



# DEEP REINFORCEMENT LEARNING FOR AUTOMATED STOCK TRADING

---

December 3, 2021

CMPE 260 Reinforcement Learning

# Team Invincibles

Arpitha Gurumurthy

Research + Variations to  
DQN + Streamlit  
integration + Report

Somya Mishra

Research + Variations to  
DQN + Dataset  
acquisition + Report

Tripura Gorla

Research + Variations to  
DQN + Report

# Problem Statement

- Financial trading is one of the most attractive areas in finance.
- Trading systems development is not easy task because it requires extensive knowledge in quantitative analysis, financial skills, and programming making it expensive.
- A human trading systems expert 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

---

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

---

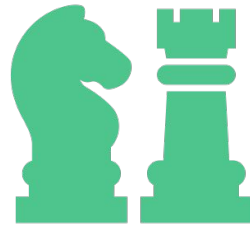
Using AI and RL, not only saves many resources but also reduces risk to a great extent if trained properly on appropriate and accurate data

# Motivation

# Motivation - continued



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



Evidences prove that DRL may have greater potential in automated stock trading

# Overview

A light-weight deep reinforcement learning framework for stock trading and portfolio management.

This project explores the possibility of applying deep reinforcement learning algorithms to stock trading in a highly modular and scalable framework.

# Methodology

---

Why DQN?

---

Data

---

Parameters used to train the DQN agent

---

Epsilon Greedy approach

---

Experience replay

---

Defining the action functions

---

Summarizing the steps in our DQN implementation

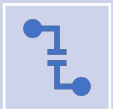
# Why DQN?



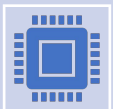
Incorporating a Q-learning algorithm with value iteration is suitable for use cases that have relatively simplistic environments and a fewer number of states and actions



Calculating and updating Q-values for each state-action pair in a huge state space becomes a computation burden and intractable due to the number 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



Hence, in this project, we have used Deep Q Network to implement our stock trader given the complexity of the environment which is the ever-fluctuating stock market



# Datasets

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

- GSPC 2010-2015
- GSPC 2000-2017
- Apple (AAPL): Historical daily data from 2016-2021

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

- GSPC 2014
- GSPC 2016
- GSPC 2018
- FB 2018
- Nasdaq composite: Historical daily data from 2011-2021
- Apple (AAPL): Historical daily data from 2016-2021
- Facebook (FB): Historical daily data from 2016-2021

# Data

---

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

---

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.

	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

Sample dataset

# Epsilon Greedy approach

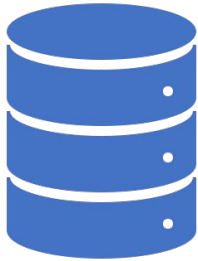
- When the agent takes random actions in the stock market environment, it does not learn very well and hence, the Epsilon-Greedy approach is employed
- In this strategy the agent will initially explore the environment in order to gain knowledge and then exploit it
- It forces the agent to learn to balance 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

# Exploration versus exploitation in our project

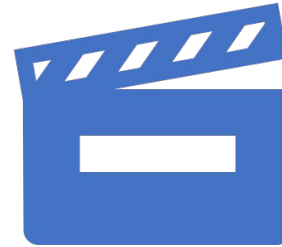
- In order to implement this in our DQN stock trader, we first generate a value between 0 and 1, if this random value is less than the value of Epsilon, exploration is carried out
- Otherwise, the next action is chosen via exploitation which is essentially the model prediction

```
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])
```

# 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:**

Current state of the environment at time 't'

The action 'a' taken at the current state

The reward given to the agent at time 't+1'

Next state of the environment at time 't+1' on performing the action 'a'

# Experience replay continued

- 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

# Parameters used to train the DQN agent

---

Trading period

---

Action dimensions

---

Memory

---

Buffer size

---

Gamma

---

Epsilon

---

Epsilon decay





# Important definitions

---

- **Inventory** - Inventory in our study is an array containing the stock prices of all the stocks the agent has bought
- **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.
- **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.
- **Balance** - This refers to the amount of money the agent has when the trading begins or the initial account balance.

# Defining the action functions



BUY



HOLD



SELL

# 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.

# 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.

# 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.

# Summarizing the steps in our DQN implementation

---

## 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 the portfolio balance and the hyperparameters.
- We get the current state representation 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).
- 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 the 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:

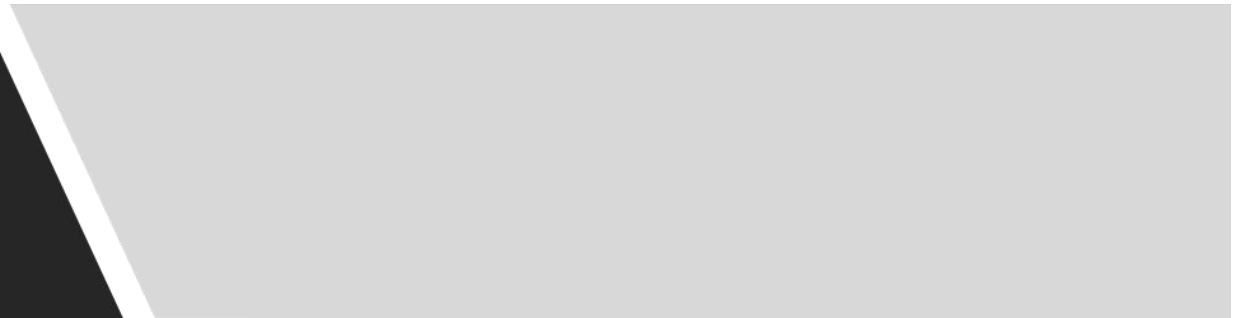
## Final DQN Algorithm

---

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  $\epsilon$ , 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



# Results



# DQN Baseline

The baseline model 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
```

# DQN Variation 1

Below are some of the modifications incorporated:

- Trained on GSPC 2010 to 2015 dataset.
- Increased the replay memory size from 100 to 10000 and reduced the buffer size 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 updated the Bellman Equation:

```
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
```

# DQN Variation 2

Below are some of the modifications incorporated:

- Trained on GSPC 2000 to 2017 for 2 episodes
- Increased the replay memory size from 100 to 10000 and updated the buffer size 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
- The underlying neural network is same as the modified DQN Part 1
- Used 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
```



# DQN Variation 3

Below are some of the modifications incorporated:

- Trained on Apple 2016-2021 dataset
- Updated the Gamma value from 0.95 to 0.9
- Updated the Epsilon decay value to 0.85
- 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
```



Demo



Initial balance: 25000			Datasets used to Evaluate			
Model name	Dataset used to train	Number of epochs	FB 2020 - 2021	Apple 2020 - 2021	Nasdaq Comp 2020 - 21	GSPC 2018
<b>DQN_v1</b>	^GSPC_2010-2015	4	Total returns = \$5716.03	Total returns = \$4455.17	Total returns = \$6307.71	Total returns = -\$2030.50
<b>DQN_v4</b>	^GSPC_2000-2017	2	Total returns = \$77	Total returns = \$41.12	Total returns = \$3245.99	Total returns = -\$64.82
<b>DQN_v5</b>	Apple 2016-2021	2	Total returns = \$5778.69	Total returns = \$4482.23	Total returns = \$5666.69	Total returns = -\$2196.00

Comparing model variants' results against same datasets

# Challenges



One of the major problems encountered was during model training, the model prediction was stuck on just 1 of the actions.



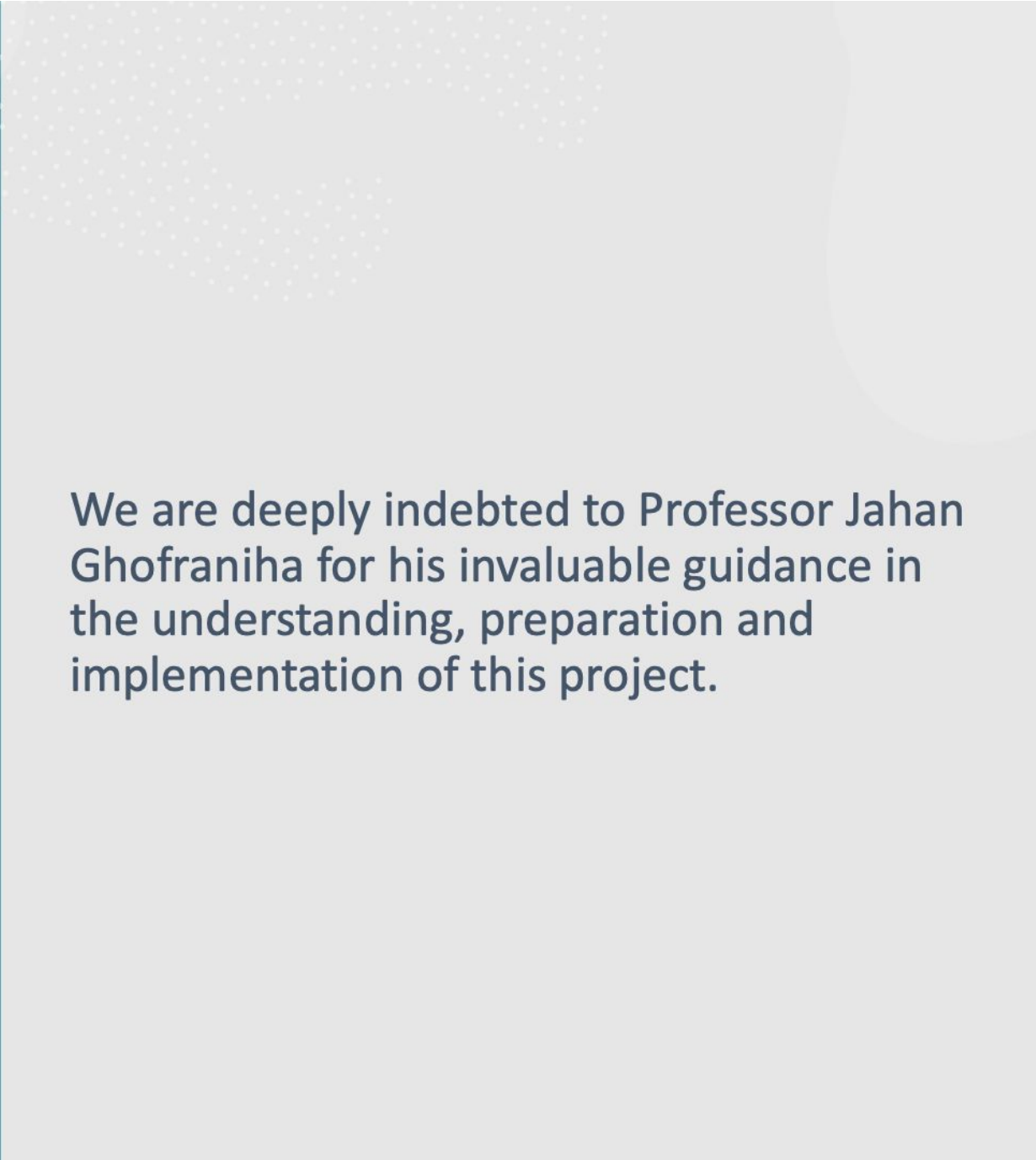
Extremely long training times with an approximate of 6 to 10 hours.



The Tensorflow and python version required to run the program is very specific



# Thank you!



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

# Link to our codebase

[https://github.com/arpithagurumurthy/CMPE260\\_Reinforcement\\_Learning](https://github.com/arpithagurumurthy/CMPE260_Reinforcement_Learning)



# References

- Professor Jahan Ghofraniha lecture slides - Deep Q-Learning
- Deep Q-Learning with Keras and Gym - <https://keon.github.io/deep-q-learning/>
- DQN implementation inspiration: <https://github.com/Albert-Z-Guo/Deep-Reinforcement-Stock-Trading>
- Streamlit inspiration: <https://github.com/DeepNeuralAI/RL-DeepQLearning-Trading>
- <https://deeplizard.com/learn/video/wrBUkpiRvCA>
- <https://arxiv.org/pdf/1312.5602.pdf>
- Yahoo Finance for Historical Stock Prices' data