Deep Reinforcement Learning for Automated Stock Trading

A Project Report
Presented to
Professor Dr. Jahan Ghofraniha

San Jose State University

CMPE 260 Reinforcement Learning

By

Arpitha Gurumurthy, Somya Mishra, Tripura Chandana Gayathri Gorla

December 2021

Acknowledgment

We are deeply indebted to Professor Jahan Ghofraniha for his invaluable guidance in the understanding, preparation and implementation of this project.

Table of Contents

Executive Summary 4

Chapter 1. Background and Introduction 5

Chapter 2. Problem Statement 6

Chapter 3. Motivation 7

Chapter 4. Differentiator and Contributions
- 4.1 Vanilla (Baseline) version of the DQN - (DQN_baseline.h5) 8
- 4.2 Our modified version of the DQN - Part 1 (DQN_v1.h5) 8
- 4.3 Our modified version of the DQN - Part 2 (DQN_v4.h5) 9
- 4.4 Our modified version of the DQN - Part 3 (DQN_v5.h5) 10

Chapter 5. Methodology
- 5.1 Why DQN? 12
- 5.2 Data 12
- 5.3 Parameters used to train the DQN agent 13
- 5.4 Neural network 14
- 5.5 Epsilon Greedy approach 15
- 5.6 Experience replay 15
- 5.7 Defining the action functions 16
- 5.8 DQN methodology used in our study 18

Chapter 6. Implementation & Results
- 6.1 Execution details 20
- 6.2 Streamlit integration 21
- 6.3 Evaluation results of the vanilla version - DQN_baseline.h5 21
- 6.4 Evaluation results of the modified version - DQN_v1.h5 23
- 6.5 Evaluation results of the modified version - DQN_v4.h5 26
- 6.6 Evaluation results of the modified version - DQN_v5.h5 28
- 6.7 Comparison and Results 30

Chapter 7. Conclusion and Future Scope 32

Appendix
- Implementation Repository 33
- References 33

EXECUTIVE SUMMARY

Deep Reinforcement Learning for Automated Stock Trading
By
Arpitha Gurumurthy, Somya Mishra, Tripura Chandana Gayathri Gorla

Stock trading plays an important role in making investment decisions. The goal of this project is to try and understand how DQN works with the daily stock data and the predictions associated with it. DRL(Deep Reinforcement Learning) aims to maximise the results or rewards after considering all the possible actions and in our case, for stock market trading, to maximize the profits after taking into account each action that can be made over a certain period of time. While taking into account all the risks involved, it's a very challenging task to actually develop a model that can give accurate predictions so the investors can rely on these predictions.

We have implemented the DQN algorithm and trained the model on various stock data. We evaluate our models and their performances on the most recent data and the results and our observations are all briefly noted down in the following sections. This project is inspired from Q-Trader which is an implementation of Q-learning algorithm for short-term stock trading applications. While this is not a very good model for long-term applications we also aim to predict the results to be more of short-term trading meaning, over the next few days for instance as all the training was done on daily data for a particular stock. Like any typical Reinforcement Learning algorithm, We have an agent, an action and a reward. The agent is the user or the investor that takes input from the action and reward to make his/her decision. Actions here in this project's scope are to buy, to sell, or to hold a stock based on the input from the agent. Here our rewards are the profit or loss that will be generated after an action is made to buy or sell or hold a stock, for instance, a stock was bought on a particular day and then was sold for another price, the difference is calculated and that is considered reward/penalty. This will in turn be used to predict the next possible action. We have used a streamlit environment to visualize our results.

Chapter 1. Background and Introduction

Profitable stock trading is very crucial for investors and investing companies. It goals to optimize the allocation of capital such that the return on investment is maximum while keeping the risks minimum. However, this is a very complicated and challenging task to analyse as the very structure of the stock market is very complex and comes with a lot of risks. “Bear and Bull” are two terms that one can often hear in the stock market. A bear run is a term that suggests a decline in the market prices over a long time while a bull run refers to its opposite. These are terms used by traders who deal in intraday trading. There can be long-term and short-term trading types of investments in the market. This project’s scope only concentrates on short-term trades. “Intraday Trade” is one such term where an investor buys and sells a financial stock within the same trading day, such that all market positions are closed before the market closes for that day. The automated stock trading aims to automate this process of predicting the raise or fall of a particular stock on that day and thus help the trader or investor to make his decision.

With the advent of Data Science and Machine Learning, various research approaches have been designed to automate this manual process. There have been various approaches that were experimented in order to achieve a balanced and low-risk strategy that can benefit most people. Though there are many models today that are efficient in the task but are not completely reliable and 100% efficient. The reason being, a model should be able to take every risk and factors into consideration before giving the predictions. Supervised machine learning algorithms were first used in this context. The predictions from those algorithms were used by the stock traders to take action and maximize their profits. Some common algorithms for stock trading include Volume Weighted Average Price (VWAP), Time Weighted Average Price (TWAP), and Percent of Value (PoV). Supervised algorithms rely on labeled training data. Labeled data occupies a very small fraction of the entire data available today, given its incredible growth spurt.

Reinforcement Learning in the past few decades has improved a lot in terms of solving various complicated problems. Trading is a continuous task without any end point and is one such application that RL algorithms are proved to be applied efficiently. DRL is a suitable candidate for this project because it doesn’t require a lot of labeled data for training. Using DRL with neural networks would be capable of handling large action spaces as well.

Chapter 2. Problem Statement

Financial trading is one of the most attractive areas in finance. Trading systems development is not an easy task because it requires extensive knowledge in several areas such as quantitative analysis, financial skills, and computer programming. A trading systems expert, as a human, also brings in their own bias when developing the system. Relying on people for making decisions on stock trading can be difficult for investment companies as it is more susceptible to error. Designing a profitable business strategy with minimal risk is very challenging.

Deep Reinforcement techniques have proved to be better than humans in numerous applications. For instance, DeepMind Technologies' AlphaGo uses DRL strategies to play the board game of Go. It outperformed the world champion Lee Sedol in March 2016. This proves that DRL may have greater potential in automated stock trading as well. Hence, this project's study focuses on exploring this idea.

Recent works of DRL in stock trading have been applied to environments with discrete and continuous action spaces. Strategies based on critic only approaches such as Q-learning and DQN, actor only approaches such as Policy Gradient and actor critic approaches such as A2C, PPO and DDPG have been explored.

The objective of this project is to come up with a deep reinforcement learning scheme that automatically learns a stock trading strategy by maximizing investment return. We have implemented a few versions of the DQN (Deep Q-Network) algorithm by modifying the underlying parameters with each run. We have also provided a detailed analysis and comparison of each of these models against specific datasets for different time periods: S&P 500 index (GSPC), Nasdaq composite, Facebook and Apple.

The goal is to be able to understand the working and structure of how each variation of our DQN model performs on our problem statement and datasets.

Chapter 3. Motivation

In the last few years, machine learning and deep learning have been widely used to build prediction and classification models for the financial market. However, most of these machine learning models are not trained to model positions and are only focused on picking high performance stocks.

Automated stock trading is extremely beneficial for investment firms, hedge funds or any individual with an avid interest in trading. Profitable automated stock trading strategy is vital to investment companies and hedge funds. Complex problems like stock trading, if done manually, requires a lot of research and labor. But in this era of Artificial Intelligence, even complex problems can be solved better by machines, thereby reducing manual work and human error. This not only saves many resources but also reduces risk to a great extent if trained properly on appropriate and accurate data.

There have been many approaches to solve the Automated Stock Trading problem in the past, but with the advancement in the technologies, there are new techniques in place to improve the accuracy and thus increase the reliability of a model. It is a good opportunity to experiment and try to improvise the existing problem using Deep Reinforcement Learning algorithms.

Our model will be applied to optimize capital allocation and maximize investment performance, such as expected return. Return maximization can be based on the estimates of potential return and risk.

Our models are trained and evaluated against multiple datasets with daily stock prices of different companies and also top indices like S&P 500. The objective of this project is to help find the most optimal trading strategy in a complex and dynamic stock market and minimize human invention for stock trading in order to avoid any errors or biases by automating the process of agent making trading decisions- buy, hold or sell.

Chapter 4. Differentiator and Contributions

In this section, we highlight the modifications done to the baseline implementation of DQN.

4.1 Vanilla (Baseline) version of the DQN - (DQN_baseline.h5)

The baseline model [3] is trained on the GSPC 2010 to 2015 dataset with 1511 days of trading data. It is trained for 10 epochs.

Below screenshot shows the values of hyperparameters used to train the baseline model:

```
self.gamma = 0.95
self.epsilon = 1.0  # initial exploration rate
self.epsilon_min = 0.01  # minimum exploration rate
self.epsilon_decay = 0.995 # decrease exploration rate as the agent becomes good at trading
```

The underlying neural network is defined as follows:

```
    model = Sequential()
    model.add(Dense(units=64, input_dim=self.state_dim, activation='relu'))
    model.add(Dense(units=32, activation='relu'))
    model.add(Dense(units=8, activation='relu'))
    model.add(Dense(self.action_dim, activation='softmax'))
    model.compile(loss="mse", optimizer=Adam(lr=0.01))
```

Bellman equation and replay memory is implemented as follows:

```
def experience_replay(self):
    # retrieve recent buffer_size long memory
    mini_batch = [self.memory[i] for i in range(len(self.memory) - self.buffer_size + 1, len(self.memory))]

    for state, actions, reward, next_state, done in mini_batch:
        if not done:
            Q_target_value = reward + self.gamma * np.amax(self.model.predict(next_state)[0])
        else:
            Q_target_value = reward
```

4.2 Our modified version of the DQN - Part 1 (DQN_v1.h5)

This model is trained on the GSPC 2010 to 2015 dataset. Below are some of the modifications incorporated including:

- Increased the replay memory size from 100 to 10000 and reduced the buffer size to from 60 to 30.
- Updated the Gamma value from 0.95 to 0.99
- Trained on a smaller window size and hence smaller input dimensions. Window size is set to 5 (earlier was 10) and state dimensions are 8 (earlier was 13).
- Updated the underlying neural network to have the below configurations:

```
    model = Sequential()
    model.add(Dense(units=256, activation="relu", input_shape=(self.state_dim,)))
    model.add(Dense(units=512, activation="relu"))
    model.add(Dense(units=512, activation="relu"))
    model.add(Dense(units=256, activation="relu"))
    model.add(Dense(units=self.action_dim, activation='softmax'))

    model.compile(optimizer=Adam(lr=0.0001),loss=Huber(delta=1.5))
    return model
```

- Updated the experience replay function to randomly sample the mini batches instead of going sequential and also updated the bellman equation, is the guiding principle to design reinforcement learning algorithms:

```
def experience_replay(self):
    # retrieve recent buffer_size long memory
    # mini_batch = [self.memory[i] for i in range(len(self.memory) - self.buffer_size + 1, len(self.memory))]
    mini_batch = random.sample(self.memory, self.buffer_size)
    for state, actions, reward, next_state, done in mini_batch:
        if not done:
            Q_target_value = reward + self.gamma * self.model.predict(next_state)[0][np.argmax(self.model.predict(next_state)[0])]
            # Q_target_value = reward + self.gamma * np.amax(self.model.predict(next_state)[0])
        else:
            Q_target_value = reward
```

4.3 Our modified version of the DQN - Part 2 (DQN_v4.h5)

This variation of DQN is trained on GSPC 2000 to 2017 for 2 episodes. It has around 4500 days worth of trading data.

Below are some of the modifications incorporated:

- Increased the replay memory size from 100 to 10000 and updated the buffer size to from 60 to 64.
- Updated the Gamma value from 0.95 to 0.99
- Updated the Epsilon decay value from 0.995 to 0.95
- Trained on a smaller window size and hence smaller input dimensions. Window size is set to 5 (earlier was 10) and state dimensions are 8 (earlier was 13).
- The underlying neural network is same as the modified DQN part 1

- Also used the average price of the stocks in the inventory function to calculate the maximum profit instead of using minimum.

```
def hold(actions):
    # encourage selling for profit and liquidity
    next_probable_action = np.argsort(actions)[1]
    if next_probable_action == 2 and len(agent.inventory) > 0:
        # max_profit = stock_prices[t] - min(agent.inventory)
        max_profit = stock_prices[t] - mean(agent.inventory)
        if max_profit > 0:
            sell(t)
            actions[next_probable_action] = 1 # reset this action's value to the highest
            return 'Hold', actions
```

4.4 Our modified version of the DQN - Part 3 (DQN_v5.h5)

This variant is trained on the Apple 2016-2021 dataset. Below are some of the modifications incorporated:

- Updated the Gamma value from 0.95 to 0.9
- Updated the Epsilon decay value to 0.85
- Trained on a smaller window size and hence smaller input dimensions. Window size is set to 5 (earlier was 10) and state dimensions are 8 (earlier was 13).
- Modified the original hold function to encourage both sell and buy actions based on the next probable action as predicted by the model.
- Also used the average price of the stocks in the inventory function to calculate the maximum profit instead of using minimum.
- Also updated the number of training epochs of the neural network from 1 to 5.

```
next_probable_action_arr = np.argsort(actions)
next_probable_action = next_probable_action_arr[1]
if next_probable_action == 2 and len(agent.inventory) > 0:
    # max_profit = stock_prices[t] - min(agent.inventory)
    max_profit = stock_prices[t] - average(agent.inventory)
    if max_profit > 0:
        sell(t)
        # actions[next_probable_action] = 1 # reset this action's value to the highest
        return 'Hold', actions
elif next_probable_action == 1 and agent.balance > stock_prices[t]:
    best_buy = stock_prices[t] - average(agent.inventory)
    if best_buy < 0:
        buy(t)
        return 'Hold', actions
```

The results and comparison of each of these models above are discussed in Chapter 6 ‘Implementation and Results’.

Chapter 5. Methodology

5.1 Why DQN?

Incorporating a Q-learning algorithm with value iteration is suitable for use cases that have relatively simplistic environments and a fewer number of states and actions. This in turn results in fewer values to update in the Q-table. Since the Q-table updates happen iteratively, the observable state space increases in size. The time taken to traverse through all the values in the Q-table to update the Q-values also increases. For example, in a video game an agent has a larger environment to explore. Each of these environments is made up of a combination of pixels, and the agent can take multiple actions in each space. This repetitive process of calculating and updating Q-values for each state-action pair in a state space like in a video game becomes a computation burden and intractable due to the amount of resources and time consumed.

In order to address this problem, neural networks are used to estimate the Q-function instead of using value iteration. Neural networks are universal function approximators and hence serve as the best fit to calculate the optimal Q function. In this study, we have used Deep Q Network to implement our stock trader given the complexity of the environment which is the ever fluctuating stock market.

5.2 Data

We are experimenting with multiple datasets to train and test our DQN agent. Below are some of the datasets used in this project:

The S&P 500 index, or Standard & Poor's 500, is a very important index that tracks the performance of the stocks of 500 large-cap companies in the U.S. The ticker symbol for the S&P 500 index is ^GSPC.

The series of letters represents the performance of the 500 stocks listed on the S&P. It is important to note that ^GSPC is a price index and is not tradeable. It only shows the movement of stock prices in the S&P 500 index.

The Nasdaq Composite is a stock market index that includes almost all stocks listed on the Nasdaq stock exchange. Along with the Dow Jones Industrial Average and S&P 500, it is one of the three most-followed stock market indices in the United States.

For training the DQN algorithm we've used the following datasets:

1. GSPC 2010-2015
2. GSPC 2000-2017

For evaluating the trained DQN agent we've used the following datasets:

1. Nasdaq composite: Historical daily data from 2011-2021
2. Apple (AAPL): Historical daily data from 2016-2021
3. Facebook (FB): Historical daily data from 2016-2021
4. GSPC 2018

Below is a screenshot of a sample dataset we have used:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2011-11-17	2,637.3701	2,637.4800	2,576.2200	2,587.9900	2,587.9900	2197320000
1	2011-11-18	2,595.0200	2,595.8401	2,567.1499	2,572.5000	2,572.5000	1755360000
2	2011-11-21	2,535.3401	2,539.8701	2,500.8899	2,523.1399	2,523.1399	2048520000
3	2011-11-22	2,517.6399	2,534.3999	2,499.1899	2,521.2800	2,521.2800	1792060000
4	2011-11-23	2,501.1799	2,503.3799	2,460.0801	2,460.0801	2,460.0801	1707770000
5	2011-11-25	2,453.0300	2,477.0300	2,441.4800	2,441.5100	2,441.5100	691750000
6	2011-11-28	2,509.6299	2,531.3201	2,507.7200	2,527.3401	2,527.3401	1626060000
7	2011-11-29	2,529.1101	2,542.4600	2,508.2700	2,515.5100	2,515.5100	1623550000
8	2011-11-30	2,586.3899	2,620.3401	2,582.4900	2,620.3401	2,620.3401	2440960000
9	2011-12-01	2,615.6699	2,636.0801	2,611.4800	2,626.2000	2,626.2000	1826860000

In stock trading, the high and low refer to the maximum and minimum prices in a given time period.

Open and close are the prices at which a stock began and ended trading in the same period.

Volume is the total amount of trading activity.

Adjusted values factor in corporate actions such as dividends, stock splits, and new share issuance.

Valuable information about a stock's performance can be gleaned from understanding the open, high, low, close of a stock, and as well as it’s trading volume.

5.3 Parameters used to train the DQN agent

Below are the definitions and functions of the parameters used in the implementation of our DQN algorithm for stock trading:

1. **Trading period** - It refers to the number of days during which trading is carried out by the DQN agent during training or evaluation.

2. **Action dimensions** - It refers to the number of actions that the DQN agent can take in every state. Our project supports 3 actions including hold, buy and sell.

3. **Memory** - It refers to the maximum size of the replay memory containing experiences which includes the current state, action, reward and next state.

4. **Buffer size** - It refers to the size of mini batches of replay memory used to train the underlying neural network.

5. **Gamma** - It refers to the discount factor which indicates the amount of importance given to future rewards.

6. **Epsilon** - Epsilon refers to the probability of the agent choosing to explore the environment.

7. **Epsilon decay** - The value of Epsilon decreases as the agent is trained to balance the exploration and exploitation. A common practice is to multiply the Epsilon value with Epsilon decay value as the training progresses.

Some of the important definitions related to stock market used in our project:

1. **Inventory** - Inventory in our study is an array containing the stock prices of all the stocks the agent has bought.

2. **Portfolio** - A portfolio in the stock market refers to the collection of financial investments. We define a portfolio as a collection of agent's current account balance, inventory list, return rates, a list of agent's account balance at every trade and the dates on which the trade (buy, sell or hold) was carried out.

3. **Stock Prices array** - This is an array of closing prices of the stocks on a daily basis. The length of this array is the same as the trading period.

4. **Balance** - This refers to the amount of money the agent has when the trading begins or the initial account balance.

All these values are reset to their default values at the end of every trading episode.

5.4 Neural network

We know that neural networks are universal function approximators. In DQN, they help in optimizing the Q function so the agent can take appropriate actions at each state.
We have experimented with a couple of neural network architectures to find the best performing network for our study.

The neural architecture for the baseline model has 5 dense layers including the input and output layers, uses the Adam optimizer with 0.0001 learning rate, and Huber loss.

```
model = Sequential()
model.add(Dense(units=256, activation="relu", input_shape=(self.state_dim,)))
model.add(Dense(units=512, activation="relu"))
model.add(Dense(units=512, activation="relu"))
model.add(Dense(units=256, activation="relu"))
model.add(Dense(units=self.action_dim, activation='softmax'))
model.compile(optimizer=Adam(lr=0.0001),loss=Huber(delta=1.5))
```

5.5 Epsilon Greedy approach

When the agent takes random actions in the stock market environment, it does not learn very well and hence, the Epsilon-Greedy approach is employed. In this strategy the agent will initially explore the environment in order to gain knowledge and then exploit it. It forces the agent to learn to balance between exploration and exploitation. At the start of each episode, the 'Epsilon' value is set to 1 so it is certain that the agent will explore the environment. This value is then slowly reduced so the likelihood of exploration reduces and the greed for exploitation begins.

In order to implement this in our DQN stock trader, we first generate a value between 0 and 1, if this random value is less than the value of Epsilon, exploration is carried out. Otherwise, the next action is chosen via exploitation which is essentially the model prediction. Below screenshot shows the code for how we implement the same in our project.

```
def act(self, state):
    if not self.is_eval and np.random.rand() <= self.epsilon:
        return random.randrange(self.action_dim)
    options = self.model.predict(state)
    return np.argmax(options[0])
```

5.6 Experience replay

A collection of the agent's experience at each time step is called the Replay memory or the Replay buffer. Each tuple representing the agent's experience will contain the following:

1. Current state of the environment at time 't'
2. The action 'a' taken at the current state
3. The reward given to the agent at time 't+1'
4. Next state of the environment at time 't+1' on performing the action 'a'

```
def remember(self, state, actions, reward, next_state, done):
    self.memory.append((state, actions, reward, next_state, done))
```

A collection of these experiences across the episode for each time step is stored in the replay buffer. These experiences are randomly sampled to train the underlying neural network. The reason for choosing random samples is to break the correlation that exists between consecutive samples. The below code snippet shows the random sampling implementation in our project:

```
def experience_replay(self):

    ##Randomly samples from the replay memory
    mini_batch = random.sample(self.memory, self.buffer_size)
```

5.7 Defining the action functions

There are 3 possible actions that the agent can take in a stock market environment. They are hold, buy and sell.

1. **Basic 'Hold' functionality**:

- The hold function encourages selling stocks for profit.
- The hold state receives an action array containing the probability values for hold, buy and sell respectively (Eg: [0.2, 0.7, 0.3]).
- Next, using the **'argsort'** method of Numpy, we sort the array in the increasing order and return their respective indices. (Eg: For [0.2, 0.7, 0.3], argsort returns [0, 2, 1]).
- We define the **next probable action** as the value in the index position 1 of the resulting array on argsort.
- If the value of the next probable action is 2 and the inventory list of the agent is not empty, we calculate maximum profit at this stage. The maximum profit is defined as the difference between the price of the stock on that particular day and the minimum value that it was bought by the agent.
- If this maximum profit is greater than 0, we sell the stock on the same day.

The below screenshot shows the hold function implementation in our project:

```
def hold(actions):
    # encourage selling for profit and liquidity
    next_probable_action = np.argsort(actions)[1]
    ##Sell if next_probable_action is 2
    if next_probable_action == 2 and len(agent.inventory) > 0:
        max_profit = stock_prices[t] - min(agent.inventory)
        if max_profit > 0:
            sell(t)
            # reset this action's value to the highest
            actions[next_probable_action] = 1
            return 'Hold', actions
```

2. **Basic 'Buy' functionality**:

- The 'Buy' function allows the agent to buy a stock on date 't' which is the input to this function.
- If the agent's portfolio balance is greater than the price of the stock on that particular day, we encourage the agent to buy the stock.
- We simultaneously decrease the portfolio balance by the price bought and append the stock to the agent's inventory.

The below screenshot shows the buy function implementation in our project:

```
def buy(t):
    if agent.balance > stock_prices[t]:
        agent.balance -= stock_prices[t]
        agent.inventory.append(stock_prices[t])
        return 'Buy: ${:.2f}'.format(stock_prices[t])
```

3. **Basic 'Sell' functionality**:

- The 'Sell' function allows the agent to sell a stock on date 't' which is the input to this function.
- If the agent has stocks in the inventory, we sell the stock.
- We simultaneously increase the portfolio balance by the price sold and calculate the profit made using the price at which the stock was previously bought.

The below screenshot shows the sell function implementation in our project:

```
def sell(t):
    if len(agent.inventory) > 0:
        agent.balance += stock_prices[t]
        bought_price = agent.inventory.pop(0)
        profit = stock_prices[t] - bought_price
        global reward
        reward = profit
        return 'Sell: ${:.2f} | Profit: ${:.2f}'.format(stock_prices[t], profit)
```

5.8 DQN methodology used in our study:

Step - 1: Create a DQN Agent object

The first step is to create an agent by setting appropriate values for:

- Input state dimension
- Output action dimension
- Replay memory and buffer size
- Gamma, epsilon, epsilon_min, epsilon_decay
- Selected Neural network
- And initial portfolio balance

Step - 2: Choosing exploration or exploitation

- For every episode, we reset balance to the default value defined and the hyperparameters mentioned in section 5.3.
- We also get the current state and next state representations for the specified window size.
- We now use the **Epsilon Greedy** strategy to determine whether to choose exploration or exploitation.
- If it is an exploration, a random action is generated and explored (hold, buy or sell).
- If it is an exploitation, the neural network prediction is carried out (hold, buy or sell).

Step - 3: Calculating the rewards

- For no action at any step, a penalty is incurred to the portfolio. This is due to a missed opportunity to invest in Treasury bonds that provide risk free returns over a long period of time (over 30 years).

```
def treasury_bond_daily_return_rate():
    r_year = 2.75 / 100  # approximate annual U.S. Treasury bond return rate
    return (1 + r_year)**(1 / 365) - 1
```

- The net profit is calculated by subtracting the initial default balance from the final portfolio value and awarded to the agent as reward.
- The experience in this step is then added to the replay buffer to continue training the neural network.

Step - 4: Calculating loss

- The loss is calculated for mini batches of these experiences based on the defined buffer size. The optimal Q value is calculated using the Bellman equation and model predictions.
- Huber loss and mse is used to train the neural network on invoking the model.fit() for training for the defined number of epochs.

Summary of the Algorithm:

Final DQN Algorithm

1.Initialize parameters for *Q(s, a)* and $\hat{Q}(s,a)$ with random weights, $\epsilon \leftarrow 1.0$, and empty replay buffer
2.With probability ϵ, select a random action *a*, otherwise $a = \arg\max_a Q_{s,a}$
3.Execute action *a* in an emulator and observe reward *r* and the next state *s′*
4.Store transition *(s, a, r, s′)* in the replay buffer
5.Sample a random minibatch of transitions from the replay buffer
6.For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step (*y* $= r + \gamma \max_{a' \in A} \hat{Q}_{s',a'}$ otherwise
7.Calculate loss: $\mathcal{L} = (Q_{s,a} - y)^2$
8.Update *Q(s, a)* using the SGD algorithm by minimizing the loss in respect to model parameters

9.Every *N* steps copy weights from Q to $\hat{Q}$
10.Repeat from step 2 until converged

Chapter 6. Implementation and Results

In this section we provide an in depth comparison of the various DQN models we have trained on different datasets, hyperparameters and the underlying neural networks.

6.1 Execution details

```
Python 3.7.6 64-bit ('deepRLTrader': conda)
(cached) ~/opt/anaconda3/envs/deepRLTrader/bin/python
```

As a prerequisite, we install the following dependencies to enable training our DQN agent:

```
msgpack>=0.6.1
tensorflow==2.1.2
grpcio>=1.24.3
h5py>=2.9.0
numpy>=1.18.1
scipy>=1.4.1
pandas>=1.0.0
matplotlib>=3.1.2
empyrical>=0.5.0
statsmodels>=0.10.2
yahoofinancials==1.5
```

On execute the training code, we can see that the training starts:

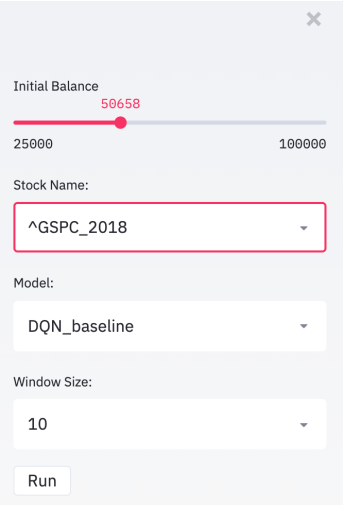
```
2021-11-30 15:38:31.130142: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this Tens
orFlow binary was not compiled to use: AVX2 AVX512F FMA
2021-11-30 15:38:31.153330: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7fcf42c75560 initialized for platf
orm Host (this does not guarantee that XLA will be used). Devices:
2021-11-30 15:38:31.153361: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Versi
on
```

The logs can be found at 'logs/{model_name}_training_{stock_name}.log'' (Eg: DQN_training_^GSPC_2010-2015.log). It can be visualized as follows:

```
[11/30/2021 22:21:46.887 train_best_ep4.py: 70] Trading Object:           ^GSPC_2010-2015
[11/30/2021 22:21:46.887 train_best_ep4.py: 71] Trading Period:           1509 days
[11/30/2021 22:21:46.887 train_best_ep4.py: 72] Window Size:              5 days
[11/30/2021 22:21:46.887 train_best_ep4.py: 73] Training Episode:         4
[11/30/2021 22:21:46.887 train_best_ep4.py: 74] Model Name:               DQN_best_ep4
[11/30/2021 22:21:46.887 train_best_ep4.py: 75] Initial Portfolio Value: $50,000
[11/30/2021 22:21:46.887 train_best_ep4.py: 79]
Episode: 1/4
[11/30/2021 22:21:46.948 train_best_ep4.py:100] Step: 1 Hold signal: 0.2531     Buy signal: 0.1811  Sell signal: 0.5658
[11/30/2021 22:21:46.966 train_best_ep4.py:100] Step: 2 Hold signal: 0.2486     Buy signal: 0.1814  Sell signal: 0.57
[11/30/2021 22:21:46.967 train_best_ep4.py:101]         'Hold' is an exploration.
[11/30/2021 22:21:46.985 train_best_ep4.py:100] Step: 3 Hold signal: 0.2515     Buy signal: 0.1785  Sell signal: 0.57
[11/30/2021 22:21:46.985 train_best_ep4.py:101]         'Buy' is an exploration.
[11/30/2021 22:21:46.985 train_best_ep4.py:116] Buy: $1141.69
[11/30/2021 22:21:47.004 train_best_ep4.py:100] Step: 4 Hold signal: 0.251  Buy signal: 0.1786  Sell signal: 0.5704
[11/30/2021 22:21:47.005 train_best_ep4.py:101]         'Hold' is an exploration.
[11/30/2021 22:21:47.022 train_best_ep4.py:100] Step: 5 Hold signal: 0.2429     Buy signal: 0.295   Sell signal: 0.4621
[11/30/2021 22:21:47.022 train_best_ep4.py:101]         'Hold' is an exploration.
[11/30/2021 22:21:47.040 train_best_ep4.py:100] Step: 6 Hold signal: 0.2499     Buy signal: 0.2889  Sell signal: 0.4611
[11/30/2021 22:21:47.040 train_best_ep4.py:116] Sell: $1136.22 | Profit: $-5.47
[11/30/2021 22:21:47.058 train_best_ep4.py:100] Step: 7 Hold signal: 0.2478     Buy signal: 0.2916  Sell signal: 0.4605
```

6.2 Streamlit integration

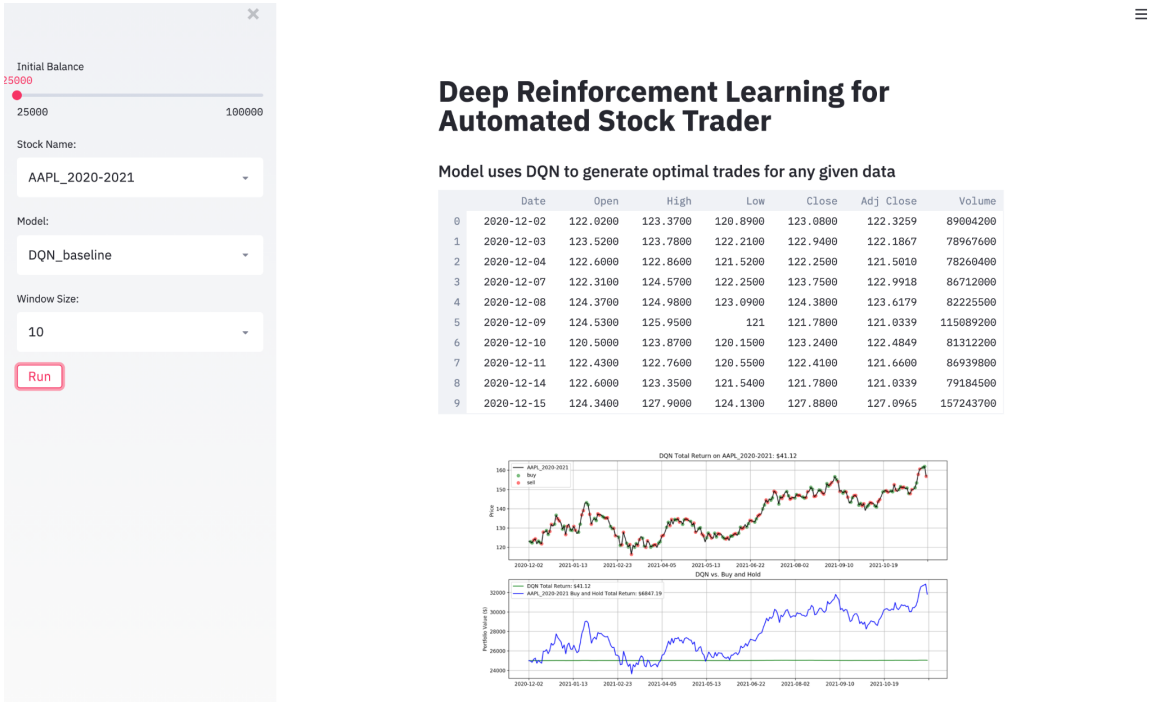
We have created a Streamlit application to evaluate our DQN stock trader. We have loaded all our trained DQN models and datasets to ease the evaluation process:



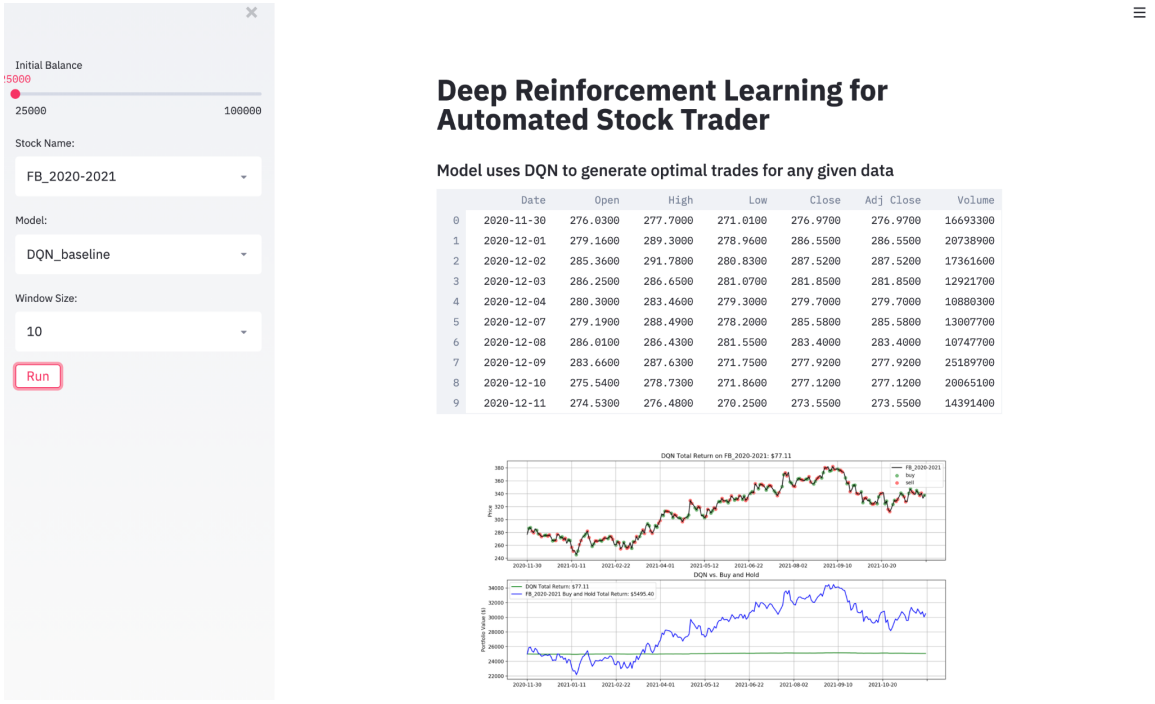
6.3 Evaluation results of the vanilla version - DQN_baseline.h5

Below are the evaluation results as visualized on the Streamlit application for each of our models on the respective datasets. The graphs shown below represent the stock trading trend (buy, sell and hold) the agent has followed for the given data sets against the trend if all the stocks were bought on the same day at every point in time.

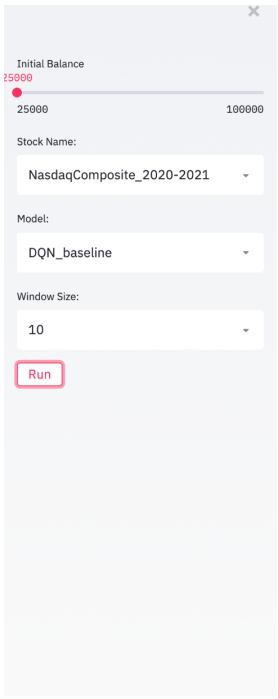
1. Dataset: Apple (AAPL_2020-2021)

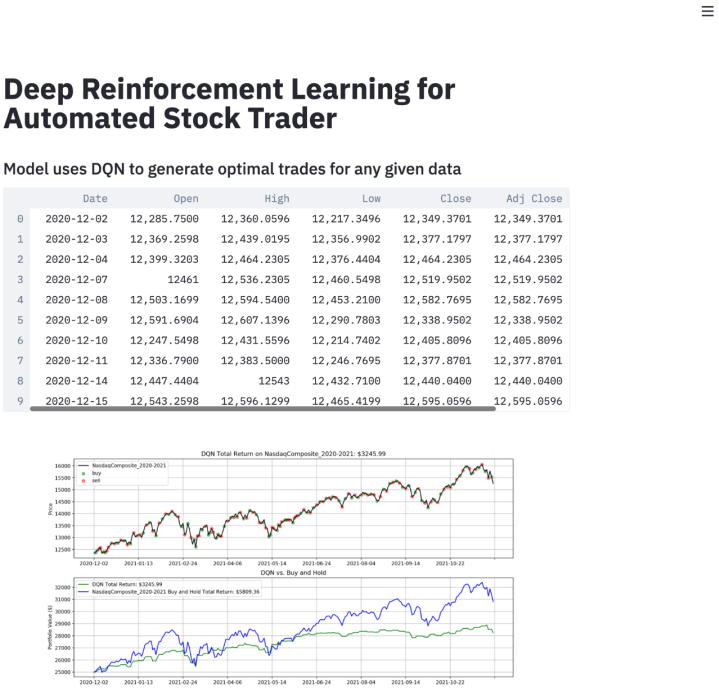


2. Dataset: Facebook (FB_2020-2021.csv)

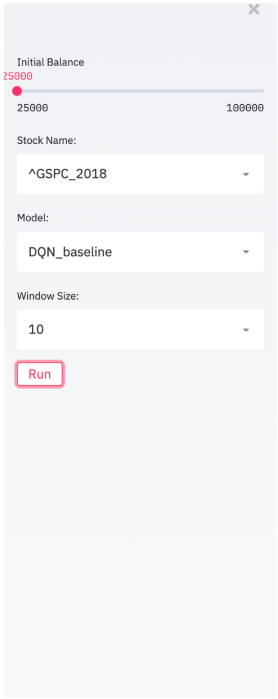


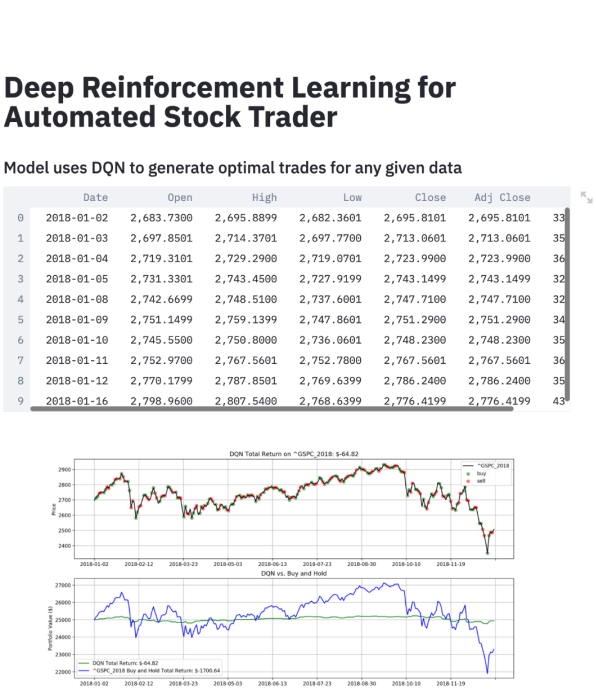
3. Dataset: Nasdaq Composite (NasdaqComposite_2020-2021)





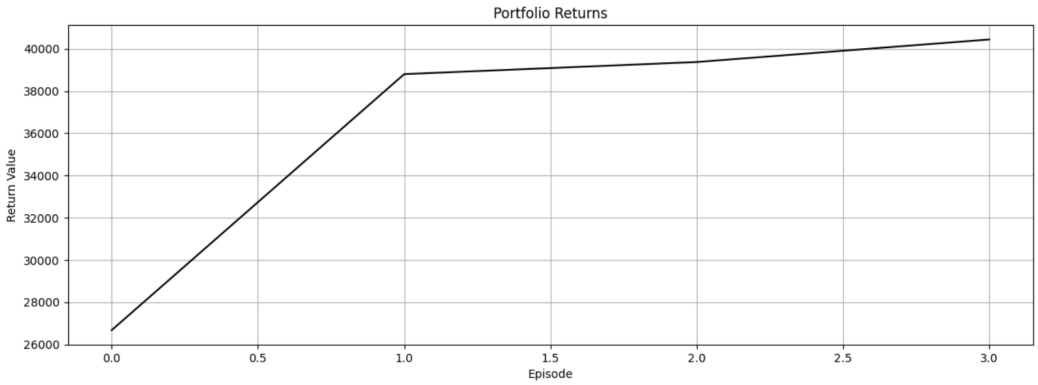
4. Dataset: ^GSPC_2018



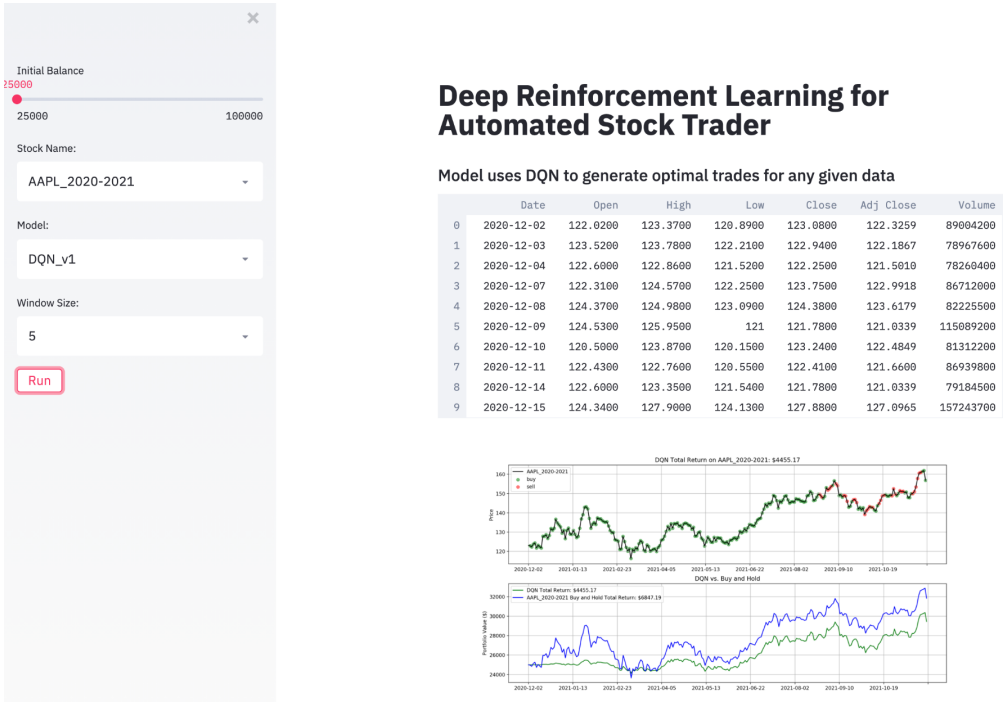


6.4 Evaluation results of the modified version - DQN_v1.h5

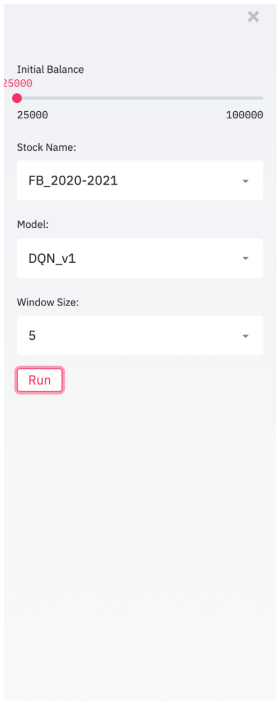
Training results across the 4 epochs the DQN agent was trained for:

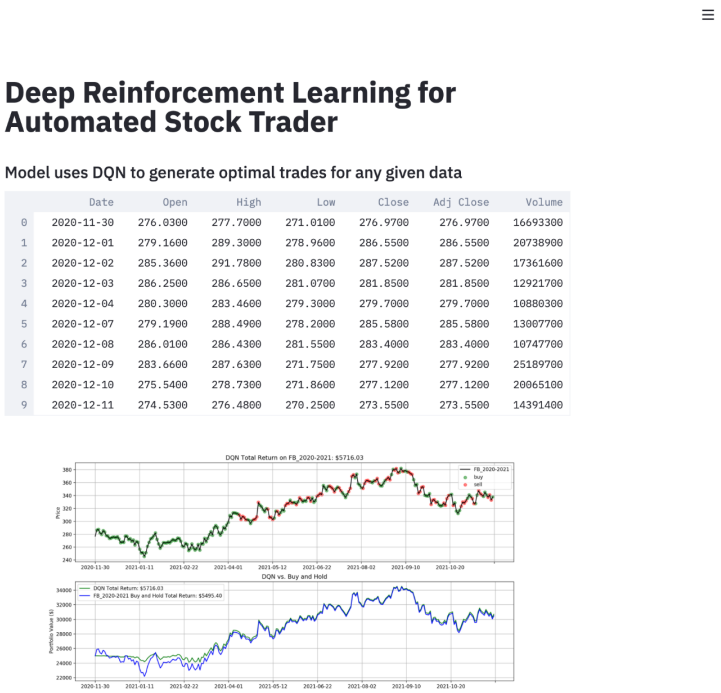


1. Dataset: Apple (AAPL_2020-2021)

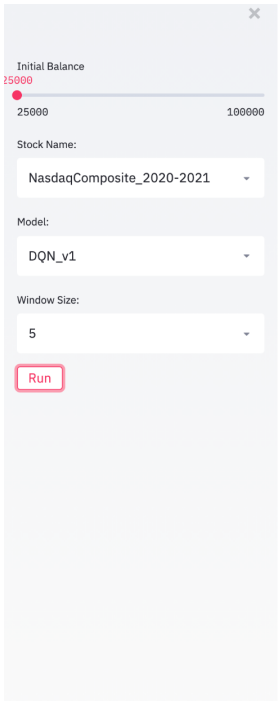


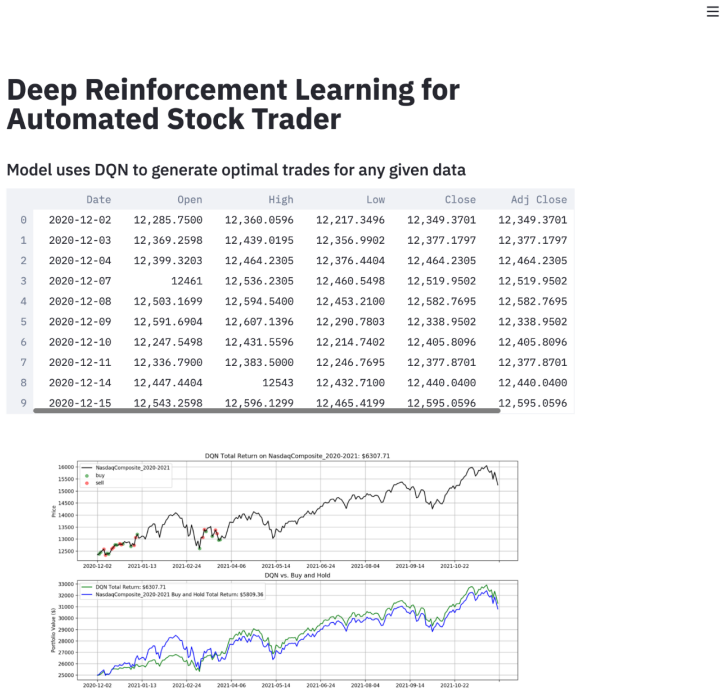
2. Dataset: Facebook (FB_2020-2021.csv)



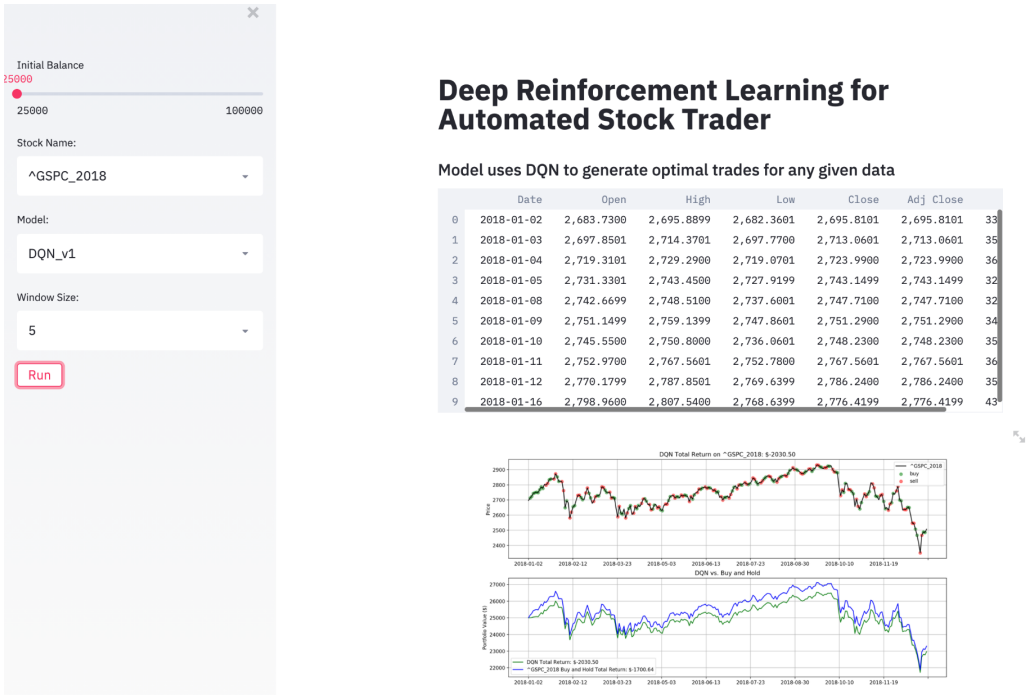


3. Dataset: Nasdaq Composite (NasdaqComposite_2020-2021)



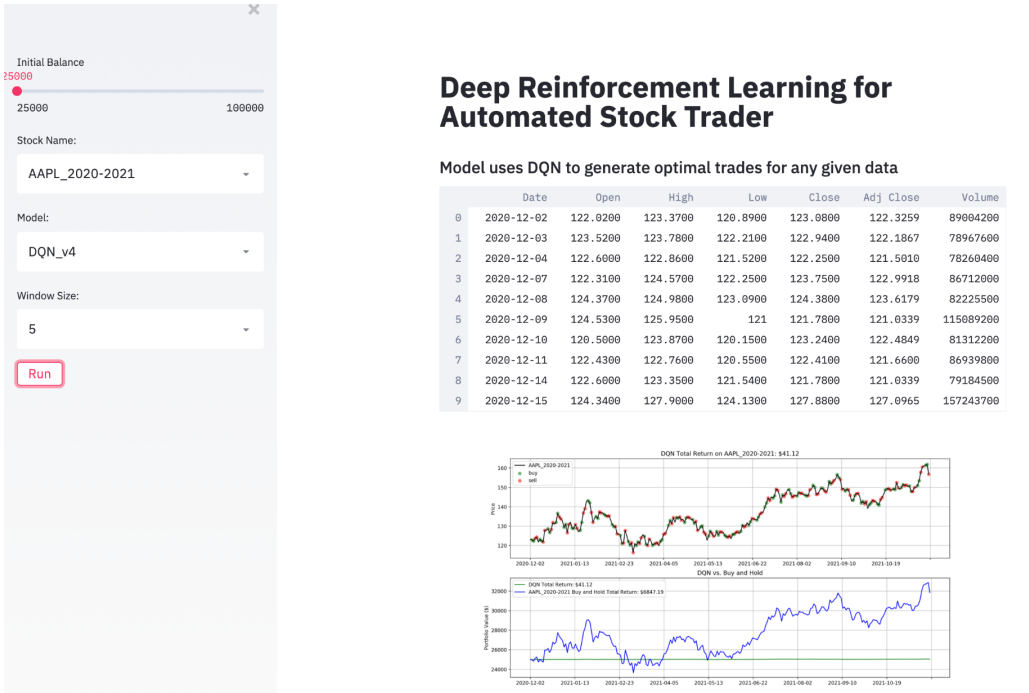


4. Dataset: ^GSPC_2018

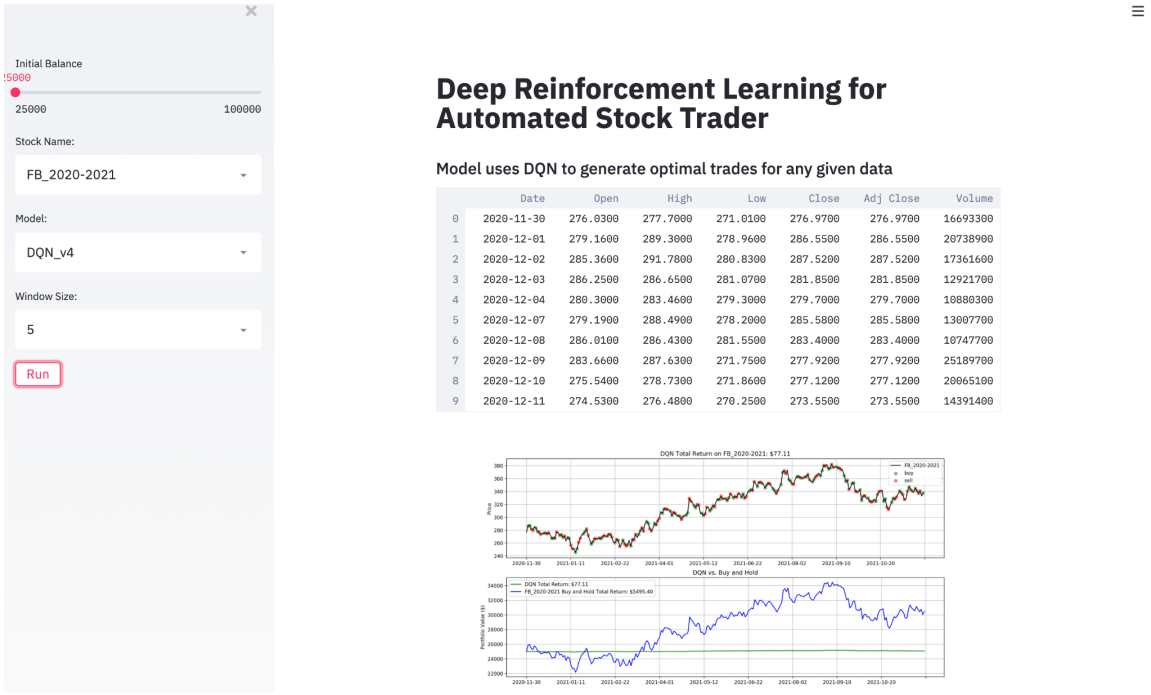


6.5 Evaluation results of the modified version - DQN_v4.h5

1. Dataset: Apple (AAPL_2020-2021)

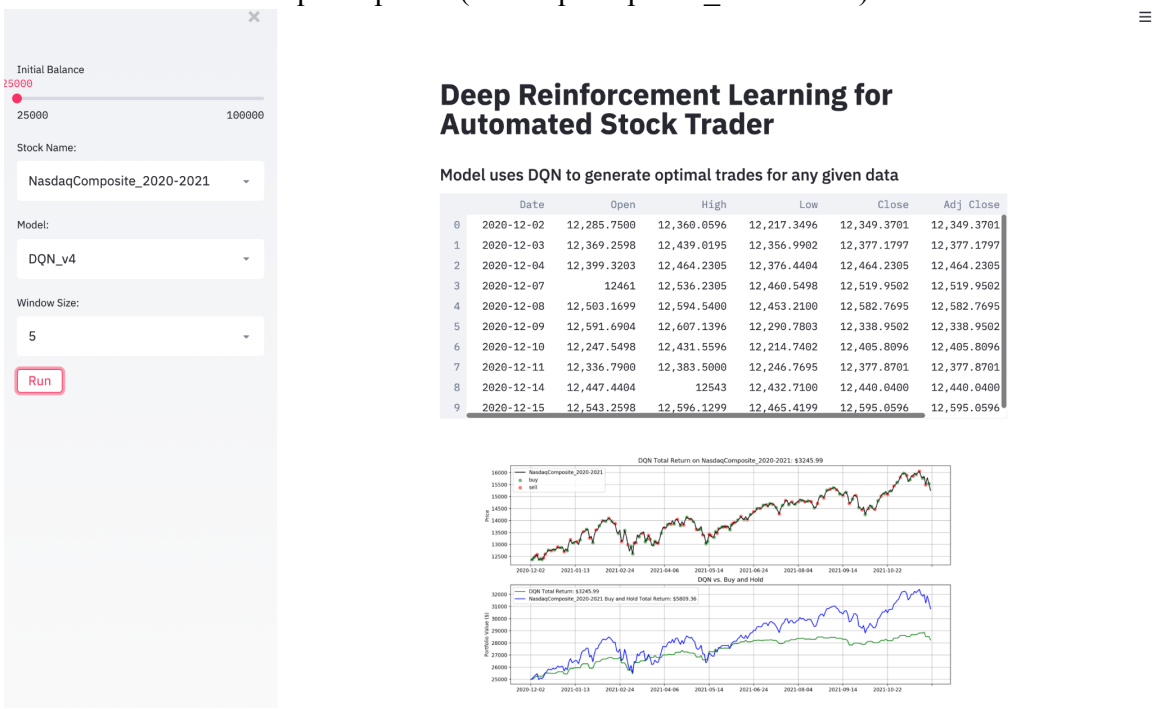


2. Dataset: Facebook (FB_2020-2021.csv)



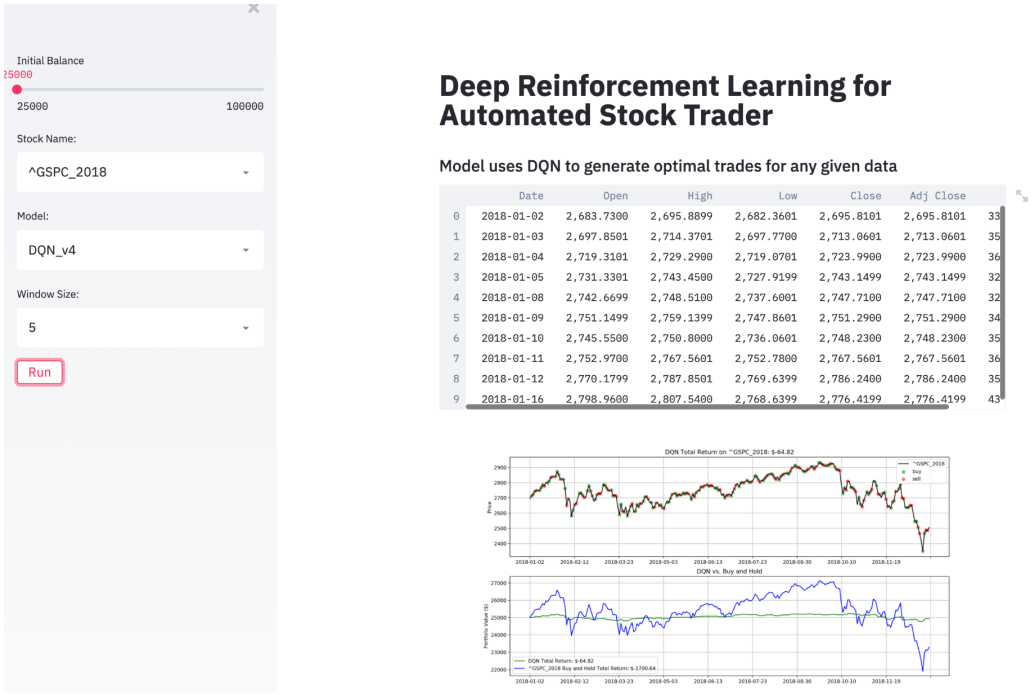
	Date	Open	High	Low	Close	Adj Close	Volume
0	2020-11-30	276.0300	277.7000	271.0100	276.9700	276.9700	16693300
1	2020-12-01	279.1600	289.3000	278.9600	286.5500	286.5500	20738900
2	2020-12-02	285.3600	291.7800	280.8300	287.5200	287.5200	17361600
3	2020-12-03	286.2500	286.6500	281.0700	281.8500	281.8500	12921700
4	2020-12-04	280.3000	283.4600	279.3000	279.7000	279.7000	10880300
5	2020-12-07	279.1900	288.4900	278.2000	285.5800	285.5800	13007700
6	2020-12-08	286.0100	286.4300	281.5500	283.4000	283.4000	10747700
7	2020-12-09	283.6600	287.6300	271.7500	277.9200	277.9200	25189700
8	2020-12-10	275.5400	278.7300	271.8600	277.1200	277.1200	20065100
9	2020-12-11	274.5300	276.4800	270.2500	273.5500	273.5500	14391400

3. Dataset: Nasdaq Composite (NasdaqComposite_2020-2021)



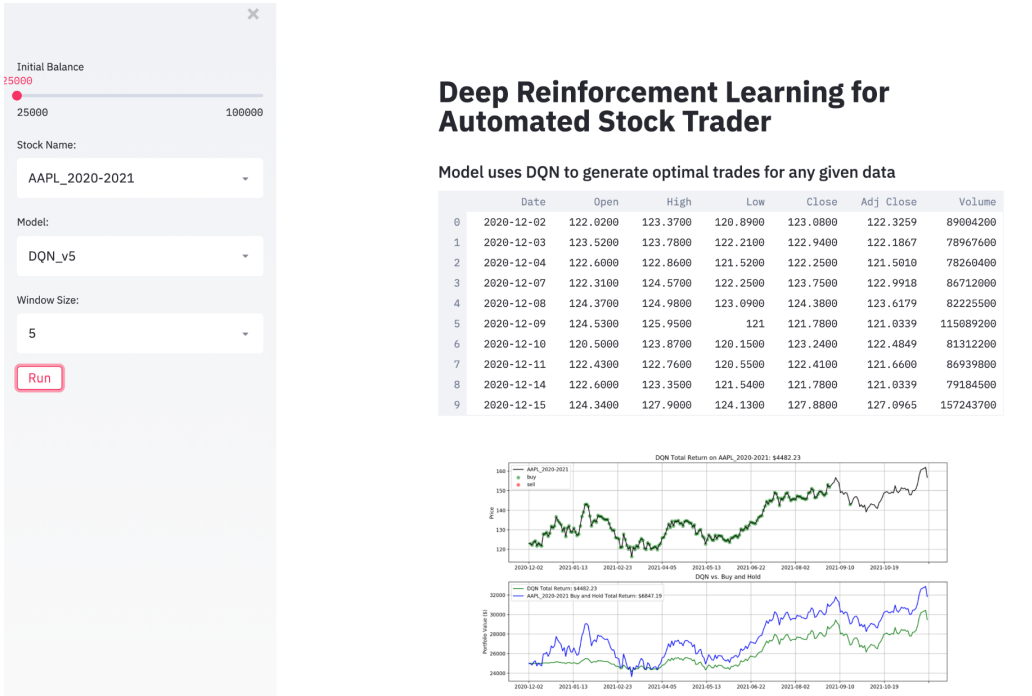
	Date	Open	High	Low	Close	Adj Close
0	2020-12-02	12,285.7500	12,360.0596	12,217.3496	12,349.3701	12,349.3701
1	2020-12-03	12,369.2598	12,439.0195	12,356.9902	12,377.1797	12,377.1797
2	2020-12-04	12,399.3203	12,464.2305	12,376.4404	12,464.2305	12,464.2305
3	2020-12-07	12461	12,536.2305	12,460.5498	12,519.9502	12,519.9502
4	2020-12-08	12,503.1699	12,594.5400	12,453.2100	12,582.7695	12,582.7695
5	2020-12-09	12,591.6904	12,607.1396	12,290.7803	12,338.9502	12,338.9502
6	2020-12-10	12,247.5498	12,431.5596	12,214.7402	12,405.8096	12,405.8096
7	2020-12-11	12,336.7900	12,383.5000	12,246.7695	12,377.8701	12,377.8701
8	2020-12-14	12,447.4404	12543	12,432.7100	12,440.0400	12,440.0400
9	2020-12-15	12,543.2598	12,596.1299	12,465.4199	12,595.0596	12,595.0596

4. Dataset: ^GSPC_2018

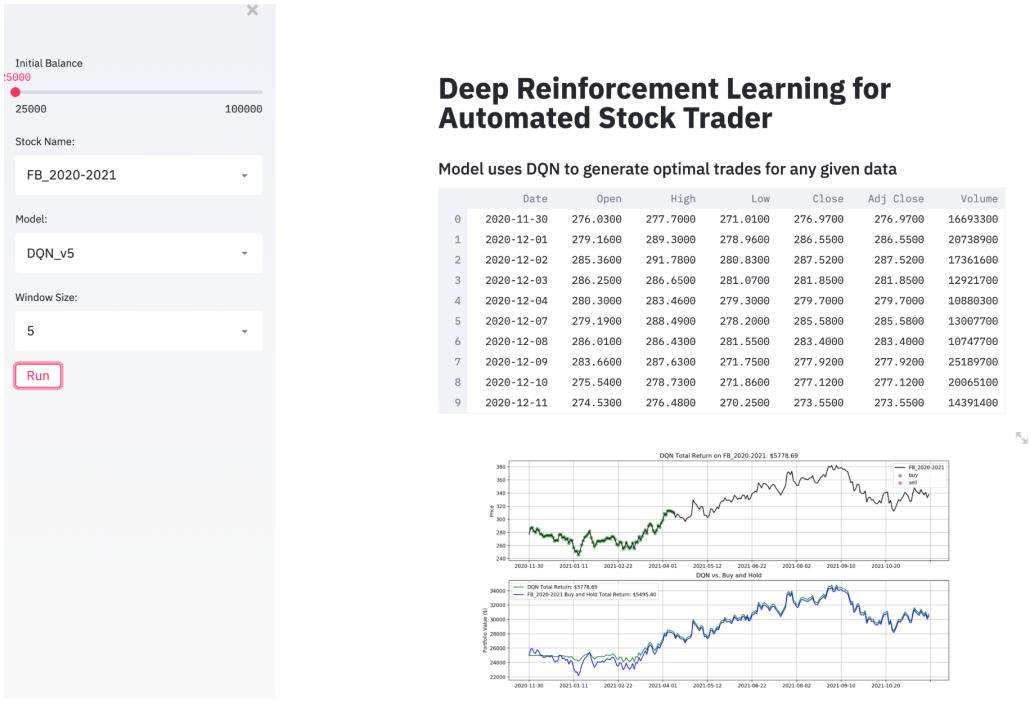


6.6 Evaluation results of the modified version - DQN_v5.h5

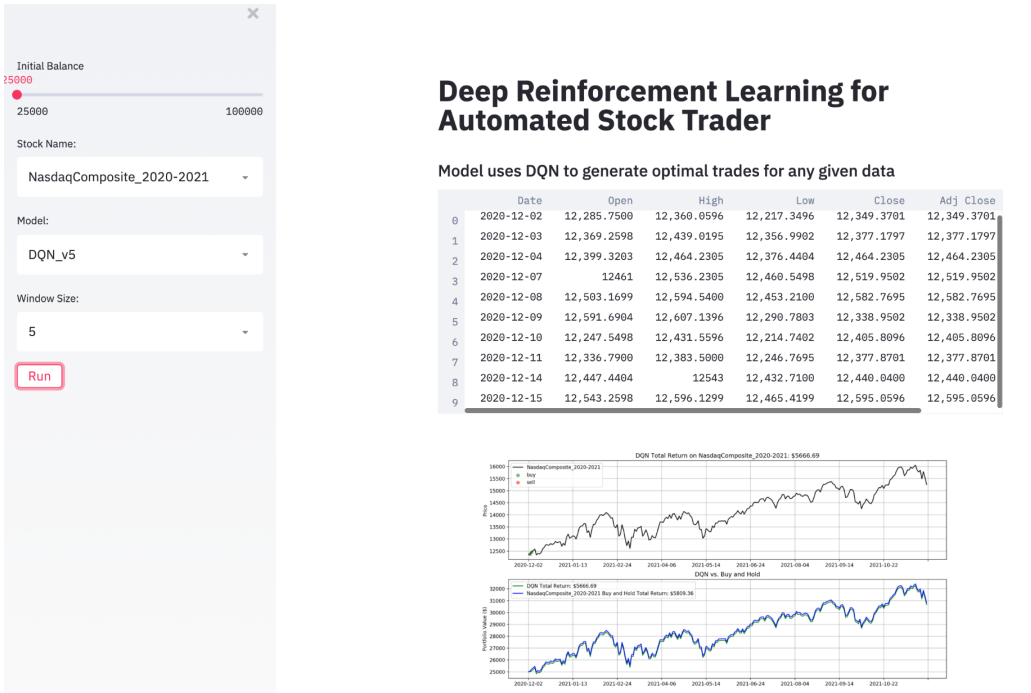
1. Dataset: Apple (AAPL_2020-2021)



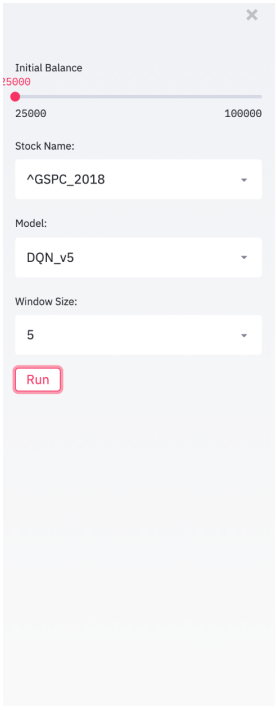
2. Dataset: Facebook (FB_2020-2021.csv)

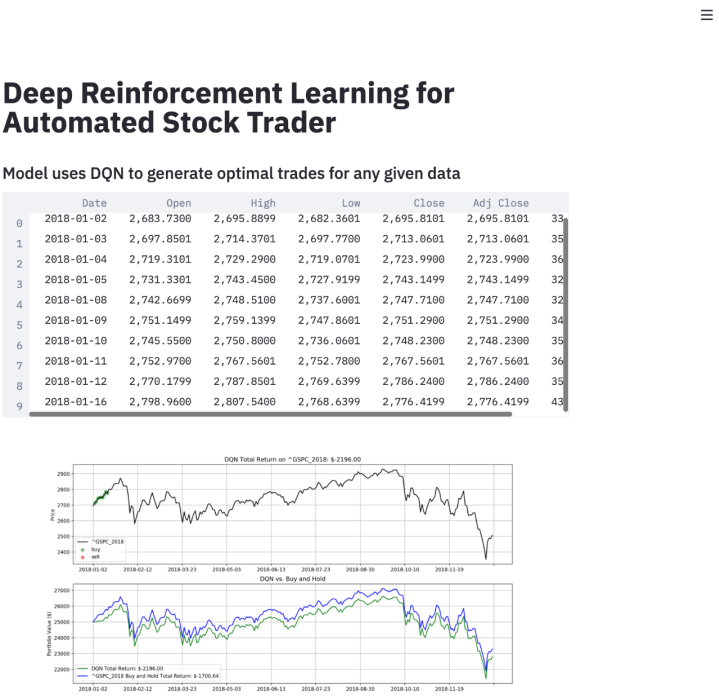


3. Dataset: Nasdaq Composite (NasdaqComposite_2020-2021)



4. Dataset: ^GSPC_2018





6.7 Comparison and Results

The table shows the comparison and results of all our model performances against the respective datasets (Overall profit made by the DQN agent).

Initial balance: 25000				**Datasets used to Evaluate**			
Model name	**Dataset used to train**	**Number of epochs**	**Input state dimension**	**FB 2020 - 2021**	**Apple 2020 - 2021**	**Nasdaq Comp 2020 - 21**	**GSPC 2018**
DQN_baseline	^GSPC_2010-2015	10	10 + action_dimension	Total returns = $77.11	Total returns = $41.12	Total returns = $3245.99	Total returns = -$64.82
DQN_v1	^GSPC_2010-2015	4	5 + action_dimension	Total returns = $5716.03	Total returns = $4455.17	Total returns = $6307.71	Total returns = -$2030.50
DQN_v4	^GSPC_2000-2017	2	5 + action_dimension	Total returns = $77	Total returns = $41.12	Total returns = $3245.99	Total returns = -$64.82
DQN_v5	Apple 2016-2021	2	5 + action_dimension	Total returns = $5778.69	Total returns = $4482.23	Total returns = $5666.69	Total returns = -$2196.00

We can see that the model DQN_v1 we trained on ^GSPC_2010-2015 outperforms the DQN_baseline model in terms of the overall profit made.

Contribution summary:

	Team Contribution
Arpitha	1. Research - stock trading and its role in RL 2. One of the DQN model implementation 3. Integrating Streamlit application for evaluation 4. Report writing and presentation
Somya	1. Research - stock trading and its role in RL 2. One of the DQN model implementation 3. Dataset acquisition for training and evaluation 4. Report writing and presentation
Tripura	1. Research - stock trading and its role in RL 2. One of the DQN model implementation 3. Report writing and presentation

Conclusion and Future Scope

In this project, we have explored the potential of training different variations of DQN (Deep Q-Network) agents to explore and learn an optimal strategy for stock trading in the complex and dynamic stock market. We have trained three versions of a DQN agent on different datasets and experimented with various values of the underlying hyperparameters and neural networks (As discussed in Chapter 4). We noticed that the first version of our model trained on GSPC 2010 to 2015 performed better than the baseline DQN agent [3]. The main differentiator was that our model used randomly sampled experiences to train the DQN's neural network instead of using it sequentially backwards as in the baseline model with mini batches at a time (As discussed in Chapter 6). The reason for choosing random samples is to break the correlation that exists between consecutive samples.

Currently, the agent is built to only support a single stock type and is allowed only 3 basic actions including buy, hold and sell. As part of future work, building an agent to support multiple stock types that can perform more complex actions like short selling would really take RL based stock trading up a notch. This project aims at performing only one action at the end of each trade day, an enhancement would be to place trades more often. Another direction for research is to be able to handle missing values in the data as our agent does not allow any missing data in the price history. Trading done by the agent is for academic purposes only and hence, does not incur any transaction costs during trading.

Appendix

Appendix A. Description of Implementation Repository

https://github.com/arpithagurumurthy/CMPE260_Reinforcement_Learning

**We have included Tensorboard logs in the GitHub repository.

Appendix B. References

1. Professor Jahan Ghofraniha lecture slides - Deep Q-Learning
2. Keon. (2019, December 9). Deep Q-Learning with Keras and Gym. Retrieved September 5, 2021, from https://keon.github.io/deep-q-learning/.
3. Guo, A. Z. (2019, June 6). Deep Reinforcement Stock Trading. Retrieved September 5, 2021, from https://github.com/Albert-Z-Guo/Deep-Reinforcement-Stock-Trading.
4. M, A. (2020, May 20). *RL DeepQLearning Trading*. RL-DeepQLearning-Trading. Retrieved September 5, 2021, from https://github.com/DeepNeuralAI/RL-DeepQLearning-Trading.
5. *Deep Q-learning - combining neural networks and reinforcement learning*. deeplizard. (n.d.). Retrieved September 5, 2021, from https://deeplizard.com/learn/video/wrBUkpiRvCA.
6. *Deep Q-learning - combining neural networks and reinforcement learning*. deeplizard. (n.d.). Retrieved September 5, 2021, from https://deeplizard.com/learn/video/wrBUkpiRvCA.
7. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December 19). *Playing Atari with deep reinforcement learning*. arXiv.org. Retrieved September 5, 2021, from https://arxiv.org/abs/1312.5602.
8. Yahoo Finance for Historical Stock Prices' data