# Applying Deep Q-Learning to Stock Trading

Tutorial notes by Jerry Orajekwe (23031035)

Github link: https://github.com/jerryorajekwe/Application_of_DeepQLearning_to_Stocktrading.git

Word count (1969)

## Contents

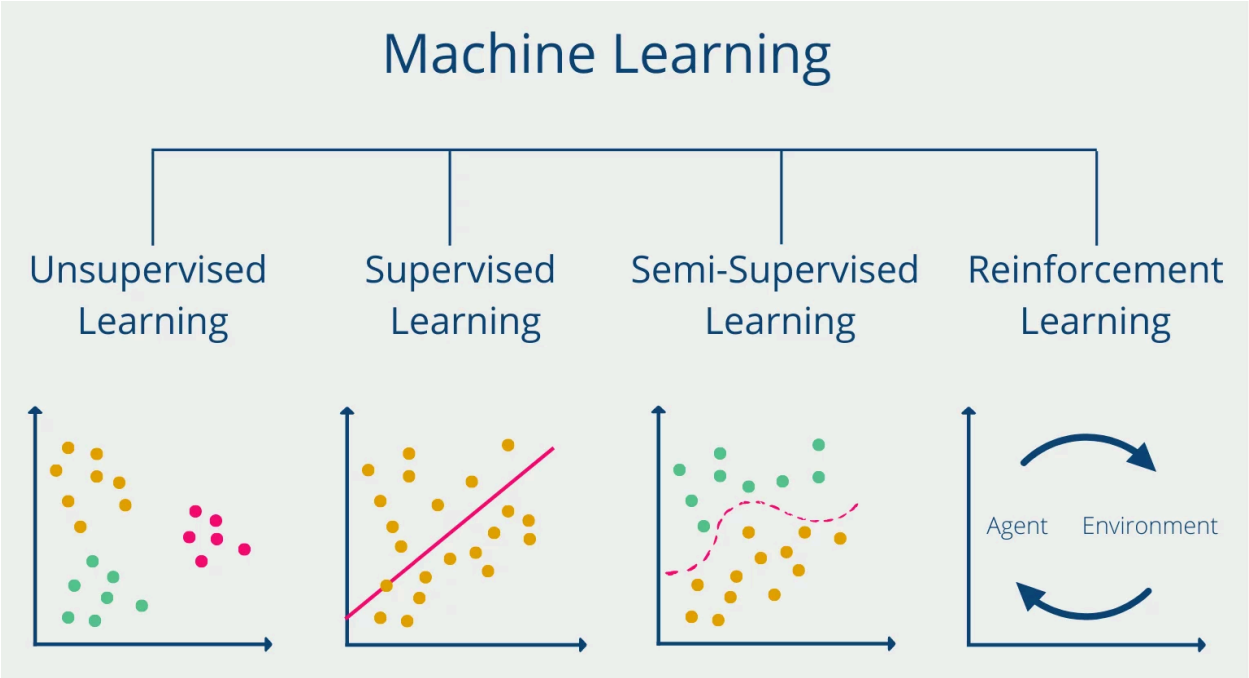
## 1. Introduction


Source: [stock trading - Search Images (bing.com)]

Stock trading is a challenging domain characterized by unpredictable price movements, complex market dynamics, and high risks. Traditional methods often rely on statistical models or human intuition to make decisions. However, with advancements in artificial intelligence, reinforcement learning (RL) has emerged as a promising approach to automating and optimizing trading strategies.

---

## What is Reinforcement Learning?




Source: [reinforcement learning a part of machine learning - Search Images (bing.com)]

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by interacting with an environment. Unlike supervised learning, where the model is provided with labeled examples, the RL agent discovers optimal actions by trial and error to maximize cumulative rewards.

In this paradigm:

- The **agent** (our trading algorithm) interacts with the **environment** (stock market simulation).
- The agent observes the **state** (e.g., stock price, indicators) and performs **actions** (buy, sell, hold).
- The agent receives a **reward** (profit or loss) based on its actions, guiding it to improve its strategy over time.

---

## Why Deep Q-Learning for Stock Trading?

Deep Q-Learning (DQL) is an advanced RL technique that combines the Q-Learning algorithm with **deep neural networks.** It's well-suited for stock trading because:

i. **Sequential Nature of Trading:** Stock prices form a time series in which future trends depend on past movements. RL inherently handles such sequential decision-making.
ii. **High-Dimensional States:** Traditional Q-Learning struggles when the state space is large. DQL uses neural networks to approximate Q-values for these high-dimensional states effectively.
iii. **Optimization Goals:** DQL maximizes cumulative portfolio returns rather than predicting future stock prices.

---

## Objective of This Tutorial

In this tutorial, we will:

a. Simulate a stock trading environment using historical price data.
b. Design a Deep Q-learning agent that learns optimal trading strategies.
c. Train and evaluate the agent, visualizing its trading performance.

By the end of this tutorial, you will understand how to:

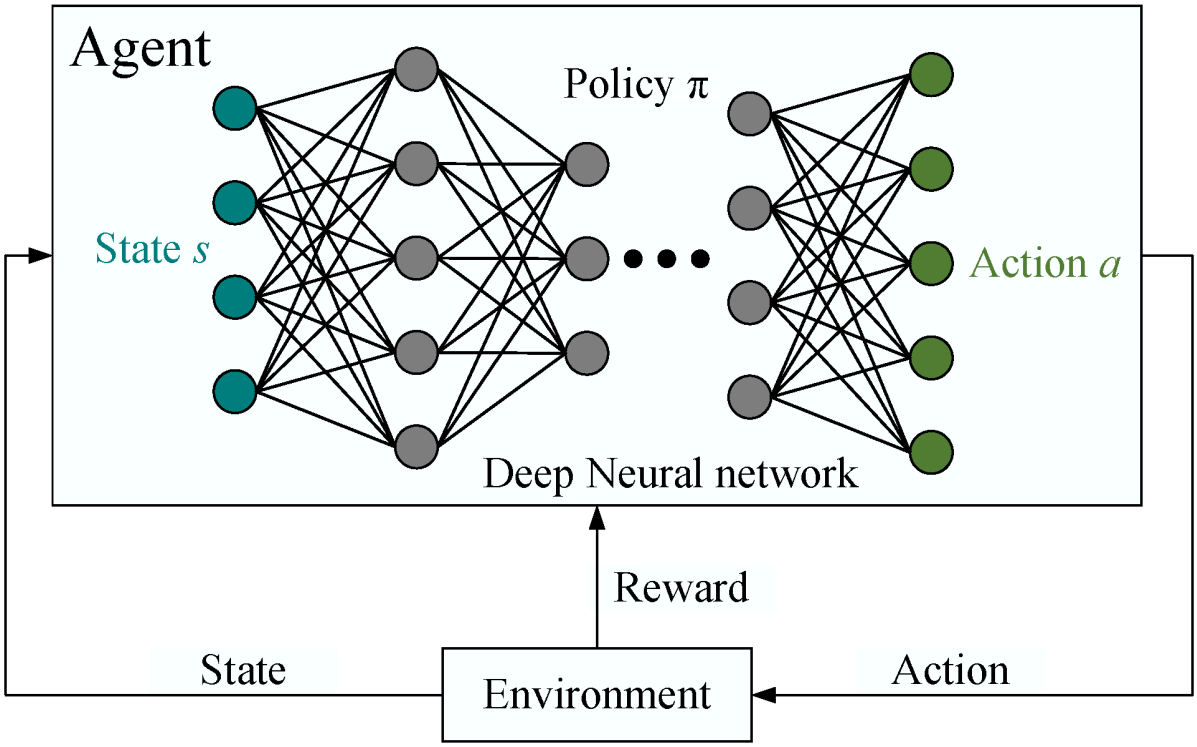
- Apply RL to stock trading problems.
- Implement a Deep Q-learning algorithm.
- Analyze and interpret trading results.

---

## 2. Problem Setup

In this section, we'll set up the stock trading simulation, define the problem, and describe the dataset.

---




Source: [reinforcement learning - Search Images (bing.com)]

## A. State

The state of the environment reflects current market conditions, encapsulated by features describing the stock's behavior. At any point, the state is a vector containing information on price movements, technical indicators, and past trends.

For this tutorial, the state vector will include:

- **Current Stock Price**: Price at the current time step.
- **Previous Prices**: Historical prices (e.g., last 5 days).
- **Technical Indicators**:
    - **Simple Moving Average (SMA-5)**: Average price over the last 5 days.

- **Simple Moving Average (SMA-10)**: Average price over the last 10 days.
- **Relative Strength Index (RSI)**: Measures speed and change of price movements to assess if a stock is overbought or oversold.

The state vector will be:

```
[state] = [Current Price, SMA-5, SMA-10, RSI, ...]
```

These features help the agent learn stock price patterns and decide what **actions** to take based on trends and current market conditions.

---

## B. Actions

In stock trading, the agent can:

- **Buy:** Purchase a fixed amount of stocks (e.g., $100,000 worth).
- **Sell:** Liquidates its position.
- **Hold:** Do nothing and wait for the next time step.

These actions are the agent's discrete choices at each time step.

---

## C. Reward

The reward is the feedback the agent receives after an action, reflecting changes in the portfolio's value.

We define the reward as:

$$\text{Reward} = \text{Change in Portfolio Value} \; ( P_{t+1} - P_t )$$

- Where $P_t$ is the portfolio value at time step t, and $P_{t+1}$ is the portfolio value at time step t+1.

- For a **buy** action, the agent incurs a cost (reducing cash reserves), but the reward depends on subsequent price movement.
- For a **sell** action, the agent receives cash from the sale, and the reward is based on the profit or loss compared to the purchase price.
- **Hold** actions receive a zero reward, as no money is spent or earned during that time step.

Additional consideration:

- **Penalty for excessive trading:** This helps to simulate real-world transaction costs (e.g., brokerage fees). To discourage frequent trades, the agent may receive a small negative reward for every buy or sell action.

---

## D. Dataset

For this tutorial, we'll use the yfinance library to access **Tesla's (TSLA)** historical stock price data from Yahoo Finance.

**Key data points:**

- Open Price
- Close Price
- High Price
- Low Price
- Volume

This data will be processed to generate the necessary technical indicators (SMA, RSI).

**Code snippet to download data:**

```
import yfinance as yf

# Define the stock ticker symbol
ticker = "TSLA"

# Download historical data
stock_data = yf.download(ticker, start="2015-01-01", end="2023-01-01")

# Show the first few rows of data
print(stock_data.head())
```

```
[**********100%**********]  1 of 1 completed
```

| Price | Date | Adjusted Closing Price | Closing Price | Highest Price | Lowest Price | Opening Price | Volume |
|---|---|---|---|---|---|---|---|
| Ticker | | TSLA | TSLA | TSLA | TSLA | TSLA | TSLA |
| 0 | 2015-01-02 00:00:00+00:00 | 14.620667 | 14.620667 | 14.883333 | 14.217333 | 14.858000 | 71466000 |
| 1 | 2015-01-05 00:00:00+00:00 | 14.006000 | 14.006000 | 14.433333 | 13.810667 | 14.303333 | 80527500 |
| 2 | 2015-01-06 00:00:00+00:00 | 14.085333 | 14.085333 | 14.280000 | 13.614000 | 14.004000 | 93928500 |
| 3 | 2015-01-07 00:00:00+00:00 | 14.063333 | 14.063333 | 14.318667 | 13.985333 | 14.223333 | 44526000 |
| 4 | 2015-01-08 00:00:00+00:00 | 14.041333 | 14.041333 | 14.253333 | 14.000667 | 14.187333 | 51637500 |

Once we have the data, we can preprocess it to calculate the moving averages and RSI.

---

## 3. Deep Q-Learning (DQL) Algorithm

This section explores the DQL algorithm, showing how our stock trading agent learns optimal strategies using neural networks to approximate Q-values.

---

## What is Q-Learning?

Before diving into Deep Q-Learning, it's essential to understand **basic Q-Learning**. In traditional Q-learning, the goal is for the agent to learn the **Q-function**:

$$Q(s,a) = \mathbb{E}[r_t + \gamma \max_{a'} Q(s_{t+1}, a')]$$

Where:

- Q(s,a) represents the expected future reward for taking action (a) in state (s).
- $r_t$ is the immediate reward after action (a) at time (t).
- γ is the discount factor for future rewards.
- $\max_{a'} Q(s_{t+1},a')$ represents the maximum Q-value for the next state $s_{t+1}$, across all possible actions a′.

The agent's goal is to find the action (a) that maximizes the Q-value for a given state (s) and leads to the best long-term reward.

---

## Why Use Deep Q-Learning?

Traditional Q-Learning uses a Q-table to store Q-values for each state-action pair. However, this becomes inefficient with large state and action spaces, like in stock trading. Deep Q-Learning addresses this by using neural networks to approximate Q-values, enabling the agent to handle more complex state spaces.

In Deep Q-Learning:

- The Q-function is approximated by a **neural network** (the **Q-network**).
- The agent uses this Q-network to predict the best action for any state.

---

## Deep Q-Learning Architecture

We will use the following architecture for our DQL algorithm:

i. **Q-Network:**

- A neural network that approximates the Q-values for each state-action pair.
- The input to the Q-network will be the state vector (features like stock price, SMA, and RSI).
- The output will be a vector of Q-values corresponding to each action (buy, sell, hold).

ii. **Action Selection (Epsilon-Greedy):**

- To balance exploration and exploitation, we use the **epsilon-greedy policy**.
- With probability $\epsilon$, the agent will choose a random action (exploration).
- With probability $1-\epsilon$, the agent will choose the action with the highest Q-value (exploitation).

The exploration rate $\epsilon$ starts high (encouraging exploration) and decays over time (encouraging exploitation as the agent learns).

iii. **Replay Buffer (Experience Replay):**

- In traditional Q-learning, the agent updates the Q-values immediately after each action. This can lead to instability due to correlated updates.
- **Experience Replay** solves this by storing the agent's experiences (state, action, reward, next state) in a replay buffer.
- At each time step, a random batch of experiences is sampled from the buffer, and the Q-network is updated based on these experiences. This helps to break correlations and stabilize training.

iv. **Target Network:**

- To further stabilize training, we use a **target Q-network**, a copy of the Q-network.
- The target network's weights are updated less frequently (e.g., every 10,000 steps), preventing oscillations and Q-values divergence.

---

## Training Process

The training process for our stock trading agent consists of the following steps:

i. **Initialize Networks:**
Set up Q-network (for training) and target network (for stability).

ii. **Initialize Replay Buffer:**
A replay buffer is initialized to store the agent's experiences.

iii. **Exploration and Action Selection:**
Use epsilon-greedy policy to select actions and interact with the environment.

iv. **Store Experience:**
The agent saves its experience (state, action, reward, next state) in the

replay buffer.

v. **Sample Batch:**
A random batch of experiences is sampled from the replay buffer.

vi. **Update the Q-Network:**
The Q-network is updated by minimizing the loss between predicted and target Q-values. The target Q-values are computed as:

$$y = r + \gamma \max_{a'} Q'(s_{t+1}, a')$$

Where Q′ is the target Q-network.

vii. **Update the Target Network:**
Periodically sync target network with Q-network.

viii. **Repeat:**
Steps iii to vii are repeated for each time step, and the agent continues learning to improve its trading strategy.

---

## 4. Implementation and Training

This section will guide you through implementing the key components: the Q-network, the epsilon-greedy policy, the experience replay, and the training loop.

---

## A. Setup and Import Libraries

First, import the necessary libraries: TensorFlow for the Q-network, NumPy for numerical operations, and Pandas for data manipulation.

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
import yfinance as yf
import random
import matplotlib.pyplot as plt
from collections import deque
```

---

## B. Load and Preprocess the Data

Next, download the stock data and preprocess it to compute the required technical indicators.

```
# Download Tesla stock data from Yahoo Finance
stock_data = yf.download('TSLA', start='2015-01-01', end='2023-01-01')

# Calculate the Simple Moving Averages (SMA)
stock_data['SMA-5'] = stock_data['Close'].rolling(window=5).mean()
stock_data['SMA-10'] = stock_data['Close'].rolling(window=10).mean()

# Calculate the Relative Strength Index (RSI)
delta = stock_data['Close'].diff()
gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
rs = gain / loss
stock_data['RSI'] = 100 - (100 / (1 + rs))

# Drop NaN values resulting from rolling operations
stock_data = stock_data.dropna()

# Visualize the data
stock_data[['Close', 'SMA-5', 'SMA-10', 'RSI']].plot(figsize=(12, 6))
plt.show()
```

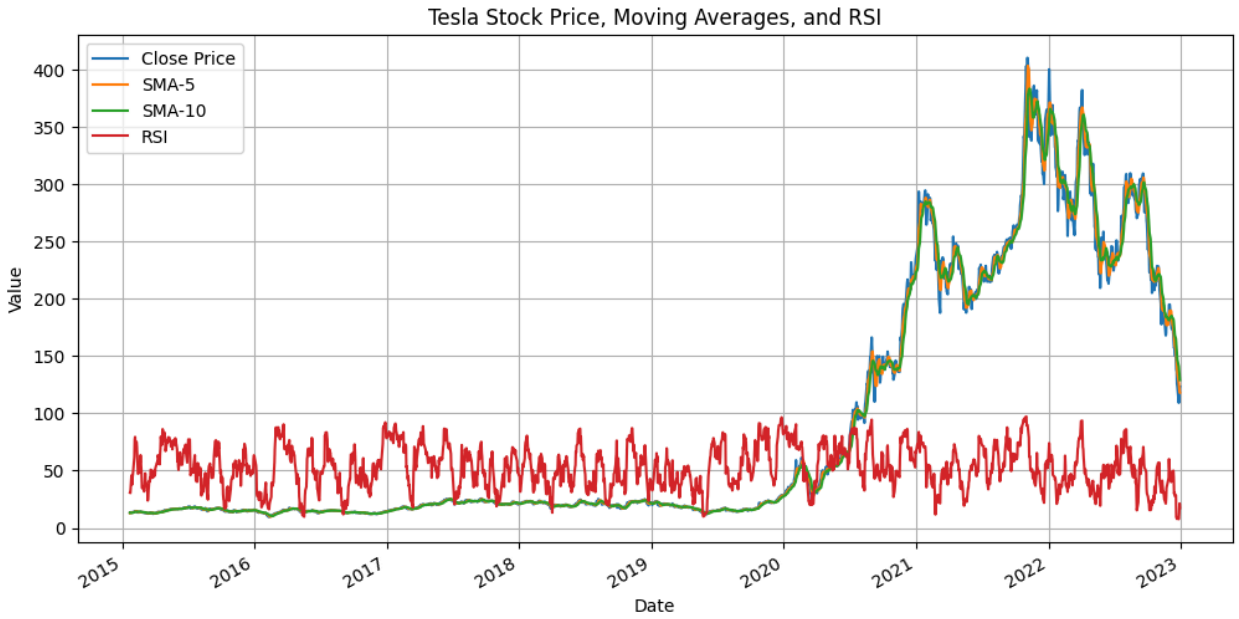



Fig i: Tesla Stock Price, Moving Averages, and Relative Strength Index (RSI)

---

## C. Define the Q-Network

The Q-network approximates Q-values for each state-action pair. The input is the state (features: stock price, SMA-5, SMA-10, RSI), and the output is a vector of Q-values for actions (buy, sell, hold).

We'll use a simple feed-forward neural network for this purpose:

```
class QNetwork(tf.keras.Model):
    def __init__(self, action_space):
        super(QNetwork, self).__init__()
        self.dense1 = layers.Dense(64, activation='relu')
        self.dense2 = layers.Dense(64, activation='relu')
        self.q_values = layers.Dense(action_space, activation='linear')  # Output Q-values for each action

    def call(self, state):
        x = self.dense1(state)
        x = self.dense2(x)
        return self.q_values(x)
```

Here, we define a neural network with two hidden layers and a linear output layer, producing a vector of Q-values for each action.

---

## D. Experience Replay

We'll use a **deque** (a double-ended queue) to store the agent's experiences. Experience replay helps the agent learn by sampling random batches for training.

```
class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def store(self, experience):
        self.buffer.append(experience)

    def sample(self):
        return random.sample(self.buffer, self.batch_size)

    def size(self):
        return len(self.buffer)
```

The ReplayBuffer class stores experiences in a buffer and allows the sampling of random batches.

---

## E. Training the Agent *(this code may take several minutes)*

We can implement the training loop with our Q-network and experience replay ready. We will use the **epsilon-greedy policy** to select actions, store experiences, and update the Q-network based on the rewards.

```
[18] # Parameters
action_space = 3  # Buy, Hold, Sell
state_size = 4    # ['Close', 'SMA-5', 'SMA-10', 'RSI']
buffer_size = 10000
batch_size = 32
learning_rate = 0.001
gamma = 0.99      # Discount factor
epsilon = 1.0     # Initial epsilon for exploration
epsilon_min = 0.01
epsilon_decay = 0.995
episodes = 100

# Initialize Q-network, target network, and replay buffer
q_network = QNetwork(action_space)
target_network = QNetwork(action_space)
target_network.set_weights(q_network.get_weights())  # Sync target network

optimizer = tf.keras.optimizers.Adam(learning_rate)
loss_fn = tf.keras.losses.MeanSquaredError()

replay_buffer = ReplayBuffer(buffer_size, batch_size)

# Epsilon-greedy policy
def select_action(state, epsilon):
    if np.random.rand() <= epsilon:
        return np.random.randint(action_space)  # Random action
    q_values = q_network(np.expand_dims(state, axis=0))  # Predict Q-values
    return np.argmax(q_values.numpy())  # Action with highest Q-value
```

```
[18] # Training loop
for episode in range(episodes):
    state = stock_data.iloc[0][['Close', 'SMA-5', 'SMA-10', 'RSI']].values
    total_reward = 0

    for t in range(len(stock_data) - 1):
        # Select action
        action = select_action(state, epsilon)

        # Simulate environment response
        next_state = stock_data.iloc[t + 1][['Close', 'SMA-5', 'SMA-10', 'RSI']].values
        reward = next_state[0] - state[0]  # Reward is the price change (simple)
        reward = reward if action == 0 else -reward  # Positive reward for correct action

        done = t == len(stock_data) - 2
        total_reward += reward

        # Store experience in replay buffer
        replay_buffer.store((state, action, reward, next_state, done))
        state = next_state

        # Update the Q-network if replay buffer has enough samples
        if replay_buffer.size() >= batch_size:
            # Sample a batch of experiences
            batch = replay_buffer.sample()
            states, actions, rewards, next_states, dones = map(np.array, zip(*batch))

            # Compute target Q-values
            target_q_values = target_network(next_states)
            targets = rewards + gamma * np.max(target_q_values.numpy(), axis=1) * (1 - dones)

            # Update Q-network
            with tf.GradientTape() as tape:
                q_values = q_network(states)
                q_values = tf.reduce_sum(q_values * tf.one_hot(actions, action_space), axis=1)
                loss = loss_fn(targets, q_values)

            grads = tape.gradient(loss, q_network.trainable_variables)
            optimizer.apply_gradients(zip(grads, q_network.trainable_variables))
```

[18]
```
            # Update target network periodically
            if t % 10 == 0:
                target_network.set_weights(q_network.get_weights())

            if done:
                break

        # Decay epsilon
        epsilon = max(epsilon_min, epsilon * epsilon_decay)

        print(f"Episode {episode + 1}/{episodes}, Total Reward: {total_reward:.2f}, Epsilon: {epsilon:.3f}")

    print("Training complete!")
```

```
Episode 44/100, Total Reward: -50.17, Epsilon: 0.802
Episode 45/100, Total Reward: 731.49, Epsilon: 0.798
Episode 46/100, Total Reward: 120.92, Epsilon: 0.794
```

## 5. Evaluation and Testing

After training the Deep Q-Learning agent, it is essential to evaluate its stock trading performance. This section covers testing data, visualizing results, and conducting a backtest to assess the strategy's effectiveness.

---

## A. Visualizing the Agent's Actions

One way to evaluate the agent's behavior is to visualize its actions (Buy, Sell, Hold) on the stock price data. Plotting these actions alongside the stock price helps you see how the agent responds to market conditions.

```
# Test the agent and collect actions
test_data = stock_data.iloc[-100:]  # Use the last 100 rows as test data
state = test_data.iloc[0][['Close', 'SMA-5', 'SMA-10', 'RSI']].values

actions = []  # To store actions
prices = []   # To store stock prices
timestamps = test_data.index  # Dates for plotting

for t in range(len(test_data) - 1):
    action = select_action(state, epsilon=0.01)  # Use low epsilon for exploitation
    actions.append(action)
    prices.append(state[0])  # Save the price for plotting
    state = test_data.iloc[t + 1][['Close', 'SMA-5', 'SMA-10', 'RSI']].values

# Convert actions to markers for visualization
actions = np.array(actions)
buy_signals = np.where(actions == 0)[0]  # Indexes where the agent decided to Buy
sell_signals = np.where(actions == 2)[0]  # Indexes where the agent decided to Sell
hold_signals = np.where(actions == 1)[0]  # Indexes where the agent decided to Hold

# Plot stock prices and agent's actions
plt.figure(figsize=(12, 6))
plt.plot(timestamps[:-1], prices, label='Stock Price', color='blue')

# Add markers for actions
plt.scatter(timestamps[:-1][buy_signals], np.array(prices)[buy_signals],
            label='Buy', marker='^', color='green', alpha=0.8)
plt.scatter(timestamps[:-1][sell_signals], np.array(prices)[sell_signals],
            label='Sell', marker='v', color='red', alpha=0.8)
plt.scatter(timestamps[:-1][hold_signals], np.array(prices)[hold_signals],
            label='Hold', marker='o', color='orange', alpha=0.8)

# Add legend and labels
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.title('Agent Actions on Stock Price')
plt.legend()
plt.show()
```

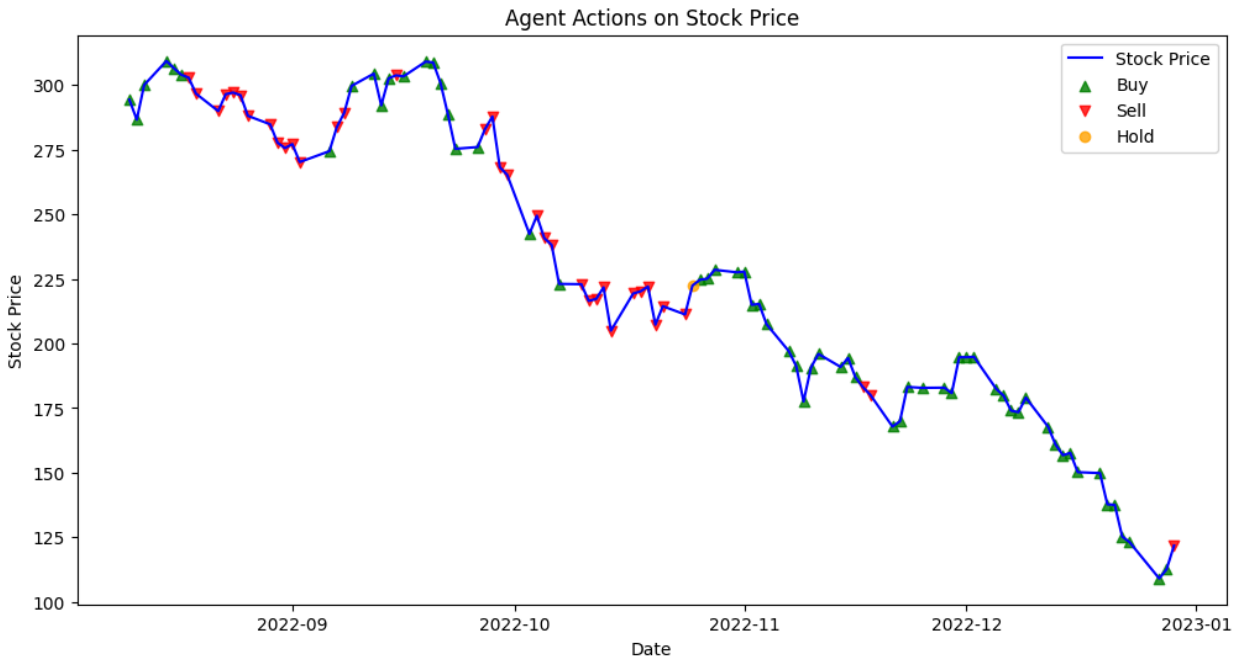



Fig ii: Agent Action on Tesla Stock Price

---

## B. Calculating the Agent's Profit

To evaluate the agent's performance honestly, we need to calculate the profit or return from the trades. Here's how we can do it:

- Start with an initial investment (e.g., $100,000).
- For every **Buy** action, buy as many shares as possible.
- For every **Sell** action, sell all the shares owned.
- **Hold** doesn't change the portfolio.
- Calculate the final balance at the end of the test period.

```
[30] # Parameters for portfolio evaluation
     initial_balance = 100000  # Initial investment amount
     balance = initial_balance
     shares = 0  # Number of shares owned
     portfolio_value = []  # Track portfolio value over time

     # Simulate the agent's actions
     for t in range(len(stock_data) - 1):
         # Get the state and the agent's action
         state = stock_data.iloc[t][['Close', 'SMA-5', 'SMA-10', 'RSI']].values
         action = select_action(state, epsilon=0)  # Use greedy policy (epsilon=0)

         # Get the current stock price (ensure it's a scalar)
         close_price = stock_data.iloc[t]['Close'].item()  # .item() ensures scalar value

         # Perform the chosen action
         if action == 0:  # Buy
             if balance > close_price:  # Ensure there is enough balance
                 shares_to_buy = int(balance // close_price)  # Calculate shares to buy
                 balance -= shares_to_buy * close_price
                 shares += shares_to_buy
         elif action == 2:  # Sell
             if shares > 0:
                 balance += shares * close_price  # Sell all shares
                 shares = 0

         # Track portfolio value (balance + current value of shares owned)
         portfolio_value.append(balance + shares * close_price)
```

```
[30] # Calculate final balance and profit
     final_balance = balance + shares * stock_data.iloc[-1]['Close'].item()
     profit = final_balance - initial_balance

     # Print results
     print(f"Initial Investment: ${initial_balance:.2f}")
     print(f"Final Balance: ${final_balance:.2f}")
     print(f"Profit: ${profit:.2f}")

     # Plot portfolio value over time
     plt.figure(figsize=(12, 6))
     plt.plot(portfolio_value, label='Portfolio Value')
     plt.plot(stock_data['Close'].values, label='Stock Price', alpha=0.5)
     plt.title('Portfolio Value vs. Stock Price')
     plt.xlabel('Time')
     plt.ylabel('Value')
     plt.legend()
     plt.show()
```

This script tracks the agent's balance after each trade, plots the portfolio value over time, and calculates profit as the difference between the initial and final balances.

```
Initial Investment: $100000.00
Final Balance: $4786576.54
Profit: $4686576.54
```

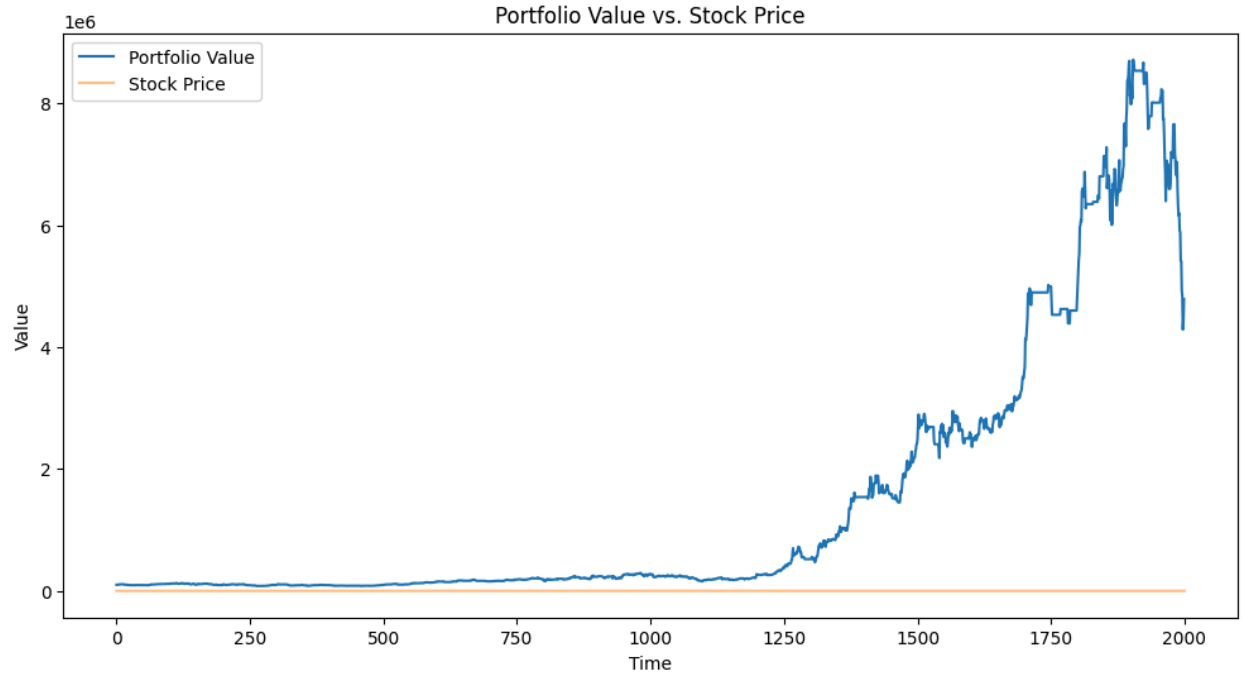



Fig iii: Comparison of Portfolio Value and Tesla Stock Price Over Time

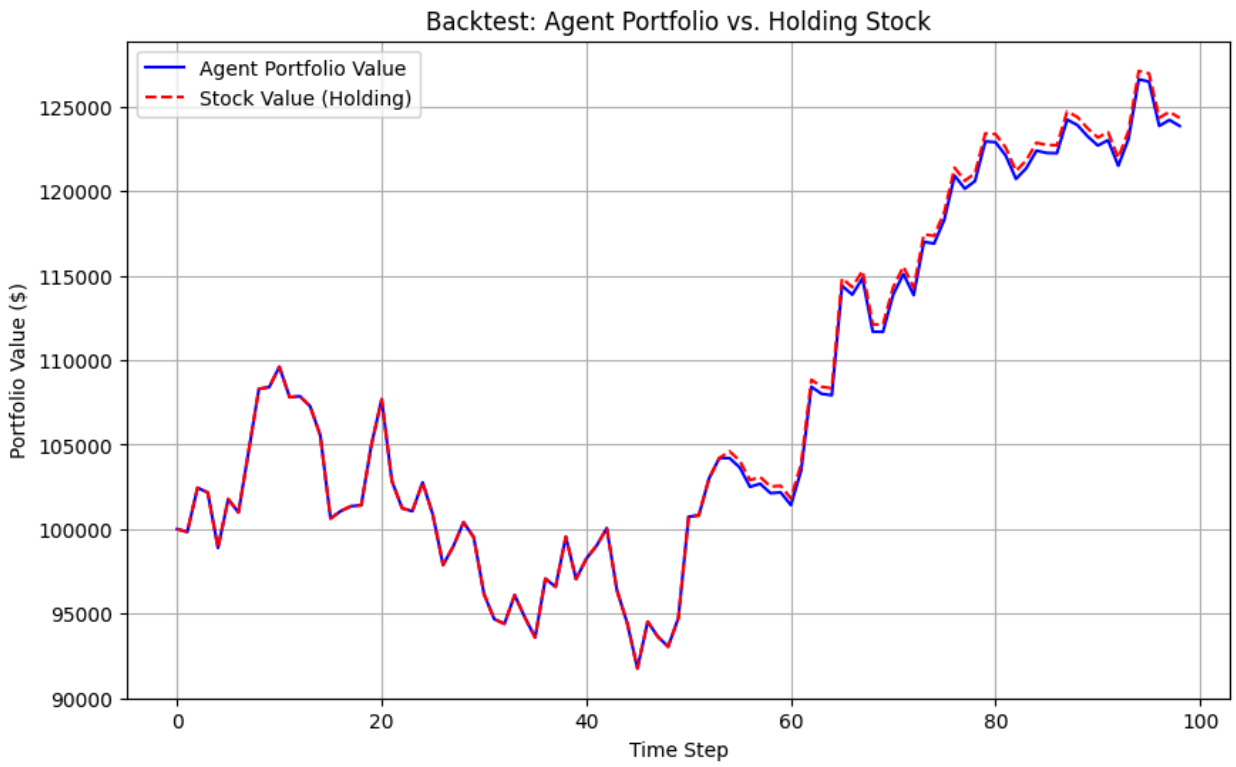
---

## C. Backtesting the Strategy

Backtesting shows how the agent's strategy would have performed in the past, comparing its trades to the stock's performance over the test period based on its learned policy.

```python
# Assuming 'stock_data' is the historical stock data with a 'Close' column
# and 'actions' is the list of actions taken by the agent (0: Buy, 1: Sell)

# Ensure the 'actions' list is of the same length as stock_data or slice accordingly
num_steps = min(len(stock_data), len(actions))

# Initialize variables for backtesting
initial_balance = 100000  # Initial balance in dollars
balance = initial_balance
shares = 0
initial_investment = initial_balance
portfolio_value = []  # Portfolio value over time
stock_value = []  # Stock value if we just hold the stock

# Loop through the stock data to track portfolio and stock performance
for t in range(num_steps):
    close_price = float(stock_data.iloc[t]['Close'].values[0])  # Get the close price for this time step

    # Backtesting agent's actions
    action = actions[t]  # Action taken by the agent at time step t

    # Agent's trading strategy: buy or sell
    if action == 0:  # Buy
        if balance > close_price:  # Ensure there is enough balance to buy at least one share
            shares_to_buy = int(balance // close_price)  # Calculate shares to buy
            balance -= shares_to_buy * close_price  # Deduct the cost from balance
            shares += shares_to_buy  # Increase number of shares owned
    elif action == 1:  # Sell
        if shares > 0:  # Ensure there are shares to sell
            balance += shares * close_price  # Sell all shares and add the proceeds to balance
            shares = 0  # Reset shares to 0 after selling

    # Track portfolio value (balance + value of owned shares)
    portfolio_value.append(balance + shares * close_price)

    # Track stock value if the agent just held the stock (buying at the start and selling at each time step)
    stock_value.append(initial_investment * (close_price / stock_data.iloc[0]['Close']))  # Value if holding stock from the start
```

```python
# Plot the comparison between agent's portfolio and holding the stock
plt.figure(figsize=(10, 6))
plt.plot(portfolio_value, label='Agent Portfolio Value', color='blue')
plt.plot(stock_value, label='Stock Value (Holding)', color='red', linestyle='--')
plt.title('Backtest: Agent Portfolio vs. Holding Stock')
plt.xlabel('Time Step')
plt.ylabel('Portfolio Value ($)')
plt.legend()
plt.grid(True)
plt.show()

# Calculate final performance metrics
final_agent_value = portfolio_value[-1]
final_stock_value = stock_value[-1].item()  # Extract scalar value from the numpy array
print(f"Final Portfolio Value (Agent's Strategy): ${final_agent_value:.2f}")
print(f"Final Stock Value (Holding the Stock): ${final_stock_value:.2f}")
print(f"Profit from Agent's Strategy: ${final_agent_value - initial_balance:.2f}")
print(f"Profit from Holding the Stock: ${final_stock_value - initial_investment:.2f}")
```




```
Final Portfolio Value (Agent's Strategy): $123859.47
Final Stock Value (Holding the Stock): $124337.87
Profit from Agent's Strategy: $23859.47
Profit from Holding the Stock: $24337.87
```

Fig iv: Backtest Results: Agent's Portfolio Value vs. Holding Tesla Stock

The agent's performance is compared to the stock price over time to assess whether the agent’s strategy outperforms just holding the stock.

---

## 6. Conclusion

---

This tutorial highlighted the complexity of applying Deep Q-Learning (DQL) to stock trading, a real-world decision-making task. We employed state-of-the-art techniques like Q-Networks (deep neural networks) to approximate the Q-value function and epsilon-greedy exploration to balance exploration and exploitation during training. Despite the challenges, our results were promising, showing that DQL can effectively balance risk and reward in stock trading. However, there is significant room for improvement, particularly in advanced RL techniques, exploration strategies, and risk management.

To address more practical problems, future tutorials could involve applying the DQN model to real-time trading data and testing it under more challenging market conditions. Additionally, this approach could be extended to other sectors, such as healthcare for personalized treatment plans, robotics for autonomous navigation, and finance for fraud detection. Incorporating sentiment analysis or macroeconomic indicators could enhance the agent's robustness and adaptability. This ongoing development underscores the potential of deep learning to tackle complex problems across various applications.

---

## 7. Bibliography

---


Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38.

Chen, L., & Gao, Q. (2019, October). Application of deep reinforcement learning on automated stock trading. In *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)* (pp. 29-33). IEEE.

Li, Y., Ni, P., & Chang, V. (2020). Application of deep reinforcement learning in stock trading strategies and stock forecasting. *Computing*, *102*(6), 1305-1322.

Massahi, M., & Mahootchi, M. (2024). A deep Q-learning based algorithmic trading system for commodity futures markets. *Expert Systems with Applications*, *237*, 121711.

Pricope, T. V. (2021). Deep reinforcement learning in quantitative algorithmic trading: A review. *arXiv preprint arXiv:2106.00123*.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Théate, T., & Ernst, D. (2021). An application of deep reinforcement learning to algorithmic trading. *Expert Systems with Applications*, *173*, 114632.