Goldman Sachs Stock Price Analysis with Deep Learning

– Yuanshan Zhang, Mengxin Zhao, Jinke Han

Introduction

We're delving into the exciting world of Goldman Sachs stock price analysis with deep learning—a fusion of financial expertise and cutting-edge technology.

At the heart of our project lies the quest to predict Goldman Sachs's stock prices, employing the prowess of both deep learning and traditional models. More than just prediction, our aim is to craft a daily trading algorithm, one that makes shrewd decisions to maximize profits.

Data Description

Our dataset encompasses the historical stock prices of Goldman Sachs Corporation, extending from May 4, 1999, to April 5, 2024. It includes precisely 6,272 entries, representing daily market activity over nearly 25 years. The dataset was meticulously chosen for its depth. Each entry contains detailed financial metrics: Open, High, Low, Close, Adjusted Close prices, and Volume. Such a comprehensive dataset enables in-depth time series analysis, critical for understanding price fluctuations and market behavior. Sourced from Yahoo Finance(https://pypi.org/project/yfinance/) through the `yfinance` Python package. (Colab link: https://colab.research.google.com/drive/1BJNOuv-MUjDbK43m16blYwzLY88ceUqA?usp=sharing#scrollTo=dqe9KSSDZjyr) . Enriching this financial tapestry, we wove in 6 additional columns with the help of the `ta` library, integrating key stock indicators like RSI into our fabric.

Traditional Model

XGBoost with random search:
Result shows that the model suffers from overfitting:The train MSE and MAE are extremely high while the tests are extremely low. The test R-squared is -0.568, indicating that the model performs worse than using the average as prediction. Potential reason could be: XGBoost cannot take time and order for data into consideration, it is not good at predicting time series data.
Polynomial Model:
The model performs quite well with train MAE of 2.53 and test MAE of 4.75.

LSTM

Basic Model: LSTM(units=100) + BatchNormalization + LSTM(units=64, with he_normal) + BatchNormalization + Dense(units=1, with he_normal), with optimizer of Adam, using MSE as loss function and MAE as evaluating metrics. The model performs well, with test MAE of 15.05.
With LeakyReLU: Basic Model with two LeakyReLU(alpha=0.1) after two BatchNormalization. To our surprise, the model performs badly, with RMSE equaling 127.09 and test MAE equaling 103.02. Potential reason could be: vanishing gradients.

Hyperparameter Tuning: Using grid search on the basic model with learning_rate of [0.01, 0.001, 0.0001] and batch_size of [32, 64, 128]. The best model is from sweep 2: Learning rate =0.001, Batch size=32, with the smallest validation MAE of 3.869 and test MAE of 29.914, R-square of 0.63.

Transfer Learning

Then we use the application of a pre-trained model to our Goldman Sachs (GS) dataset as part of our transfer learning strategy. We obtained the pre-trained model from Kaggle(https://www.kaggle.com/code/shiratorizawa/nyse-stock-price-prediction-and-transfer-learning), which was initially trained on Google stock price data. The model consists of six layers. To begin with, we froze the top two layers to retain the learned features, while allowing the weights in the remaining four layers to be updated.

Furthermore, we implemented an early stopping mechanism to monitor the validation loss during training. This function is set with a threshold of 100 epochs and a patience level of 10. Thus, if the validation loss does not improve after 10 consecutive epochs, the training will automatically stop. This approach helps in preventing overfitting and ensures that the model generalizes well on new, unseen data. Finally, it restored the model weights from the end of the best epoch: 6.

The MSE value is 0.0282, which means that the prediction error of the model is relatively small. The MAE has a value of 0.1128, which indicates that, on average, the difference between the model's predicted value and the actual value is about 0.113. This error is acceptable.
The R^2 is 0.2426, which indicates that only about 24.26% of the stock price movement is explained by the model. This means that nearly 75% of the change is not captured by the model.

Although models can predict stock prices to some extent, their predictive power is limited. Models capture only a small fraction of the changes, leaving a large number of fluctuations unexplained. This may be because stock market prices are affected by a variety of factors, including market sentiment, economic indicators, political events, etc., which may not be adequately accounted for in current models.

Where actual prices have fluctuated wildly, the model's predictions are less accurate, which may be the main reason for the low R^2. Models need to be improved to better predict these fluctuations, especially when trying to capture the nonlinear behavior of the stock market.

DQN (Deep Q Learning)

Deep Q learning is a combination of model-free reinforcement learning and deep neural network. DQN is suitable for tasks with large or continuous state or action space or when environment dynamics are indeterministic (reason for being a model-free algorithm). Therefore, using the action value function Q(s,a) enables the algorithm to learn the best policy directly from the action without the necessity of the full knowledge of the environment dynamics.

DQN is therefore suitable in trading scenarios since the state space is large (the stock market is changing every second) and the full knowledge of the environment dynamics is impossible to gain (no one knows what tommorrows' stock market will look like).

The building of our DQN model consists of 4 major parts: 1. Environment 2. Model (MLP & LSTM) 3. Replay Memory 4. Agent

Please see the Appendix for detailed implementation of the parts and the experiment results.

Conclusion

The traditional model of polynomial regression can outperform other traditional models, indicating polynomial regression is complex enough to capture all the variations. The XGBoost would lead to overfitting. For LSTM, we can try more combinations of layers to capture enough variation in the future. The reinforcement learning, on the other hand, took into account complex trading logics and disordered the time information in the ReplayMemory, thereby offering more robust trading strategies. However, huge amounts of training episodes are required for convergence, and like traditional and deep learning methods, stock prices information and technical indices alone are not enough for capturing the complexity of the stock market.

The transfer learning application from Google's stock model to Goldman Sachs has shown a low prediction error with MSE at 0.0282 and an acceptable average error (MAE) of 0.1128, yet it explains only 24.26% of stock price movements (R^2), indicating limited predictive power and substantial unexplained variability. Moreover, the model's limited capability in capturing stock price movements might be exacerbated by its failure to integrate diverse and unpredictable factors such as market sentiment, economic indicators, and political events, which are crucial for accurate stock price prediction.

Future Work

DQN

1. Run more episodes and observe the convergence
2. Apply sequential modeling such as LSTM
3. OHLC and technical indexes alone are not sufficient to capture the complexity and volatility of stock trading. For instance, sentiment analysis of news can also be incorporated into the state information. As a result, more data sources should be incorporated.
4. Combine multiple stocks of the same genre to make the model more robust.
5. Design more realistic trading environment and more comprehensive rewarding mechanism
6. Fine-tune the hyperparameters.

LSTM

1. It is difficult to select the best model for LSTM, since for some sweeps the test scores are better than the validation scores.

Appendix

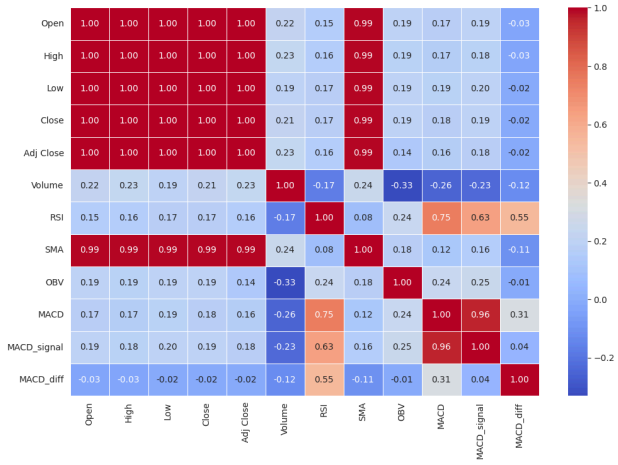
Contribution

Name	Contribution	Description
Mengxin Zhao	30%	Finish all traditional models and LSTM models, utilize wandb to log all experiments in. Compare all models and find the best model.
Jinke Han	30%	Do the transfer learning part. Model Adaptation. Parameter Tuning and Model Enhancement. Early Stopping Implementation
Yuanshan Zhang	40%	1.Calculate the technical indexes 2. Build the trading environment using OpenAI gym 3. Build the MLP & LSTM for RL 4. Build the Agent/ReplayMemory/Trainer 5. Train the model

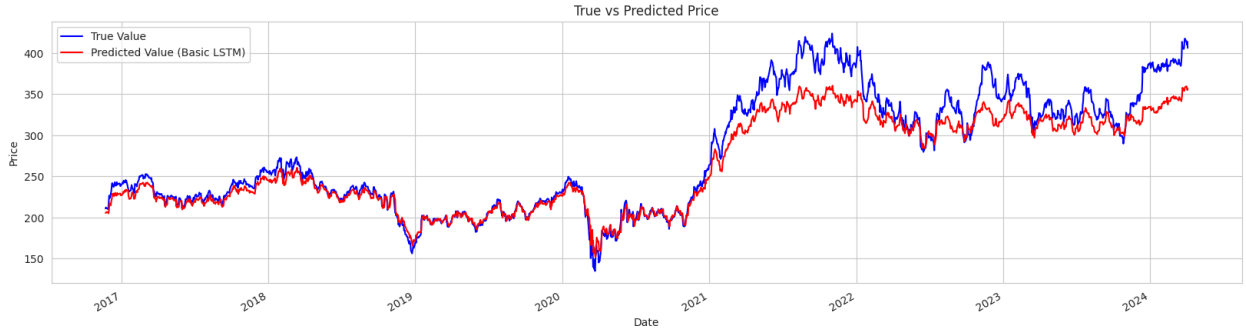
References:

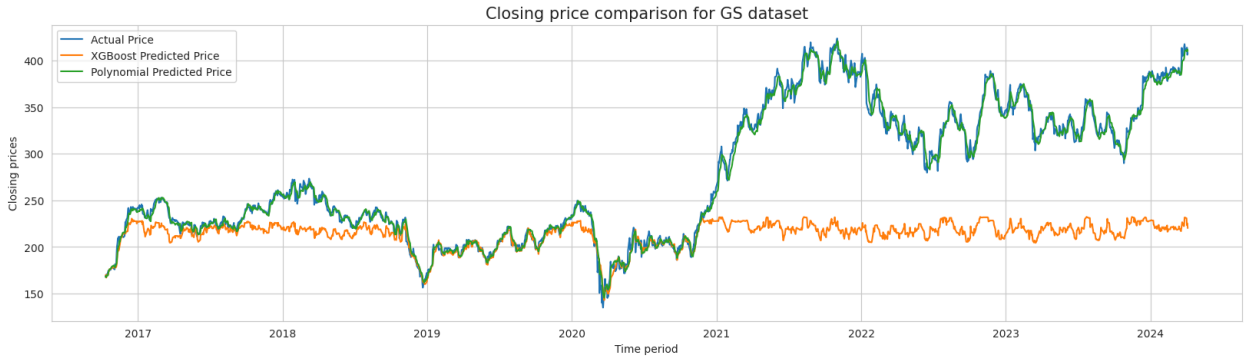
1.https://www.kaggle.com/code/shiratorizawa/nyse-stock-price-prediction-and-transfer-learning
2. Raschka, S., & Mirjalili, V. (2019). Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow 2. Packt.
3. https://github.com/AminHP/gym-anytrading
4. We also use ai to debug and help us to find high level techniques to solve problems.

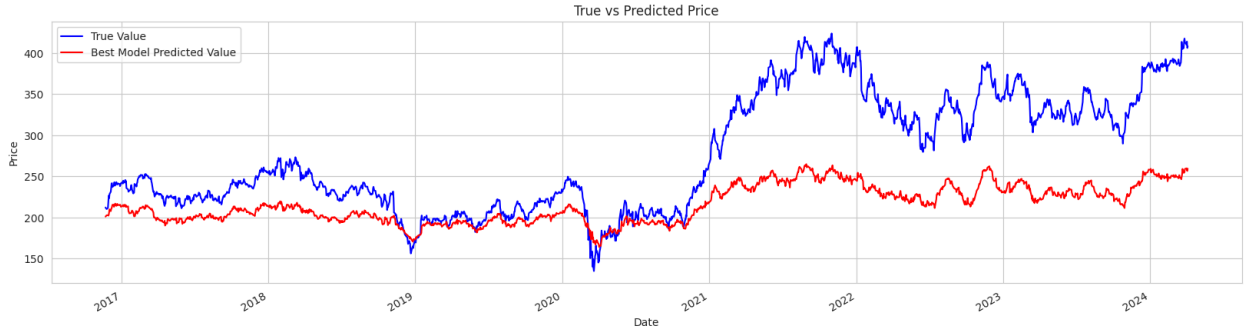
1. Data Description

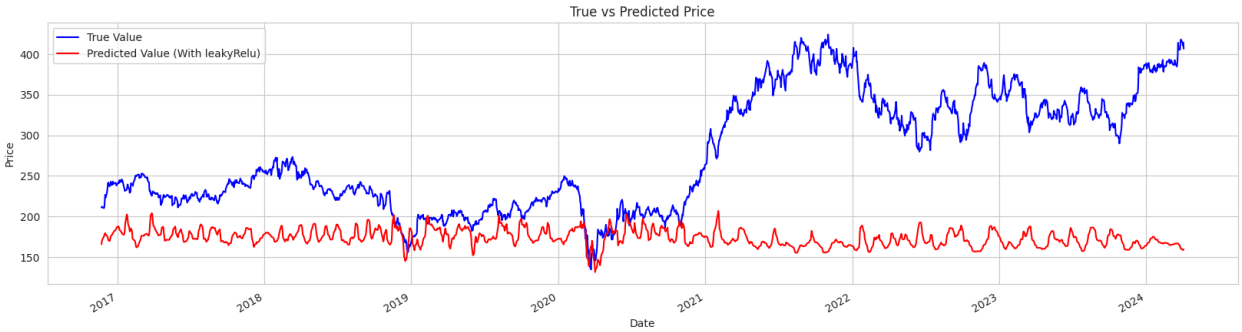


2. Traditional Model & LSTM







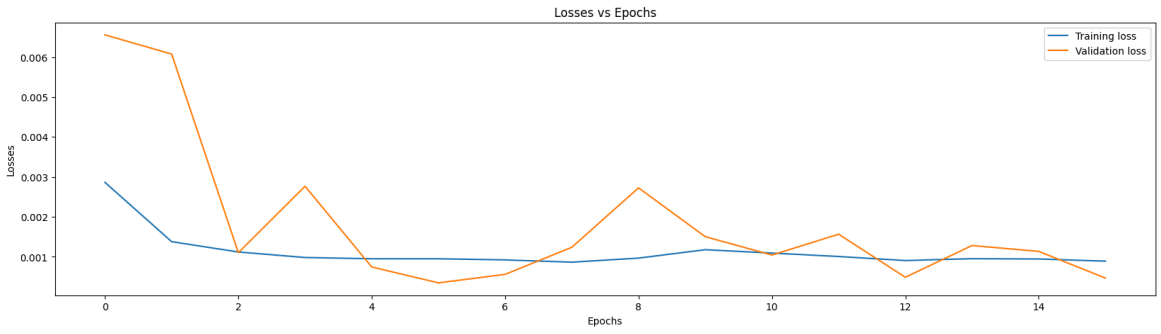


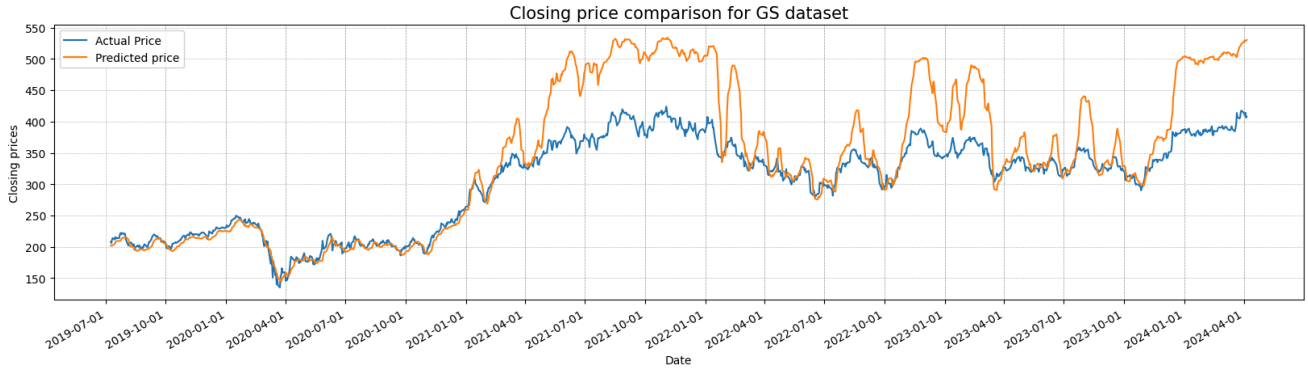
Link of wandb report, including test scores for LSTM model:
https://api.wandb.ai/links/mxzhao0108/9fww2dq1

3. Transfer Learning Part:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_4 (Sequential)   (None, 300, 256)          135168

 sequential_5 (Sequential)   (None, 300, 128)          164864

 bidirectional_5 (Bidirecti  (None, 128)               98816
 onal)

 batch_normalization_5 (Bat  (None, 128)               512
 chNormalization)

 dropout_5 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 1)                 129

=================================================================
Total params: 399489 (1.52 MB)
Trainable params: 99201 (387.50 KB)
Non-trainable params: 300288 (1.15 MB)
_________________________________________________________________
```

Transfer learning pre-trained model





4. DQN

Definition

State: stock price history shown to the agent at the current time step, which includes (Open, High, Low, Ajd Close, Close, Volume, price_diff, and technical indices RSI, SMA, OBV, MACD, MACD_signal, MACD_diff), total profit, available balance, and position value
Action: hold, buy, and sell
Reward: no reward for hold and buy. Reward is only given after selling, and equal to the amount of profits from selling. However, a negative reward is assigned to selling with no position.
Episode: an episode starts at (window_size+1) and ends when reaches the end of the data/exceeds the loss limit
Agent: the algorithm that can be trained to choose the optimal actions at each state to maximize the total rewards

1.Environment:

First, an environment is built using the OpenAI gym to simulate stock trading. The environment consists of 3 crucial methods: 1. reset() 2. get_observation() 3. step() 4. render()

– the reset method restores the environment to the initial state at the beginning of a new episode.
– the get_observation method concatenates the state information.
– the step method assigns rewards and returns 3 components of the transition quintuple: next_state, reward, and done(if an episode is done). The render method is designed to render real-time information to the screen.

A testing dataset is used to prove the functionality of the environment and the window size is set to 3:

Date	Close	RSI	SMA	MACD
1999-05-04	70.3750	4.079308	-1.416558	-0.049732
1999-05-05	69.1250	-4.381850	-1.430845	-0.082475
1999-05-06	67.9375	-4.381850	-1.444655	-0.138861
1999-05-07	74.1250	1.874544	-1.416201	-0.019945
1999-05-10	70.6875	-0.044335	-1.414844	-0.017160
1999-05-11	70.6250	-0.070228	-1.414177	-0.016986
1999-05-12	73.5000	0.876825	-1.404312	0.058086
1999-05-13	73.1875	0.739990	-1.397807	0.107489
1999-05-14	70.1875	-0.345776	-1.400367	0.065820
1999-05-17	68.6250	-0.774681	-1.405986	-0.009072
1999-05-18	68.6250	-0.774681	-1.410584	-0.068212
1999-05-19	69.3125	-0.524998	-1.413106	-0.096327
1999-05-20	68.3750	-0.795880	-1.416915	-0.142381
1999-05-21	67.5000	-1.032335	-1.420011	-0.200327
1999-05-24	65.3125	-1.537319	-1.425011	-0.301311

The initial state returns the price information of the first 3 days, total profit, available balance, and position value:

```
array([ 7.03750000e+01,  4.07930768e+00, -1.41655812e+00, -4.97324031e-02,
        6.91250000e+01, -4.38185033e+00, -1.43084471e+00, -8.24753696e-02,
        6.79375000e+01, -4.38185033e+00, -1.44465508e+00, -1.38861370e-01,
        0.00000000e+00,  5.00000000e+04,  0.00000000e+00])
```

The first step begins at step4, and the trade price at the current time step is the latest ‘Close’ in the current state (the current state of step4 is the initial state):

```
Step:4
Next State: [ 6.91250000e+01 -4.38185033e+00 -1.43084471e+00 -8.24753696e-02
  6.79375000e+01 -4.38185033e+00 -1.44465508e+00 -1.38861370e-01
  7.41250000e+01  1.87454438e+00 -1.41620096e+00 -1.99450368e-02
  0.00000000e+00  5.00000000e+04  0.00000000e+00]
Action:2, Reward:-10, Done:False, Info:{'Trade price': 67.9375, 'Total profit': 0, 'Available balance': 50000, 'Position value': 0}
```

The episode ends at step15 because the the testing dataset has 15 samples:

```
Step:15
Next State: [ 6.83750000e+01 -7.95879607e-01 -1.41691529e+00 -1.42381048e-01
  6.75000000e+01 -1.03233502e+00 -1.42001071e+00 -2.00327367e-01
  6.53125000e+01 -1.53731933e+00 -1.42501102e+00 -3.01310914e-01
  1.62500000e+00  5.00016250e+04  0.00000000e+00]
Action:0, Reward:0, Done:True, Info:{'Trade price': 67.5, 'Total profit': 1.625, 'Available balance': 50001.625, 'Position value': 0}
```

The testing result shows that the environment is working as our design expected. However, this is a rather simple environment built upon simple rules. In reality, the design should be much more complex.

2.Model

An MLP with 3 layers and an LSTM is used to estimate the Q function by using the current state as input. However, like mentioned above, since the environment dynamics is indeterministic, the ground truth of Q function can only be approximated. In reinforcement learning, approximating

the target is called ‘bootstrapping’. The target Q is bootstrapped by using the same network but with the next state as input and less frequent network parameters update.

3.Replay Memory[1]

Since we are now approximating the Q with a MLP model, updatings the weights for a state-action pair will affect the output of other states as well. When training NNs using stochastic gradient descent for a supervised task (for example, a classification task), we use multiple epochs to iterate through the training data multiple times until it converges. This is not feasible in Q-learning, since the episodes will change during the training and as a result, some states that were visited in the early stages of training will become less likely to be visited later.

Furthermore, another problem is that when we train an NN, we assume that the training examples are IID (independent and identically distributed). However, the samples taken from an episode of the agent are not IID, as they obviously form a sequence of transitions.

To solve these issues, as the agent interacts with the environment and generates a transition quintuple, we store a large (but finite) number of such transitions in a memory buffer, often called replay memory. After each new interaction (that is, the agent selects an action and executes it in the environment), the resulting new transition quintuple is appended to the memory.

To keep the size of the memory bounded, the oldest transition will be removed from the memory . Then, a mini-batch of examples is randomly selected from the memory buffer, which will be used for computing the loss and updating the network parameters.

4.Agent

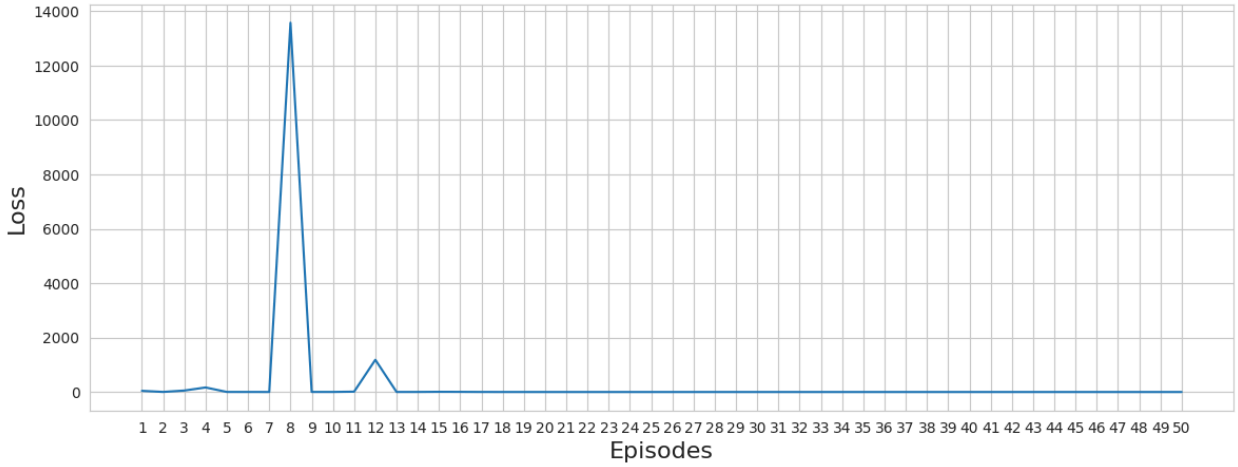
Finally, an agent that can be trained to choose the optimal actions at each state to maximize the total rewards is built. Specifically, the agent performs 2 major functionality: 1. It will push the latest transition quintuple to the memory during its training 2. It will calculate the Q values at the current state and pick the action with the highest value. Moreover, to let the agent fully explore different strategies during early stages of episodes, the epsilon-greedy policy is applied to give small chances to non-optimal actions to be picked by the agent, and the chances decrease with time.

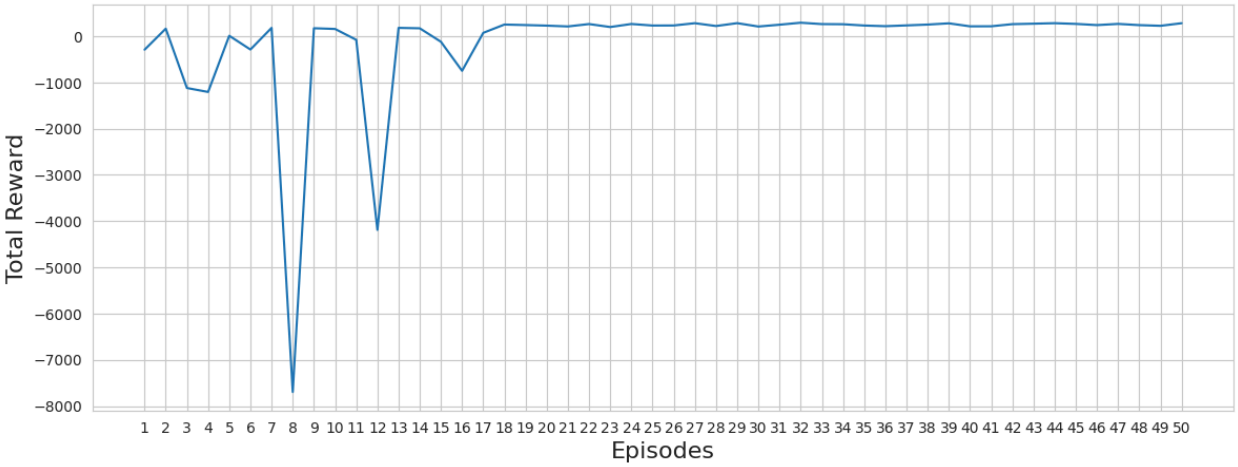
Experiment Results of MLP

Hyperparameters include window_size, learning rate, gamma(decay factor for reward), replay_memory_size, batch_size, episodes, epsilon_decay. Tuning these hyperparameters are crucial to enhance the model performance. However, designing a good reward mechanism and

[1] Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning: Machine Learning and Deep Learning with Python, Scikit-learn, and TensorFlow 2*. Packt.

trading logic is even more important. So, most of our time was invested into building, experimenting, and testing the reward mechanism and trading logic within the environment, and eventually, not much time is left for hyperparameter tuning. So, we will focus on discussing how the reward design will influence the model's performance.





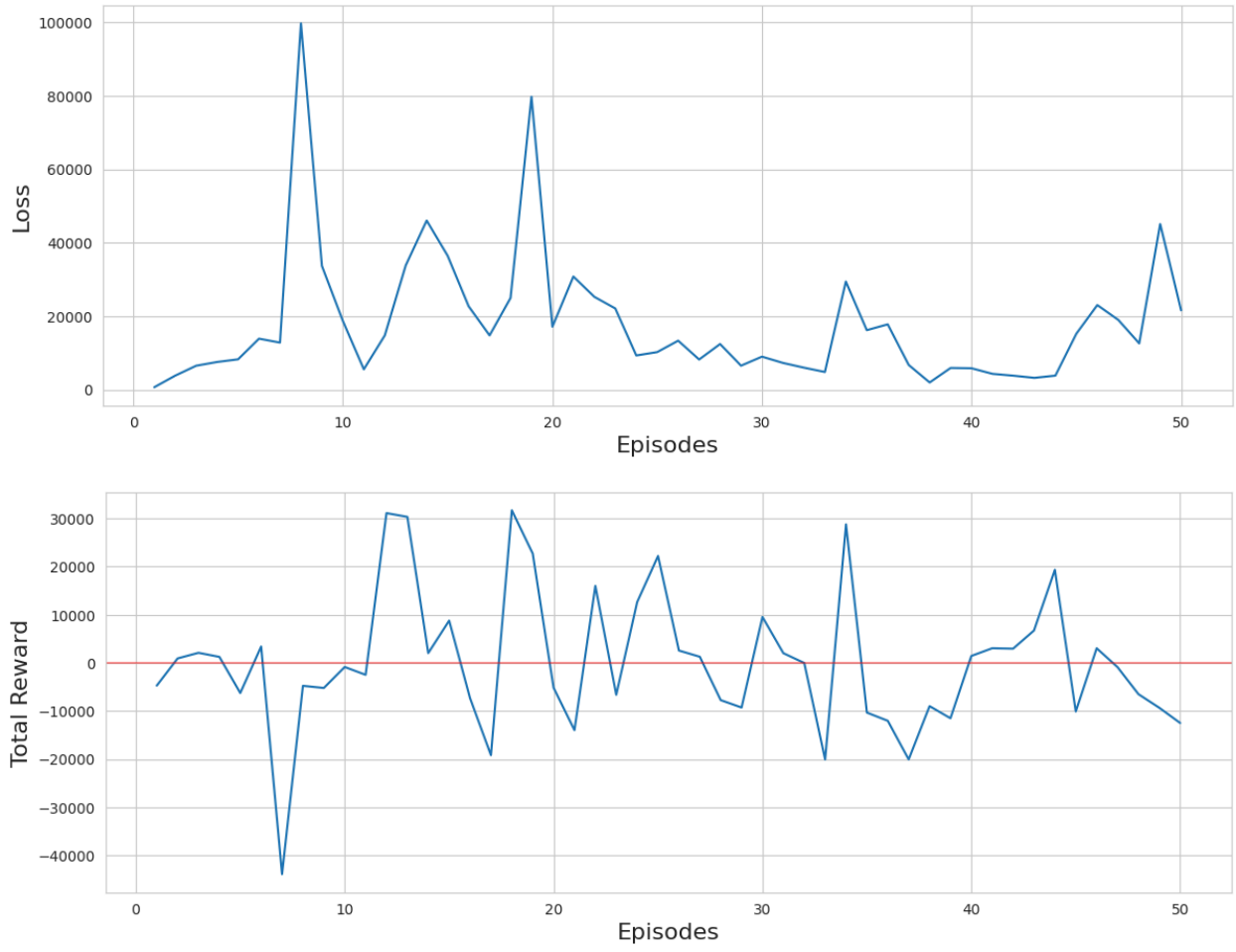
Experiment results for experiment 1 with MLP

In the first experiment, we split train and test data evenly, and implemented more complex reward mechanism for holding as follows:

```
else: # hold
    if price_change > 0 and len(self.positions) > 0:
        reward = 0.5 # positive reward for holding during an up market
    elif price_change < 0 and len(self.positions) > 0:
        reward = -0.5 # negative reward for holding during a down market
    else:
        reward = 0.1 # small reward for holding in stable market or no position
```

From the result, the loss and total reward converge quickly after only about 20 episodes, which, in theory, should not happen so fast. Moreover, during testing, the trading strategy that the agent is taking turned out to be deterministic because the train and test data generates the same testing result - test Loss: 0.0293, Test Reward: 303.6000. This is because instead of buying and selling, the agent learned always choose hold when there is no position and keeps getting the small 0.1 reward, and 303.6 is exactly 0.1*len(data)-window_size.

So, we revised the train and test split to use only 0.3 as the test set, assign 0 reward for holding, and use actual profits as reward for selling. This is because reward should only be given when profit is made, and buying or holding will not directly bring in profits. Therefore, assigning 0 to buying and holding will let the model learn the potential consequences of these two actions in relation to profits. And the experiment result is as follows:

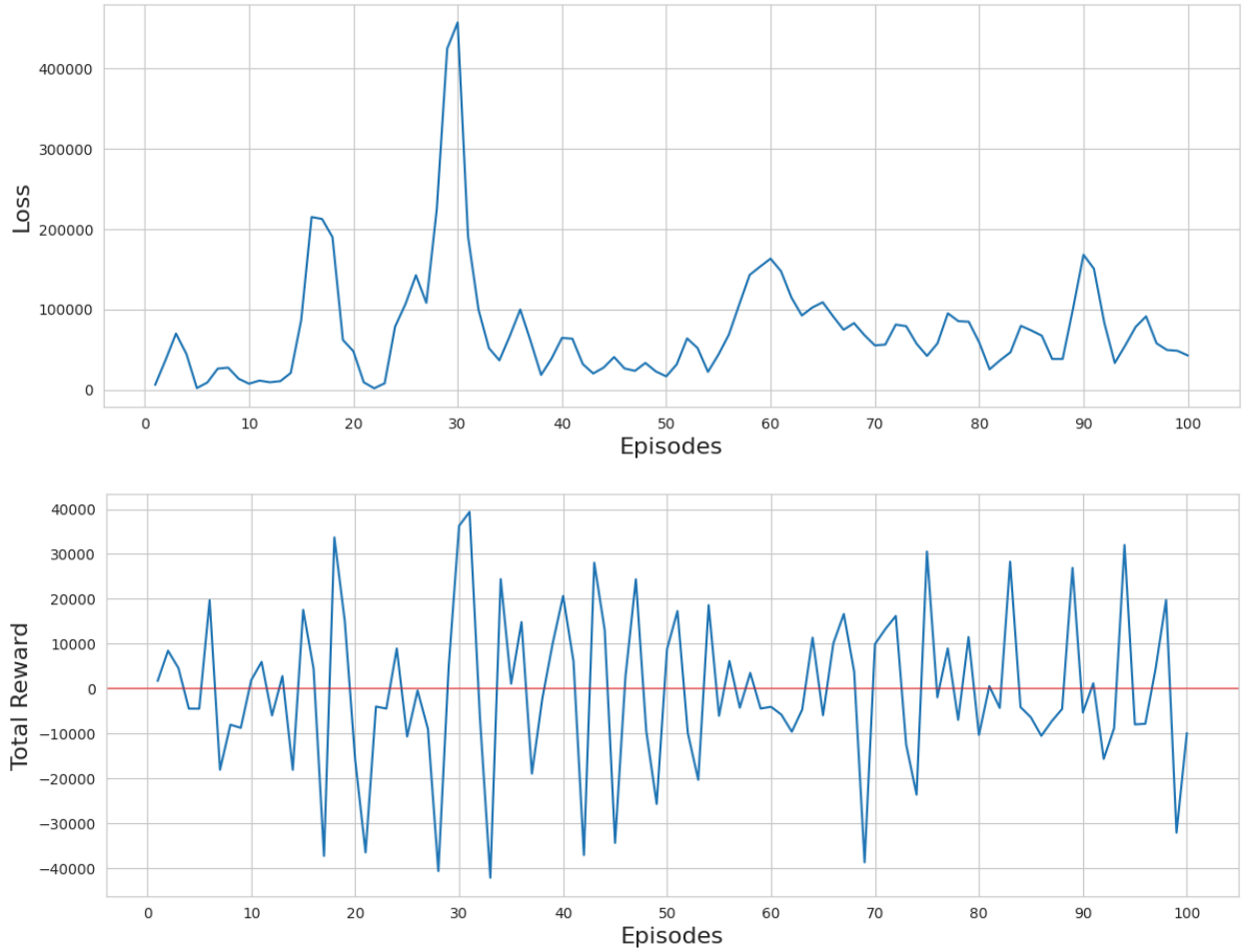


Experiment results for experiment 2 with MLP

Now, we see that the model is not even close to converging by the end of 50 episodes. However, at least we are presenting the model from making deterministic moves, and we got: Test reward: 17977.2100, Total profit: 21267.209991455078. Since the model has yet converged, whether the

model can make profit is purely by chance. So, we may run more episodes (hundreds or thousands) for future experiments.

Experiment Result of LSTM



Experiment results for LSTM

Test reward: 19310.4821, Total profit: 19310.48210144043. With LSTM and 100 episodes, the LSTM is still far from converging. Therefore, like MLP, the test profit is purely by chance.