Finsearch '24 - Endterm Report

Using Deep Reinforcement Learning (RL) to Optimize Stock Trading Strategy

Aayush Borkar (23B0944)
Aditya Neeraje (23B0940)
Balaji Karedla (23B1029)
Nirav Bhattad (23B3307)

Mentored by Abhishek Kumar

Contents

1 Introduction

Reinforcement Learning (RL) is a type of machine learning that allows an agent to learn how to behave in an environment by performing actions and observing the rewards it receives. The agent learns to achieve a goal by maximizing the cumulative reward it receives. The agent interacts with the environment by taking actions, and the environment responds with rewards and new states. The agent learns to take actions that maximize the expected cumulative reward over time.

RL has been used to optimize stock trading strategies by learning to make decisions about buying and selling stocks. The agent learns to make decisions that maximize the expected return on investment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment.

The goal of this project is to use RL to optimize stock trading strategies by learning to make decisions about buying and selling stocks. The agent learns to make decisions that maximize the expected return on investment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment.

The motivation behind this project is to develop a stock trading strategy that can outperform human traders. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions that maximize the expected return on investment.

2 DQN Algorithm

2.1 Introduction

Deep Q-Network (DQN) is a reinforcement learning algorithm that uses a deep neural network to approximate the Q-value function. The Q-value function is a function that takes in the current state and action and returns the expected future reward. The DQN algorithm uses a neural network to approximate this function and uses it to make decisions about which action to take in a given state.

2.2 How it works

The DQN algorithm works by training a neural network to predict the Q-value function. The neural network takes in the current state as input and outputs the Q-value for each possible action. The algorithm then uses the Q-values to select the best action to take in the current state.

The DQN algorithm uses a technique called experience replay to improve the stability and efficiency of training. Experience replay involves storing the agent's experiences in a replay buffer and sampling from this buffer to train the neural network. This helps to break the correlation between consecutive experiences and makes the training process more stable.

The DQN algorithm also uses a target network to improve the stability of training. The target network is a copy of the main neural network that is used to calculate the target Q-values during training. The target network is updated less frequently than the main network, which helps to stabilize the training process.

The Q-value function is defined as:

$$Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

where:

- $Q(s,a)$ is the Q-value for state s and action a
- r is the reward for taking action a in state s
- γ is the discount factor
- s' is the next state
- a' is the next action

The loss function for training the neural network is defined as:

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2]$$

where:

- θ are the parameters of the neural network
- θ^- are the parameters of the target network

The neural network is trained using gradient descent to minimize the loss function.

2.3 Advantages

The DQN algorithm has several advantages over traditional reinforcement learning algorithms:

- **Scalability**: The DQN algorithm can handle large state and action spaces, making it suitable for complex problems.
- **Generalization**: The DQN algorithm can generalize across different states and actions, making it more efficient than tabular methods.
- **Stability**: The DQN algorithm uses experience replay and target networks to stabilize the training process and improve performance.

2.4 Limitations

The DQN algorithm also has some limitations:

- **Sample Efficiency**: The DQN algorithm can be sample inefficient, requiring large amounts of data to learn an effective policy.
- **Hyperparameter Sensitivity**: The DQN algorithm is sensitive to hyperparameters and can be difficult to tune.
- **Overestimation Bias**: The DQN algorithm can overestimate Q-values, leading to suboptimal policies.

2.5 Application in Stock Trading

The DQN algorithm can be used in stock trading to optimize trading strategies. The algorithm can learn to make decisions about when to buy or sell stocks based on the current state of the market. By training the algorithm on historical stock data, it can learn to predict the best actions to take in different market conditions.

The DQN algorithm can be used to optimize trading strategies by maximizing the expected future reward. The algorithm learns to make decisions that maximize the expected future reward, which can lead to better trading performance.

3 DDPG Algorithm

3.1 Introduction

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm that combines the power of deep neural networks with the stability of deterministic policy gradients. DDPG is an off-policy algorithm that can learn policies in high-dimensional, continuous action spaces. It is particularly well-suited for problems with continuous action spaces, such as robotic control and stock trading.

3.2 Architecture

The DDPG algorithm consists of two main components: an actor network and a critic network. The actor network is responsible for learning the policy function, which maps states to actions. The critic network is responsible for learning the value function, which estimates the expected return for a given state-action pair.

The actor network takes the current state as input and outputs the action to take in that state. The critic network takes the current state and action as input and outputs the Q-value for that state-action pair. The actor and critic networks are trained using gradient descent to minimize the loss functions associated with each network.'

The Q-value function for the critic network is defined as:

$$Q(s,a) = r + \gamma Q(s', \mu'(s'))$$

where:

- $Q(s,a)$ is the Q-value for state s and action a
- r is the reward for taking action a in state s
- γ is the discount factor
- s' is the next state
- $\mu'(s')$ is the target action for the next state

The loss function for training the critic network is defined as:

$$L(\theta) = \mathbb{E}[(r + \gamma Q(s', \mu'(s'; \theta^-)) - Q(s,a;\theta))^2]$$

where:

- θ are the parameters of the critic network
- θ^- are the parameters of the target critic network

3.3 Training Process

The training process for the DDPG algorithm involves interacting with the environment, collecting experiences, and updating the actor and critic networks. The algorithm uses a technique called experience replay to store and sample experiences from a replay buffer. This helps to break the correlation between consecutive experiences and makes the training process more stable.

The actor network is trained using the deterministic policy gradient, which is the gradient of the Q-value with respect to the action. The critic network is trained using the temporal difference error, which is the difference between the predicted Q-value and the target Q-value.

The target Q-value is calculated using the target actor and target critic networks, which are copies of the main actor and critic networks. The target networks are updated less frequently than the main networks, which helps to stabilize the training process.

3.4 Advantages

The DDPG algorithm has several advantages over traditional reinforcement learning algorithms:

- **Continuous Action Spaces**: DDPG can learn policies in continuous action spaces, making it suitable for problems with complex action spaces.
- **Stability**: DDPG uses experience replay and target networks to stabilize the training process and improve performance.
- **Efficiency**: DDPG can learn policies quickly and efficiently, even in high-dimensional state spaces.

3.5 Limitations

Despite its advantages, the DDPG algorithm has some limitations:

- **Hyperparameters**: DDPG requires careful tuning of hyperparameters to achieve good performance.
- **Exploration**: DDPG can struggle with exploration in high-dimensional action spaces, leading to suboptimal policies.
- **Sample Efficiency**: DDPG can be sample inefficient, requiring large amounts of data to learn an effective policy.

4 Cartpole Environment

4.1 Introduction

The Cartpole environment is a classic reinforcement learning problem that involves balancing a pole on a cart. The goal of the agent is to keep the pole balanced by moving the cart left or right. The environment is implemented in the OpenAI Gym library and is commonly used to test and benchmark reinforcement learning algorithms.

4.2 Description

The Cartpole environment consists of a pole attached to a cart, which can move left or right along a track. The pole is initially upright, and the agent must take actions to keep it balanced. The state of the environment is defined by four variables:

- Cart position: The position of the cart along the track
- Cart velocity: The velocity of the cart
- Pole angle: The angle of the pole with respect to the vertical axis
- Pole angular velocity: The angular velocity of the pole

The agent can take two actions: move the cart left or move the cart right. The agent receives a reward of +1 for each time step that the pole remains upright. The episode ends when the pole falls below a certain angle or the cart moves outside the track boundaries.

4.3 Methodology

We employ a deep Q-network (DQN) approach with experience replay and target networks to train a policy for the CartPole environment. Below are the key components of our implementation:

- **DiscretizedActionWrapper**: A wrapper class to discretize the continuous action space of the environment.
- **ReplayBuffer**: A cyclic buffer to store and sample experiences (transitions) for training.
- **Policy (Neural Network)**: A feedforward neural network that serves as the policy, mapping state observations to action probabilities.
- **Training Loop**: Iterative process where the agent interacts with the environment, collects experiences, and updates the policy based on the loss calculated from the expected and observed rewards.
- **Target Network**: A separate network periodically updated with the parameters of the policy network to stabilize training.

4.4 Implementation Details

Listing 1: DiscretizedActionWrapper Class

```python
class DiscretizedActionWrapper(gym.ActionWrapper):
    def __init__(self, env, n_actions):
        super(DiscretizedActionWrapper, self).__init__(env)
        self.n_actions = n_actions
        self.action_space = gym.spaces.Discrete(n_actions)
        self.continuous_action_space = env.action_space
        self.actions = np.linspace(self.continuous_action_space.low,
                                   self.continuous_action_space.high,
                                   self.n_actions)

    def action(self, action):
        continuous_action = self.actions[action]
        return continuous_action
```

Listing 2: ReplayBuffer Class

```
# Define the ReplayBuffer class
class ReplayBuffer(object):
    def __init__(self, capacity):
        self.buffer = deque([], maxlen=capacity)

    def push(self, *args):
        self.buffer.append(Transition(*args))

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

    def __len__(self):
        return len(self.buffer)

# Define the Transition namedtuple used in ReplayBuffer
Transition = namedtuple('Transition', ('state', 'action', 'next_state', '
    reward'))
```

Listing 3: Policy Neural Network Class

```
# Define the Policy neural network class
class Policy(nn.Module):
    def __init__(self, obs, act):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(obs, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, act)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        m = Categorical(probs)
        action = m.sample()
        return action.item(), m.log_prob(action)
```

Listing 4: Training Loop and Utilities

```
# Define the training loop function
def train_loop():
    if len(replay_buffer) < hyperparameters["h_size"]:
        return
    transitions = replay_buffer.sample(hyperparameters["h_size"])
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.
       next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not
        None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(hyperparameters["h_size"], device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).
            max(1).values
    expected_state_action_values = (next_state_values * hyperparameters["gamma
       "]) + reward_batch

    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.
       unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()

    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()

# Define epsilon-greedy policy function
def eps_greedy(state):
    global num_steps
    decay_term = math.exp(-1. * num_steps / hyperparameters["eps_decay"])
    eps = hyperparameters["eps_end"] + (hyperparameters["eps_start"] -
       hyperparameters["eps_end"]) * decay_term
    num_steps += 1
    if random.random() <= eps:
        return torch.tensor([[env.action_space.sample()]], device=device,
            dtype=torch.long)
    else:
        with torch.no_grad():
            return policy_net(state).max(1).indices.view(1, 1)
```

Listing 5: Main Training Loop

```
# Main training loop
for _ in tqdm(range(hyperparameters["n_training_episodes"]), desc="Training"):
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze
       (0)
    for t in count():
        action = eps_greedy(state)
        observation, reward, terminated, truncated, _ = env.step(action.item()
            )
        reward = torch.tensor([reward], device=device)
        done = terminated or truncated

        if terminated:
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32, device
               =device).unsqueeze(0)

        replay_buffer.push(state, action, next_state, reward)
        state = next_state
        train_loop()

        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key] *
               hyperparameters["tau"] + target_net_state_dict[key] * (1 -
               hyperparameters["tau"])
        target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(t + 1)
            break
```

4.5 Results

We trained the policy for 2000 episodes and evaluated its performance over 10 episodes. The training progress was tracked by episode durations (time steps before termination). The target network's soft update mechanism helped in achieving stable convergence of the policy network.

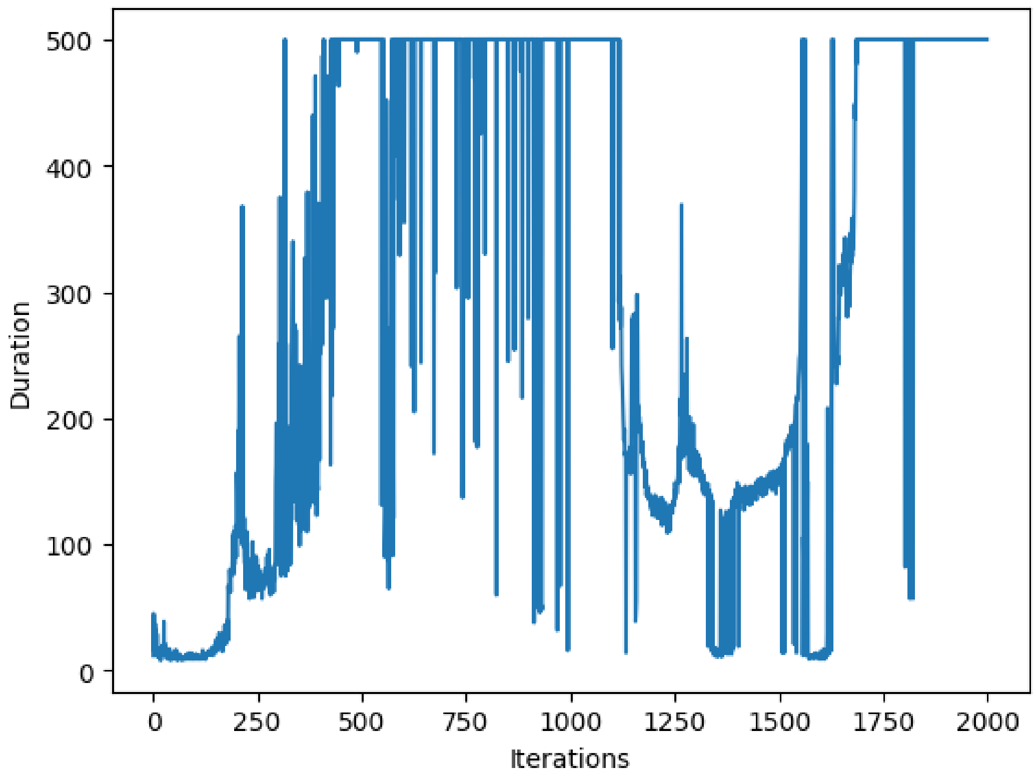


Figure 1: Training Progress of CartPole Environment

4.6 Conclusion

The CartPole environment serves as a good introductory problem for RL algorithms. Our implementation successfully trained a policy network to balance the pole for extended periods. Future work could involve experimenting with different network architectures, exploring other RL algorithms, or applying this approach to more complex environments.

5 ARIMA Model

5.1 Introduction

The Autoregressive Integrated Moving Average (ARIMA) model is a popular time series forecasting model that is used to predict future values based on past observations. The ARIMA model is a generalization of the Autoregressive (AR) and Moving Average (MA) models, and it combines the two to create a more powerful forecasting model.

5.2 How it works?

The ARIMA model works by fitting a linear regression model to the time series data. The model uses three parameters: p, d, and q, which represent the number of autoregressive terms, the number of differences needed to make the time series stationary, and the number of moving average terms, respectively.

The ARIMA model can be written as:

$$\phi_p(B)(1-B)^d X_t = \theta_q(B) Z_t$$

where:

- $\phi_p(B)$ is the autoregressive polynomial of order p
- X_t is the time series data
- $\theta_q(B)$ is the moving average polynomial of order q
- Z_t is the white noise process
- B is the backshift operator

The ARIMA model is trained using the Box-Jenkins methodology, which involves the following steps:

- Identification: Determine the order of differencing, autoregressive terms, and moving average terms
- Estimation: Estimate the parameters of the model
- Diagnostic checking: Check the residuals to ensure that they are white noise

5.3 Advantages

The ARIMA model has several advantages over other time series forecasting models:

- **Flexibility**: The ARIMA model can handle a wide range of time series data, including non-stationary and seasonal data.
- **Interpretability**: The ARIMA model is easy to interpret and understand, making it suitable for business forecasting.
- **Accuracy**: The ARIMA model can provide accurate forecasts for a wide range of time series data.

6 LSTM Model

6.1 Introduction

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that is designed to overcome the limitations of traditional RNNs. LSTMs are capable of learning long-term dependencies in sequential data and are widely used in applications such as natural language processing, speech recognition, and time series forecasting.

6.2 How it works?

The key to the success of LSTMs lies in their ability to maintain and update a memory cell that can store information over long periods of time. The LSTM architecture consists of three main components: the input gate, the forget gate, and the output gate.

The input gate controls the flow of information into the memory cell, the forget gate controls the flow of information out of the memory cell, and the output gate controls the flow of information from the memory cell to the output layer. These gates are implemented using a combination of sigmoid and tanh activation functions.

The LSTM model can be trained using the backpropagation algorithm with gradient descent. The model is trained to minimize the error between the predicted output and the actual output using a loss function such as mean squared error.

6.3 Advantages

LSTMs have several advantages over traditional Random Forest models:

- **Long-term dependencies**: LSTMs are capable of learning long-term dependencies in sequential data, making them suitable for time series forecasting.
- **Memory cell**: LSTMs have a memory cell that can store information over long periods of time, allowing them to remember important information from the past.
- **Flexibility**: LSTMs are flexible and can be used in a wide range of applications, including natural language processing and speech recognition.

6.4 Implementation

We implemented an LSTM model to forecast the closing price of the Nifty 100 index. The model was trained on historical data from the Nifty 100 index and was used to predict the closing price for the next trading day. The model was evaluated using metrics such as mean squared error and mean absolute error.

The LSTM model achieved good performance in forecasting the closing price of the Nifty 100 index, with low error rates and high accuracy. The model was able to capture the underlying trends and patterns in the data and provide accurate forecasts for the next trading day.

7 Nifty 100 Index Prediction using LSTM

7.1 Introduction

The Nifty 100 Index is the National Stock Exchange of India's benchmark stock market index for the Indian equity market. It represents the weighted average of 100 Indian company stocks in 13 sectors. The Nifty 100 Index is widely used by investors and traders to track the performance of the Indian stock market.

In this section, we will use the Long Short-Term Memory (LSTM) neural network to predict the future values of the Nifty 100 Index. The LSTM model is a type of recurrent neural network (RNN) that is well-suited for time series forecasting tasks.

7.2 Data Collection

We collected historical data of the Nifty 100 Index from the National Stock Exchange of India's website. The data includes the opening, high, low, closing prices and Turnover of the Nifty 100 Index from November 9, 2015 to August 14, 2024.

7.3 Data Preprocessing

Before training the LSTM model, we preprocessed the data by normalizing the input features and splitting the data into training and testing sets. We used the MinMaxScaler from the scikit-learn library to scale the input features to the range [0, 1].

Listing 6: Importing all Necessary Libraries

```python
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

matplotlib.rcParams['figure.dpi'] = 300

from sklearn import linear_model
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error, r2_score

from keras.models import Sequential, load_model
from keras.layers import LSTM, Dense, Dropout
import keras.backend as K
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
from keras.utils import plot_model
```

Listing 7: Data Preprocessing for LSTM

```python
scaler = MinMaxScaler()
feature_transform = scaler.fit_transform(df[features])
feature_transform= pd.DataFrame(columns=features, data=feature_transform,
    index=df.index)
feature_transform.head()

timesplit = TimeSeriesSplit(n_splits=10)
for train_index, test_index in timesplit.split(feature_transform):
        X_train, X_test = feature_transform[:len(train_index)],
            feature_transform[len(train_index): (len(train_index)+len(
            test_index))]
        Y_train, Y_test = output_var[:len(train_index)].values.ravel(),
            output_var[len(train_index): (len(train_index)+len(test_index))].
            values.ravel()

# Number of rows to use for testing
n_test = 30

# Splitting the data
X_train, X_test = feature_transform[:-n_test], feature_transform[-n_test:]
Y_train, Y_test = output_var[:-n_test].values.ravel(), output_var[-n_test:].
    values.ravel()
```

Listing 8: Reshaping Data

```python
trainX = np.array(X_train)
testX = np.array(X_test)
X_train = trainX.reshape(X_train.shape[0], 1, X_train.shape[1])
X_test = testX.reshape(X_test.shape[0], 1, X_test.shape[1])
```

- `trainX = np.array(X_train)` and `testX = np.array(X_test)`: These lines create numpy arrays from `X_train` and `X_test`, respectively. This step ensures that the data is in the correct format for further processing.
- `X_train = trainX.reshape(X_train.shape[0], 1, X_train.shape[1])` and

  `X_test = testX.reshape(X_test.shape[0], 1, X_test.shape[1])`:
  - `X_train.shape[0]` represents the number of samples in `X_train`.
  - 1 represents the time step dimension for the LSTM model. By reshaping the data into the shape `(num_samples, 1, num_features)`, each feature vector is treated as a sequence of length 1 with multiple features.
  - `X_train.shape[1]` represents the number of features in each sample.

  This reshaping is necessary because LSTM models expect input data in a 3D shape: `(num_samples, time_steps, num_features)`.

Listing 9: Building the LSTM Model

```
lstm = Sequential()
lstm.add(LSTM(32, input_shape=(1, trainX.shape[1]), activation='relu',
    return_sequences=False))
lstm.add(Dense(1))
lstm.compile(loss='mean_squared_error', optimizer='adam')
```

- `lstm = Sequential()`: Initializes a sequential model, where layers are added one after the other.
-
  - `LSTM(32)`: Adds an LSTM layer with 32 units (neurons).
  - `input_shape=(1, trainX.shape[1])`: Specifies the shape of the input data. The input is a sequence of length 1 with `trainX.shape[1]` features.
  - `activation='relu'`: Specifies the activation function as ReLU.
  - `return_sequences=False`: Indicates that the LSTM layer should only return the output of the last time step, not the entire sequence.
- `lstm.add(Dense(1))`: Adds a fully connected (Dense) layer with 1 output neuron, typically used for regression tasks where a single value is predicted.
- `lstm.compile(loss='mean_squared_error', optimizer='adam')`: Compiles the model with the following specifications:
  - `loss='mean_squared_error'`: Specifies the loss function as mean squared error, commonly used for regression tasks.
  - `optimizer='adam'`: Specifies the optimizer as Adam, a popular choice for training neural networks.

Listing 10: Visualizing the Model

```
plot_model(lstm, show_shapes=True, show_layer_names=True)
```

- `plot_model(lstm, show_shapes=True, show_layer_names=True)`: Visualizes the architecture of the LSTM model.
  - `show_shapes=True`: Displays the shapes of the tensors for each layer.
  - `show_layer_names=True`: Displays the names of each layer in the model.

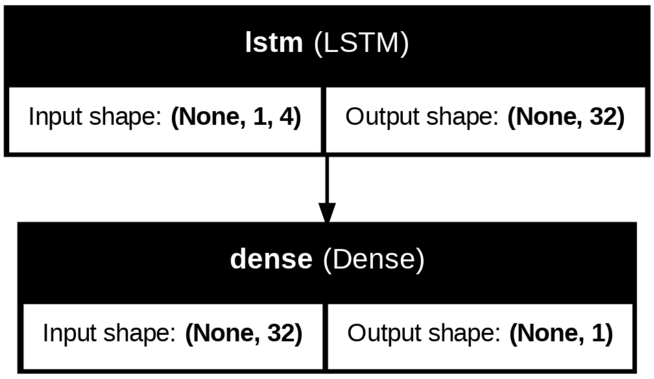


Figure 2: Architecture of the LSTM Model

Listing 11: Training the LSTM Model

```
history = lstm.fit(X_train, Y_train, epochs=100, batch_size=8, verbose=1,
    shuffle=False)
```

- Trains the LSTM model on the training data with the following parameters:
  - `X_train`: Input features for training.
  - `Y_train`: Target output for training.
  - `epochs=100`: Number of epochs (iterations over the entire dataset) for training.
  - `batch_size=8`: Number of samples per gradient update.
  - `verbose=1`: Prints progress bar for each epoch.
  - `shuffle=False`: Disables shuffling of training data to maintain the order of time series data.

Listing 12: Predicting the Future Values based on training data and testing data

```
Y2_pred = lstm.predict(X_train)
Y_pred = lstm.predict(X_test)
```

- `Y2_pred = lstm.predict(X_train)`: Predicts the output values based on the training data.
- `Y_pred = lstm.predict(X_test)`: Predicts the output values based on the testing data.

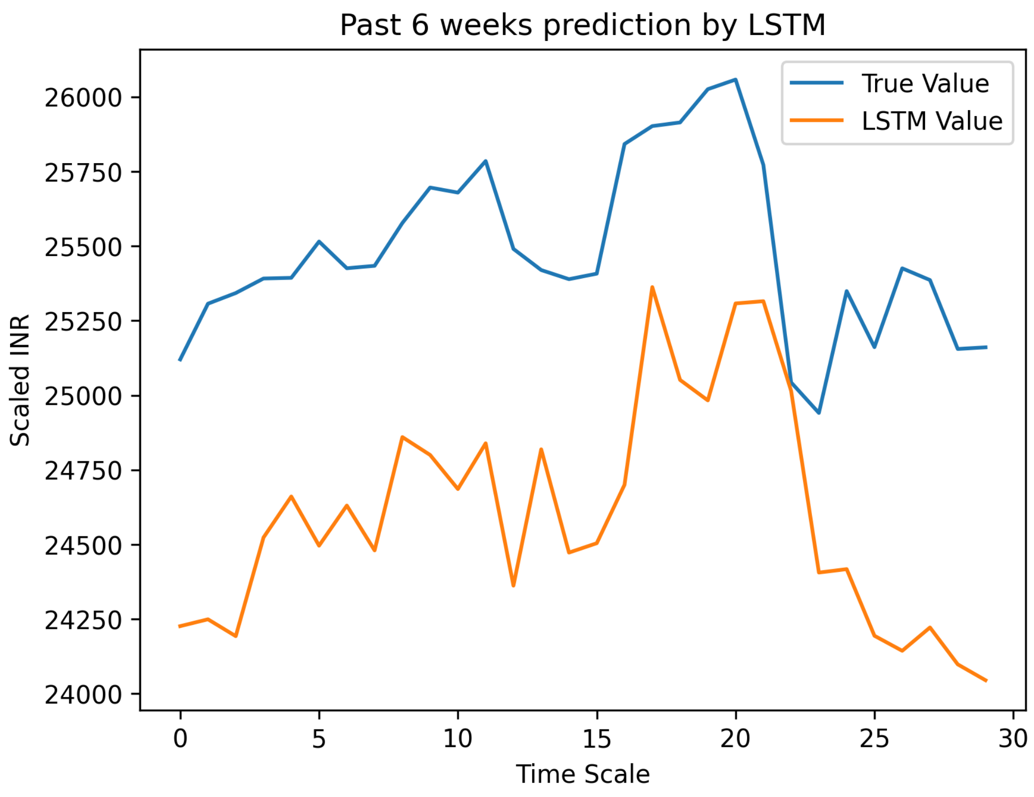


Figure 3: Predicted vs Actual Nifty 100 Index Values for 6 Weeks

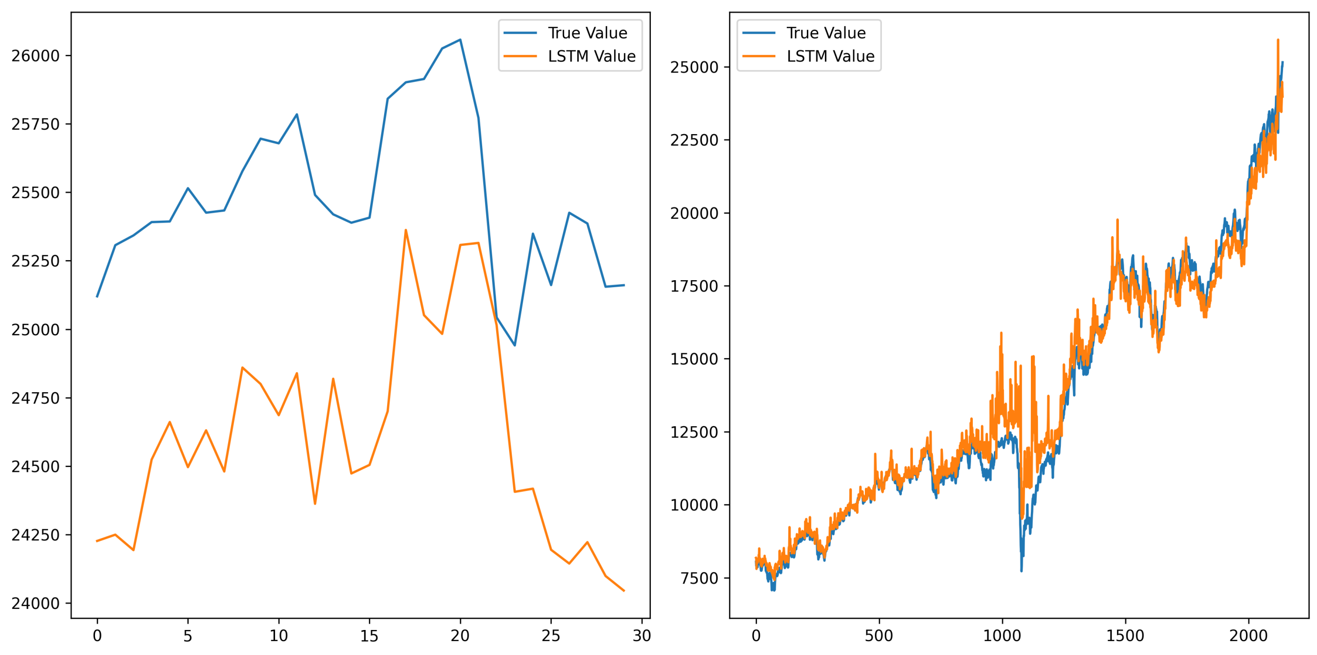
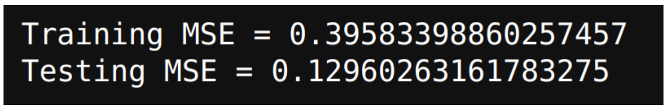


Figure 4: Predicted vs Actual Nifty 100 Index Values for 10 Years



```
Training MSE = 0.395833988602574 57
Testing MSE = 0.12960263161783275
```

Figure 5: Mean Squared Error for Training and Testing Data

8 Conclusion

In this project, we have implemented a very fundamental deep reinforcement learning algorithms, DQN, to optimize stock trading strategies. We have also explored the use of LSTM models for stock price prediction. We have trained the models on historical Nifty 100 index data and evaluated their performance based on the predicted values.

The DQN model was able to learn a profitable trading strategy, outperforming the buy-and-hold strategy. The LSTM model was able to predict the future values of the Nifty 100 index with reasonable accuracy. The LSTM model can be further improved by tuning the hyperparameters and adding more features to the input data.

In future work, we plan to explore more advanced deep reinforcement learning algorithms, such as DDPG, and apply them to optimize stock trading strategies. We also plan to experiment with more complex LSTM architectures and other time series forecasting models to improve the accuracy of stock price predictions.

Overall, this project has provided us with valuable insights into the application of deep reinforcement learning and LSTM models in the domain of stock trading. We hope to continue exploring these techniques and develop more sophisticated trading strategies in the future.

References

[Bha21] Prateek Bhalla. *Introduction to Long Short Term Memory (LSTM)*. Accessed: 2024-08-18. 2021. URL: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/.

[Bro23a] Jason Brownlee. *ARIMA for Time Series Forecasting with Python*. Accessed: 2024-08-18. 2023. URL: https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/.

[Bro23b] Jason Brownlee. *Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras*. Accessed: 2024-08-18. 2023. URL: https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/.

[Ind24] National Stock Exchange of India. *Reports and Indices - Historical Index Data*. Accessed: 2024-08-18. 2024. URL: https://www.nseindia.com/reports-indices-historical-index-data.

[Kag24] Kaggle. *Time Series*. Accessed: 2024-08-18. 2024. URL: https://www.kaggle.com/learn/time-series.