters were set to the default values specified in the source code. It's worth noting that this model type does not allow for incremental learning, therefore it must be retrained from initialization at every update.

### 3.2.2 Passive Aggressive Regressor

The Passive Aggressive Regressor is a linear model capable of Incremental Learning, which consists of loading a pre-trained model and fine-tuning the model over the course of a simulation without fully forgetting past knowledge [ea20b]. It aims to strike a balance between "passivity" and "aggression" by optimizing the new model weight $w_t$ at each timestep to minimize the following loss function:

$$\mathcal{L} = \frac{1}{2}||w_t - \hat{w}_{t-1}||_2^2 + C\ell(y_t, x_t^\top w_t; \epsilon)$$

$\hat{w}_{t-1}$ is the model weights vector at time $t-1$, where $C$ is an aggressiveness parameter set by the user, $\ell$ is the loss function, $y_t$ is the target vector at time $t$, $x_t$ is the input vector at time $t$, and $\epsilon$ is the insensitivity of the loss function [ea15]. Setting $C$ to be extremely low would create a completely passive, unchanging model, while setting it high would cause the model to make more aggressive changes in favor of reducing the loss $\ell$. In this study, Scikit-Learn's PassiveAggressiveRegressor function is used [SL25a]. In this implementation, $C = 1.0$, promoting a balance between passivity and aggressiveness.

### 3.2.3 Multi-Layer Perceptron

The Multi-Layer Perceptron is a basic neural network that consists of an input layer, an arbitrary number of hidden layers, and an output layer, each separated by non-linear layers [ea09]. At the end of the perceptron is an activation layer that converts the embeddings to the proper data type, such as logits for a classification problem or a certain domain for a regression problem. For this study, the following architecture was used for the Multi-Layer Perceptron:

| Layer Number | Layer Type | Features |
|---|---|---|
| 1 | Linear Layer | 14 in-channels, 16 out-channels |
| 2 | ReLU | - |
| 3 | Linear Layer | 16 in-channels, 32 out-channels |
| 4 | ReLU | - |
| 5 | Linear Layer | 32 in-channels, 1 out-channel |
| 6 | Exponential Activation Function | Outputs $exp(value/Z)$ |

where $Z$ is a stabilization constant that prevents computationally unstable outputs from the exponential function when the model is initialized; for this model, $Z = 1000$. The Exponential Activation is meant to map the outputs of the regressor to the domain $[0, \infty)$, as stocks should not be assigned a negative value in this environment. This model is incredibly lightweight: only approximately 8 KB. This drastically reduced the load on the NVIDIA GeForce MX150 GPU that was used to complete this study. Additionally, this model is capable of undergoing Incremental Learning. Finally, to vary the models from each other at the beginning of each simulation, model parameters are injected with noise from the following distribution:

$$\mathcal{N}(0, 0.05).$$

### 3.2.4 Multi-Layer Perceptron with Price Masking

This variant of the Multi-Layer Perceptron is identical to the one defined above, but the only difference is that it masks the stock's current price in the feature vector. This forces the model to determine a stock's valuation without being influenced by rest of the market's current valuation of the

stock. All stock prices are replaced with -1.0. Finally, models are injected with noise from the same distribution as the standard Multi-Layer Perceptron above when instantiated.

## 3.3 Policy

### 3.3.1 Action Selection

Actions were determined by using a Top-N Greedy Q-Learning Policy. To determine buy orders at each timestep, the agent samples a predefined $N_{sample}$ stocks from the market (or the total number of stocks $N_{market}$ in the market if this is less than $N_{sample}$) to be valued by its model. A placeholder stock representing cash is also included; when the "cash stock" is encoded into features, it includes percent-change values that are reflective of the risk free rate of return, $r$, which makes holding cash more attractive to the model when $r$ is high. Since an agent sees its cash compound over time by an annual rate of $r$, the model should recognize that the opportunity cost of buying stocks rises with $r$, just as a real investor would recognize this higher opportunity cost as a result of monetary policy [Hay23]. Once the stocks are sampled and encoded into features, the model assigns each stock a projected 30-timestep valuation. Stocks are then sorted in descending error by their projected percent change. The agent iterates through this list, creating a buy order for each stock, until it reaches the cash stock, (indicating that it values the future growth of cash over the growth of the available stocks) or until it reaches the buy order limit, 10 in this study. The bid prices in the buy orders are deterministically set as 10% of the difference from the stock's current price to the valuation.

**Algorithm 1** Top-N Greedy Q-Learning Buy Order Policy

$N \leftarrow min(N_{sample}, N_{market})$
$\mathbf{S}_{market} \leftarrow$ all stock feature encodings in the market.
$\mathbf{S}_{sample} \leftarrow$Sample $N$ stocks from $\mathbf{S}_{market}$.
$S_{cash} \leftarrow$ the stock feature encoding for cash.
$\mathbf{S}_{sample} \leftarrow$ **concatenate**$(\mathbf{S}_{sample}, S_{cash})$
$\hat{V} \leftarrow$ the valuation model
**for** $i \leftarrow \{1, 2, ..., N+1\}$ **do**
    $V_i \leftarrow \hat{V}(\mathbf{S}_{sample_i})$
    $P_i \leftarrow P_{stock}$
    $\delta_i \leftarrow (V_i - P_i)/P_i$
**end for**
Sort $\mathbf{S}_{sample}$ and $V$ indices by $\delta$, **descending**.
Let $O_{buy}$ be a list of buy orders.
$i \leftarrow 0$
**while** $i < N$ and $\mathbf{S}_i$ is not $S_{cash}$ **do**
    Current Stock $S_{current} \leftarrow \mathbf{S}_{sample_i}$
    Quantity $q \leftarrow 1$
    Bid $B \leftarrow P_i + (V_i - P_i)/10$
    Append $(S_{Current}, q, B)$ to $O_{buy}$
    $i \leftarrow i + 1$
**end while**
    **return** $O_{buy}$.

A similar but inverse approach is taken for sell orders. The agent samples $N$ stocks from the market (or the total number of stocks in its portfolio if this is less than $N$) to be valued, including the "cash stock". The model projects the valuations in 30 timesteps, calculates the projected percent changes, and sorts the stocks in *ascending* order by projected percent change. The agent iterates through this list, creating a sell order for each stock, until it reaches the cash stock (indicating that

it values the future growth of cash *lower* than the growth of its stocks and should therefore not trade the stocks for more cash) or until it reaches the sell order limit, 10 in this study. The asking prices in the sell orders are deterministically set as 10% of the difference from the stock's current price to the valuation.

**Algorithm 2** Top-N Greedy Q-Learning Sell Order Policy

$N \leftarrow min(N_{sample}, N_{portfolio})$
$\mathbf{S}_{portfolio} \leftarrow$ all stock feature encodings in the portfolio.
$\mathbf{S}_{sample} \leftarrow$Sample $N$ stocks from $\mathbf{S}_{portfolio}$.
$S_{cash} \leftarrow$ the stock feature encoding for cash.
$\mathbf{S}_{sample} \leftarrow$ **concatenate**$(\mathbf{S}_{sample}, S_{cash})$
$\hat{V} \leftarrow$ the valuation model
**for** $i \leftarrow \{1, 2, ..., N+1\}$ **do**
    $V_i \leftarrow \hat{V}(\mathbf{S}_{sample_i})$
    $P_i \leftarrow P_{stock}$
    $\delta_i \leftarrow (V_i - P_i)/P_i$
**end for**
Sort $\mathbf{S}_{sample}$ and $V$ indices by $\delta$, **ascending**.
Let $O_{sell}$ be a list of sell orders.
$i \leftarrow 0$
**while** $i < N$ and $\mathbf{S}_i$ is not $S_{cash}$ **do**
    Current Stock $S_{current} \leftarrow \mathbf{S}_{sample_i}$
    Quantity $q \leftarrow 1$
    Bid $B \leftarrow P_i + (V_i - P_i)/10$
    Append $(S_{current}, q, B)$ to $O_{sell}$
    $i \leftarrow i + 1$
**end while**
    **return** $O_{sell}$.

### 3.3.2 Valuation Model Initialization

To provide a stable market initialization, the price-observant models are trained on a dataset mapping standard stock features to their current price as labels. The price-masked Multi-Layer Perceptron is trained on a dataset mapping the stock features to a random price from $\mathcal{U}(10, 1010)$ to fit its outputs to a realistic distribution.

### 3.3.3 Valuation Model Learning

At each timestep, the randomly sampled stock features for the buying algorithm are inserted into the replay memory, $M$ [ea20a]. Valuation model parameters are updated at each timestep by fitting this past sample of stock features, 30-timesteps ago, with the stock prices at the current timestep. The squared Bellman error is used to calculate the gradients for the Deep Neural Network policy, as follows [ea18]:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(\hat{V}(S_i) - V(S_i))^2$$

where $\hat{V}$ is the 30-timestep valuation model estimate for each stock $S_i$ for $i \in [1, 2, ..., N]$, and $V$ is the true valuation.

Among the models listed above, the Multi-Layer Perceptron and PassiveAggressiveRegressor are capable of performing Incremental Learning, meaning that these regressors are able to adjust their

valuations over time without re-initialization. The Random Forest Regressor does not have the capability to update existing model parameters to accommodate new data, so it is initialized and retrained at every timestep with the loss function above.

# 4 Results

In this study, stock parameters were generated from the following distributions:

| Parameter | Distribution |
|---|---|
| Cash | $\mathcal{U}(-10000, 30000)$ |
| Earning Value of Assets | $\mathcal{U}(10000, 30000)$ |
| Latest Quarterly Earnings | $\mathcal{U}(10000, 30000)$ |
| Price | $\mathcal{U}(10, 200)$ |
| Stock Quality of Leadership | $\mathcal{U}(0, 1)$ |
| Stock Volatility | $\mathcal{U}(0.0, 0.1)$ |

Simulations with two different model configurations were used. These consisted of the following:

**Model Configuration 1**

| Model | Count |
|---|---|
| Multi-Layer Perceptron | 20 |
| Price-Masked Multi-Layer Perceptron | 20 |
| Random Forest Regressor | 20 |
| Passive Aggressive Regressor | 20 |

**Model Configuration 2**

| Model | Count |
|---|---|
| Multi-Layer Perceptron | 40 |
| Price-Masked Multi-Layer Perceptron | 0 |
| Random Forest Regressor | 20 |
| Passive Aggressive Regressor | 20 |

Each simulation was defined with the following parameters:

| Parameter | Value |
|---|---|
| Number of Trials | 25 |
| Number of Timesteps | 365 |
| Number of Stocks | 100 |

Note that simulations with exploding stock valuations were dropped. These cases occurred when regressors learned to value stocks with runaway valuations; this is a phenomenon that needs to be studied in the simulator. The vast majority of trials do not exhibit this behavior.

## 4.1 Model Performance Comparison

Model performance was characterized in this study as the mean portfolio percent change from the beginning to the end of each trial separated by model type, subtracted by the mean portfolio percent change from the beginning to the end of each trial for all models. Using Model Configuration 1, 25 trials were performed, and 24 returned valid results.

Model Performances (Configuration 1; N=24)

| **Model** | **Performance (Relative %-change)** |
|---|---|
| Multi-Layer Perceptron | -5.1 |
| Price-Masked Multi-Layer Perceptron | 19.7 |
| Random Forest Regressor | -5.9 |
| Passive Aggressive Regressor | -8.7 |

The Price-Masked Multi-Layer Perceptron significantly outperforms the other agents. Since it's not influenced by the current price, it has the stubbornness to influence valuations more than the other agents, which appears to self-fulfill the success of its decisions over the long term. The stubbornness of agents can have a significant effect on overall environmental patterns, so this result is intuitive [ea23a]. Furthermore, the results from Model Configuration 2 support this theory:

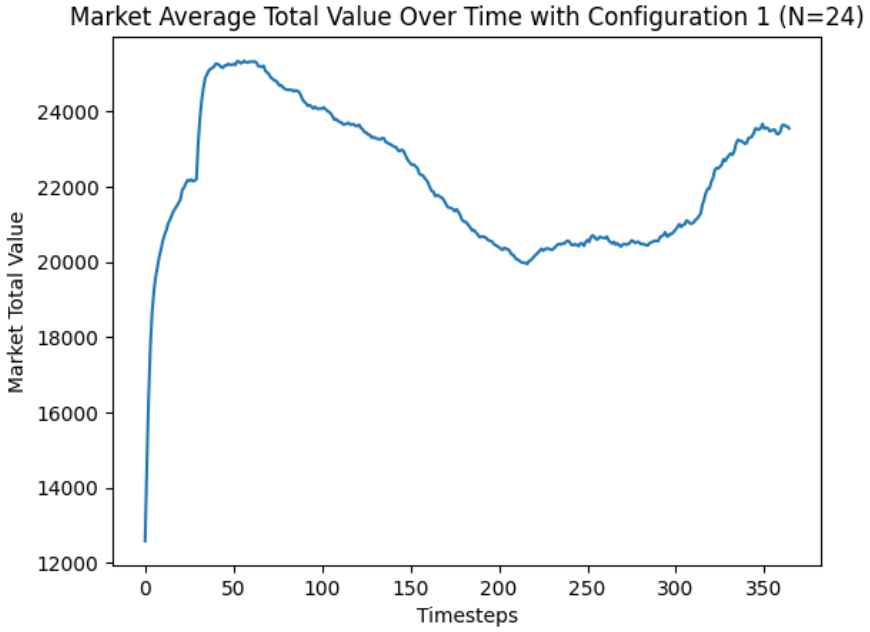
Model Performances (Configuration 2; N=20)

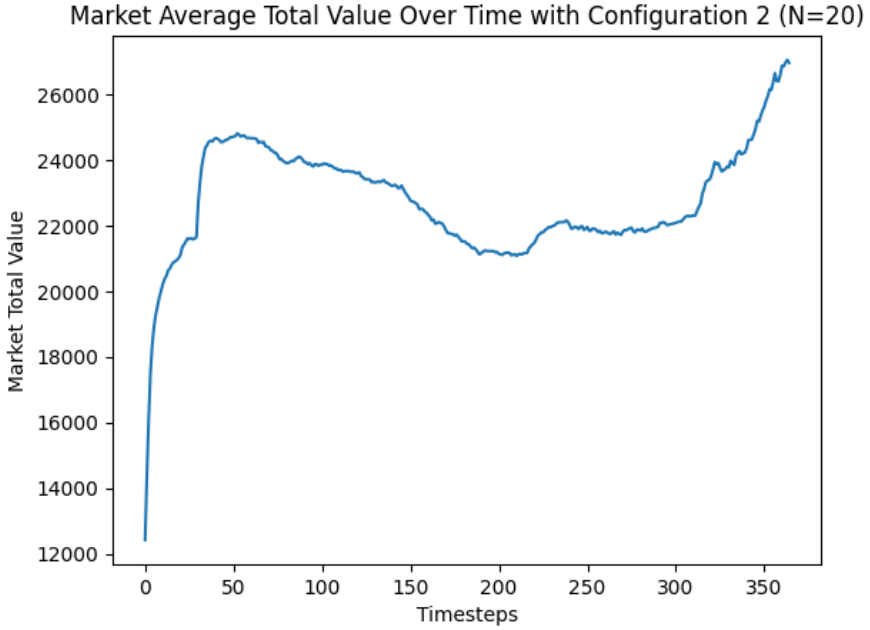
| **Model** | **Performance (Relative %-change)** |
|---|---|
| Multi-Layer Perceptron | -3.8 |
| Random Forest Regressor | 1.8 |
| Passive Aggressive Regressor | 5.8 |

The Multi-Layer Perceptron valuations are likely aligned more closely with the Price-Masked Multi-Layer Perceptron, leading to more success than the other regressors. Once the Price-Masked Perceptron is removed, its performance significantly drops compared to the other regressors in the Configuration 1 trials. This effect is inverted for the Passive Aggressive Regressor; it's possible that this becomes the most stubborn regressor by leaning more Passive than Aggressive in its selection of $C$ in its model definition. Finally, the Random Forest Regressor has median performance in both configurations. This is likely due to its adaptability, as it is completely retrained at every timestep. This model will not influence market valuations, but it instantly reacts to changes in valuation trends. These results suggest that in this environment, the feature selection, policy, and policy update conditions likely have more of an effect on agent performance than the Machine Learning and Deep Learning regression models that determine stock valuations.

## 4.2 Market Performance by Configuration

Below is the average total market valuation by model configuration:




Both configurations had similar shapes: the beginnings of the trials consisted of a sharp valuation increase, likely due to undervaluations of initial stocks relative to the training data, followed by a market correction as models began to learn and recalibrate their valuation expectations for the new market, and ended with price increases in healthier stocks as they became more apparent versus struggling stocks.

It's worth noting that the concluding valuation increase was sharper in Configuration 2 than Configuration 1. Additionally, of the 25 trials with Configuration 2, 5 of them had exploding valuations, versus only 1 with Configuration 1. It could be found that without the Price-Naive Perceptron participants, the market valuations were more unstable. This shows that Configuration 2 may be more prone to market bubbles than Configuration 1. This is intuitive, as "most economists would define a bubble as a situation where an asset's price exceeds the "fundamental" value of the asset", and the

Price-Naive Perceptron values stocks without considering their current nominal market price, which may have a role in preventing runaway valuation growth in this environment [Jan25].

### 4.3 Random Forest Regressor Feature Importance Study

The Random Forest Regressor's feature importances at the end of each non-explosive Configuration 1 trial were averaged into the following results:

| **Feature** | **Importance (%)** |
|---|---|
| Price | 52.5 |
| Cash | 17.7 |
| Earning Value of Assets | 6.4 |
| Latest Quarterly Earnings | 1.9 |
| 1-Day Percent Change | 1.9 |
| 5-Day Percent Change | 1.5 |
| 10-Day Percent Change | 1.7 |
| 1-Month Percent Change | 1.9 |
| 3-Month Percent Change | 2.9 |
| 6-Month Percent Change | 4.7 |
| 1-Year Percent Change | 2.2 |
| 3-Year Percent Change | 2.5 |
| 5-Year Percent Change | 1.6 |

Current price is by far the most significant contributor in valuation, which is intuitive. Cash is the second-most important feature, which could potentially be explained by the fact that cash is required for continued growth, and that a very negative cash is the primary reason that a stock would be delisted and lose all value. The Random Forests also assign more value to 3-Month and 6-Month percent change than the other time periods, showing that these models learned to focus on medium-term trends more than short-term ones.

## 5 Conclusion

The framework provided by this stock market environment for inter-agent stock trading allows for a mostly-stable environment with suitable trading volume that allows for model and policy evaluation. The most significant takeaway was the effect of stubbornness by the participants using the Price-Masked Perceptron: since these participants were not affected by the market's nominal valuation of stocks, they had significant pricing power over the other participants. They also kept the market more stable by sticking closely to fundamentals, rather than letting current market valuations of stocks affect their own valuations. Finally, current price, followed by current cash and medium-term stock growth, are the most important features in this environment for valuations by the Random Forest Regressors.

### 5.1 Next Steps

Exploring different loss functions for the Perceptron would be an interesting study, as different regression loss functions may have different advantages. Mean-Squared Error loss was used to train the Perceptrons in this study, but this loss function weights higher-priced stocks more than lower-priced ones by nature because MSE loss is scale-dependent [ea23b]. A Mean Absolute Percent Error (MAPE)

loss may be more appropriate, especially in a stock market environment where Nominal Rate of Return is a main measure of performance, which consists of calculating percent change of a portfolio [Cor]. This loss would more heavily weight smaller-valued stocks, but produces unstable values for stocks with valuations close to 0 [Kim15].

Additionally, simulating with more market participants could lead to different results. The Price-Masked Perceptrons could have less pricing power if there were more price-observant participants to balance them. Also, changing the number of stocks in the market and the number of stocks sampled per timestep by each participant would be a worthy experiment. However, increasing the scale of the simulation would be computationally expensive; running the 50 simulations observed in this study took about 17 hours on average hardware (Intel Core i7-8565U CPU, NVIDIA GeForce MX150 GPU). Continuing to use simple Perceptrons will benefit runtime efficiency [ea24b]. Additionally, a single optimizer could be used to update all Perceptrons' parameters in one single backpropagation, which would also improve speed [AI25]. However, this would not apply to the non-Neural Network regressors.

Finally, making use of a generative model to create stocks and influence their parameters over time could be more life-like than the random sampling techniques that are currently being used. A variety of stock market datasets exist that could be used to train models to generate more realistic market landscapes [Sto]. Similar to how Generative AI can be used to improve the realism of gaming, many Reinforcement Learning environments could benefit from this improved realism as well, including this stock market simulator [GM23]. It has been shown that these models, such as Variational Autoencoders and Generative Adversarial Networks, are able to learn and regenerate real data distributions, which would remove the need for explicitly defining the contrived, simple random distributions used for this study [dAea25].

# References


[AI25] Lightning AI. Multiple models and optimizers. *PyTorch Lightning Python Library Documentation*, 2025.

[Cor] Corvin. How to calculate the nominal rate of return. *The Trading Analyst*.

[dAea25] de Albuquerque et al. Generative ai applied for synthetic data in pmu. *Energy Reports*, 2025.

[ea09] Popescu et al. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 2009.

[ea13] Minh et al. Playing atari with deep reinforcement learning. *DeepMind Technologies*, 2013.

[ea15] Salas et. al. A variational bayesian state-space approach to online passive-aggressive regression. 2015.

[ea18] Zhou et. al. Reinforcement learning for stock transactions. 2018.

[ea20a] Fan et. al. A theoretical analysis of deep q-learning. *Learning for Dynamics and Control*, 2020.

[ea20b] Wu et al. Incremental learning via rate reduction. 2020.

[ea22] El Mrabet et al. Random forest regressor-based approach for detecting fault location and duration in power systems. *Cybersecurity and Privacy-Preserving in Modern Smart Grid*, 2022.

[ea23a] Kareeva et al. Influence in social networks with stubborn agents: From competition to bargaining. *Applied Mathematics and Computation*, 2023.

[ea23b] Terven et al. A comprehensive survey of loss functions and metrics in deep learning. *Springer Artifical Intelligence Review*, 2023.

[ea24a] Dakalbab et al. Artificial intelligence techniques in financial trading: A systematic literature review. *Journal of King Saud University - Computer and Information Sciences*, 2024.

[ea24b] Liu et al. Scaling up multi-agent reinforcement learning: An extensive survey on scalability issues. *IEEE*, 2024.

[ea25] Zhang et al. Generalized linear markov decision process. 2025.

[Fer25] Fernando. Bid and ask definition, how prices are determined, and example. *Investopedia*, 2025.

[GM23] Gozalo-Brizuela Garrido-Merchan. A survey of generative ai applications. 2023.

[Hay23] Hayes. How interest rates help promote saving and investing. *Investopedia*, 2023.

[Isk24] Ayyildiz Iskenderoglu. How effective is machine learning in stock market predictions? *Heliyon*, 2024.

[Jan25] Baumann Janischewski. What are asset price bubbles? a survey on definitions of financial bubbles. 2025.

[Kim15] Kim Kim. A new metric of absolute percentage error for intermittent demand forecasts. *Internaltional Journal of Forecasting*, 2015.

[Mil21] Millea. Deep reinforcement learning for trading - a critical survey. *Imperial College London*, 2021.

[oB25] Raymond A. Mason School of Business. Artificial intelligence in the stock market. *Raymond A. Mason School of Business Online Business Blog*, 2025.

[Pom] Pomerantz. Stock market environment github repository. Located at https://github.com/jaredpomerantz/stockmarketsim/tree/main.

[SL25a] Scikit-Learn. Passiveaggressiveregressor. 2025.

[SL25b] Scikit-Learn. Randomforestregressor. 2025.

[Sto] Explore the top stock market datasets for ai and machine learning. *iMerit*.

[Sto21] What is the lowest a stock can go? *Financhill Blog*, 2021.

[Vipwn] Vipond. Risk-free rate. *Corporate Finance Institute*, Unknown.

[Xu25] Liu Xu. Optimizing lightweight neural networks for efficient mobile edge computing. *Scientific Reports*, 2025.