**Deep Reinforcement Learning**

**Assignment - Design**

**Amruth A - BL.EN.U4AIE20002**

**Ramanan R - BL.EN.U4AIE20052**

**Rhea Paul - BL.EN.U4AIE20053**

**Vimal C - BL.EN.U4AIE20073**

*Abstract*: In this paper, the use of deep reinforcement learning in the stock market area is investigated. In particular, a Deep Q Network with Double DQN algorithm is used to train an agent to make trading decisions and select trading strategies using stock price information acquired from the Alpha vantage API. The agent's policy eventually develops the ability to choose activities that produce favorable rewards, leading to successful trading outcomes. The generated earnings, which show the trained model's capacity to make profitable trading decisions, are used to gauge its effectiveness. Examining the effect of increasing the number of training episodes on the profit/loss results, it is shown that there aren't many differences after a certain point. Furthermore, different reward functions that take into account risk and transaction costs, the use of advanced deep learning architectures, and the inclusion of new features to the agent's state representation are all possible future areas for this research.

DRL formulation is already done in the previous assignments; this assignment is mainly focused on the 7 decision points, The table depicts a higher level abstraction. Further details can be found in the following sections.

| **Decision Point** | **Decision Made** |
| --- | --- |
| Decision 1 | Approximate the action-value function Q(s,a;θ) |
| Decision 2 | LSTM layers followed by Dense layers |
| Decision 3 | Minimize loss between predicted and target Q-values |
| Decision 4 | Use Temporal-Difference (TD) target for policy evaluation |
| Decision 5 | Epsilon-Greedy Strategy with 20% chance of random action |
| Decision 6 | Use Mean Squared Error (MSE) as loss function |
| Decision 7 | Mini-batch Gradient Descent with Adam Optimizer |

**Decision Point 1: Selecting a Value Function to Approximate**

**Decision Made**: We chose to approximate the action-value function Q(s,a;θ)

**Justification**: Approximating the action-value function allows the agent to learn the expected return for taking a particular action a*a* in a given state s*s*. This is particularly useful in stock trading where the agent must decide which trading strategy to use in different market conditions.

**Evidence from Code**: In the **DDQNAgent** class, the **act** method predicts Q-values for each action in a given state, which suggests that the action-value function Q(s,a;θ) is being approximated.

def act(self, state):

if np.random.rand() <= self.epsilon:

return np.random.choice(self.action\_size)

state = np.reshape(state, (1, self.state\_size, 1))

q\_values = self.model.predict(state)

return np.argmax(q\_values[0])

**Decision Point 2: Selecting a Neural Network Architecture**

**Decision Made**: The architecture comprises LSTM layers followed by Dense layers. State in value out architecture.

**Justification**: LSTM layers are effective for sequence-based data, such as time-series stock prices. They capture the temporal dependencies which are crucial for making trading decisions. The Dense layers help in the approximation of the action-value function.

**Evidence from Code**: The **build\_model** method in the **DDQNAgent** class reveals the architecture used for approximating the action-value function. The use of LSTM layers is particularly suitable for sequence-based data like stock prices. Aka time series data

def build\_model(self):

model = tf.keras.Sequential()

model.add(tf.keras.layers.LSTM(32, input\_shape=(self.state\_size, 1), return\_sequences=True))

model.add(tf.keras.layers.LSTM(64, return\_sequences=False))

model.add(tf.keras.layers.Dense(32, activation='relu'))

model.add(tf.keras.layers.Dense(self.action\_size, activation='linear'))

model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(lr=self.learning\_rate))

return model

**Decision Point 3: Selecting What to Optimize**

**Decision Made**: The optimization goal is to minimize the loss between the predicted Q-values and the target Q-values.

**Justification**: By minimizing this loss, the agent learns to make better approximations of the action-value function, which is essential for making more accurate trading decisions.

**Evidence from Code**: In the **replay** method, the target is computed using the equation Target=r+γmaxQ(s′,a′;θ)

targets[np.arange(len(targets)), actions] = rewards + self.gamma \* np.amax(next\_q\_values, axis=1) \* ~dones

**Decision Point 4: Selecting the Targets for Policy Evaluation**

**Decision Made**: Temporal-Difference (TD) target is used for policy evaluation, specifically an off-policy TD target.

**Justification**: TD methods are well-suited for problems where the full model of the environment is unknown and only samples are available, which is often the case in stock trading.

**Evidence from Code**: The same **replay** method indicates the use of TD target for policy evaluation. The target value for each state-action pair is updated based on the maximum Q-value for the next state, which is characteristic of off-policy learning.

targets[np.arange(len(targets)), actions] = rewards + self.gamma \* np.amax(next\_q\_values, axis=1) \* ~dones

**Decision Point 5: Selecting an Exploration Strategy**

**Decision Made**: Epsilon-Greedy Strategy with a 20% chance of random action during training.

**Justification**: The epsilon-greedy strategy ensures sufficient exploration of the state space while allowing the agent to exploit its current knowledge most of the time. This balance is crucial for the agent to discover profitable trading strategies.

**Evidence from Code**: The **act** method in the **DDQNAgent** class employs an epsilon-greedy strategy. The agent chooses a random action with a probability of **epsilon** (set to 0.2) and the action that maximizes the Q-value with a probability of 1−ϵ.

def act(self, state):

if np.random.rand() <= self.epsilon:

return np.random.choice(self.action\_size)

state = np.reshape(state, (1, self.state\_size, 1))

q\_values = self.model.predict(state)

return np.argmax(q\_values[0])

**Decision Point 6: Selecting a Loss Function**

**Decision Made**: Mean Squared Error (MSE)

**Justification**: MSE is a commonly used loss function for regression problems. It effectively penalizes large errors in prediction, which is desirable for our stock trading application.

**Evidence from Code**: The model is compiled with the MSE loss function. This measures the average of the squares of the differences between the estimated and target Q-values.

Def build\_model(self):

model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(lr=self.learning\_rate))

**Decision Point 7: Selecting an Optimization Method**

**Decision Made**: Mini-batch Gradient Descent with Adam Optimizer

**Justification**: The Adam optimizer combines the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp. In the fast-paced and volatile stock market, using a robust optimization algorithm like Adam can be beneficial.

**Evidence from Code**: The **replay** method samples a mini-batch from the memory and fits the model on this mini-batch. The Adam optimizer is used for this purpose, as specified in the **build\_model** method.

def replay(self, batch\_size):

minibatch = random.sample(self.memory, batch\_size)

#... (Rest of the code)

self.model.fit(states, targets, epochs=1, verbose=0)

def build\_model(self):

model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(lr=self.learning\_rate))

--------------------------------------------------------------------END--------------------------------------------------------------------