REINFORCEMENT LEARNING

LAB MANUAL

Department Of Computer Science Engineering

Name - Dev Sanghvi

Enrollment No - 211310142006

Class - CSE (AI-ML)

Batch - A1

Practical 1

❖ Using Q-Learning

Q-learning is a reinforcement learning technique that operates without a model. It assists an agent in determining the best actions to take in a specific state, with the goal of maximizing cumulative rewards over time.

It employs a Q-table, where rows represent states and columns correspond to actions. Each value in the table reflects the expected reward for choosing a specific action in a given state.

The objective is to find the optimal Q-values (action quality) for each state-action combination.

Formula:

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \cdot \max Q(s',a') - Q(s,a)\right]$$

Where `α` is the learning rate, `γ` is the discount factor, `r` is the reward, and `s'` is the next state.

Code:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from termcolor import colored
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    classification_report,
)
from sklearn.metrics import mean_squared_error

# Preprocessing steps (retaining loan_status but not using it until the last part)
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Keep the 'loan_status' column for later
comparison
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
```

```
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Discretize loan_amount (actions) into bins (for Q-Learning)
n_actions = 10  # Define the number of discrete actions (loan amount bins)
loan_amount_bins = np.linspace(min(y_train), max(y_train), n_actions + 1)
y_train_discretized = (
    np.digitize(y_train, loan_amount_bins) - 1
)  # Discretize the loan amount

# Initialize Q-table (states = feature combinations, actions = discretized loan
amounts)
n_states = X_train.shape[1]
Q_table = np.zeros((n_states, n_actions))

# Q-learning parameters
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.3  # Exploration-exploitation tradeoff
n_episodes = 100  # Number of episodes

# Q-learning algorithm
for episode in range(n_episodes):
    for i, state in enumerate(X_train):  # Loop through each data point as a state
        if np.random.uniform(0, 1) < epsilon:  # Exploration
            action = np.random.randint(0, n_actions)
        else:  # Exploitation
            state_index = np.digitize(state, bins=np.linspace(-3, 3, n_states)) - 1
            action = np.argmax(Q_table[state_index])

        # Find the next state and reward (predicted loan amount vs actual)
        reward = -np.abs(
            loan_amount_bins[action] - y_train.iloc[i]
        )  # Reward is negative error

        # Update Q-value using the Q-learning update rule
```

```
        next_state = X_train[i]
        Q_table[:, action] = Q_table[:, action] + alpha * (
            reward + gamma * np.max(Q_table[:, action]) - Q_table[:, action]
        )

# Testing
y_pred_discretized = []
for state in X_test:
    action = np.argmax(Q_table[:, np.random.randint(0, n_states)])
    predicted_loan_amount = loan_amount_bins[action]
    y_pred_discretized.append(predicted_loan_amount)

# Inverse scaling the predicted values
y_pred = y_scaler.inverse_transform(
    np.array(y_pred_discretized).reshape(-1, 1)
).flatten()

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input (as per template)
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
```

```
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using Q-learning
y_custom_pred_discretized = []
for state in X_custom:
    action = np.argmax(Q_table[:, np.random.randint(0, n_states)])
    predicted_loan_amount = loan_amount_bins[action]
    y_custom_pred_discretized.append(predicted_loan_amount)

# Inverse scaling custom predictions
y_custom_pred = y_scaler.inverse_transform(
    np.array(y_custom_pred_discretized).reshape(-1, 1)
).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")

# Loan approval predictions for custom input
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
              precision    recall  f1-score   support

    Approved       0.63      1.00      0.77       536
    Rejected       0.00      0.00      0.00       318

    accuracy                           0.63       854
   macro avg       0.31      0.50      0.39       854
weighted avg       0.39      0.63      0.48       854


Predicted loan amounts:
[2.71911285e+12 2.71911285e+12 2.71911285e+12 2.71911285e+12]

Actual applied loan amounts:
0    12300000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64

Mean Squared Error: 73934932604597757154426 88.00
Average Error: 2719097876204.46
Final Accuracy considering the reward/penalty mechanism: 0.00%


Loan Approval Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will be approved
```

Practical 2

❖ Using Deep Q-Networks (DQN)

DQN is an extension of Q-learning that leverages a neural network to approximate the Q-value function, allowing it to manage large or continuous state spaces. Instead of relying on a Q-table, it uses a deep neural network to predict Q-values for actions based on the current state. DQN also incorporates experience replay and target networks to improve learning stability. It is commonly applied in environments with vast or infinite state spaces, such as playing Atari games.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, classification_report
from collections import deque
import random
from termcolor import colored

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Discretize loan_amount (actions) into bins (for DQN)
n_actions = 10  # Define the number of discrete actions (loan amount bins)
loan_amount_bins = np.linspace(min(y_train), max(y_train), n_actions + 1)
y_train_discretized = (
    np.digitize(y_train, loan_amount_bins) - 1
)  # Discretize the loan amount


# Define the Deep Q-Network (DQN) architecture
class DQNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(DQNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_size)

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


# Parameters for DQN
input_size = X_train.shape[1]  # Number of features (states)
output_size = n_actions  # Number of actions (discretized loan amounts)
learning_rate = 0.001
gamma = 0.9  # Discount factor
epsilon = 1.0  # Exploration-exploitation tradeoff
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
memory_size = 10000
target_update_frequency = 10  # Frequency of updating the target network

# Initialize the DQN model and optimizer
model = DQNetwork(input_size, output_size)
target_model = DQNetwork(input_size, output_size)
target_model.load_state_dict(model.state_dict())  # Copy the weights
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()

# Replay memory
memory = deque(maxlen=memory_size)
```

```python
# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Epsilon-greedy action selection
def select_action(state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(0, n_actions)  # Exploration
    else:
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        q_values = model(state_tensor)
        return torch.argmax(q_values).item()  # Exploitation


# Update the target network
def update_target_network():
    target_model.load_state_dict(model.state_dict())


# Train the DQN
n_episodes = 5
for episode in range(n_episodes):
    for i, state in enumerate(X_train):
        action = select_action(state, epsilon)
        next_state = X_train[i]
        reward = -np.abs(
            loan_amount_bins[action] - y_train.iloc[i]
        )  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)

        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(states)
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
            next_states = torch.FloatTensor(next_states)

            # Compute target Q-values
            q_targets_next = target_model(next_states).max(1)[0].detach()
            q_targets = rewards + (gamma * q_targets_next)

            # Compute predicted Q-values
```

```
            q_values = model(states)
            q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze()

            # Compute loss and update the model
            loss = loss_fn(q_values, q_targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # Update the target network
    if episode % target_update_frequency == 0:
        update_target_network()

    # Decay epsilon (reduce exploration)
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

# Testing
y_pred_discretized = []
for state in X_test:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_pred_discretized.append(predicted_loan_amount)

# Inverse scaling the predicted values
y_pred = y_scaler.inverse_transform(
    np.array(y_pred_discretized).reshape(-1, 1)
).flatten()

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
```

```
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using DQN
y_custom_pred_discretized = []
for state in X_custom:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_custom_pred_discretized.append(predicted_loan_amount)

# Inverse scaling custom predictions
y_custom_pred = y_scaler.inverse_transform(
    np.array(y_custom_pred_discretized).reshape(-1, 1)
).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
              precision    recall  f1-score   support

    Approved       0.63      1.00      0.77       536
    Rejected       0.00      0.00      0.00       318

    accuracy                           0.63       854
   macro avg       0.31      0.50      0.39       854
weighted avg       0.39      0.63      0.48       854


Predicted loan amounts:
[1.09307742e+14 1.09307742e+14 1.09307742e+14 1.09307742e+14]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will be approved
```

Practical 3

❖ State-action-reward-state-action (SARSA)

SARSA is an on-policy reinforcement learning algorithm that updates Q-values based on the actions actually taken by the current policy, rather than the optimal actions. It adjusts the Q-values according to the action that was performed, even if it's not the best possible choice.

Formula:

$$Q(s,a) = Q(s,a) + \alpha\left[r + \gamma \cdot Q(s',a') - Q(s,a)\right]$$

The difference here is the use of the next action `a'` taken by the current policy, not the one with the maximum Q-value.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from collections import deque
import random
from termcolor import colored

# Preprocessing steps (from your template)
print("Reading dataset...")
data = pd.read_csv("loan_approval_dataset.csv")
print("Dataset read successfully.")
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
print("Label encoding columns...")
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})
print("Label encoding completed.")

# Separate features and target variable
print("Separating features and target variable...")
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]
print("Features and target separated.")
```

```python
# Split the data into training and testing sets
print("Splitting data into training and testing sets...")
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
print("Data split completed.")

# Standardize the features
print("Standardizing features...")
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
print("Feature standardization completed.")

# Scale the target variable as well
print("Scaling target variable...")
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()
print("Target scaling completed.")

# Discretize loan_amount (actions) into bins (for SARSA)
print("Discretizing loan amounts...")
n_actions = 10  # Define the number of discrete actions (loan amount bins)
loan_amount_bins = np.linspace(min(y_train), max(y_train), n_actions + 1)
y_train_discretized = (
    np.digitize(y_train, loan_amount_bins) - 1
)  # Discretize the loan amount
print("Loan amount discretization completed.")


# Define the SARSA Network architecture
class SARSANetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(SARSANetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_size)

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


# Parameters for SARSA
input_size = X_train.shape[1]  # Number of features (states)
output_size = n_actions  # Number of actions (discretized loan amounts)
learning_rate = 0.001
gamma = 0.9  # Discount factor
epsilon = 1.0  # Exploration-exploitation tradeoff
```

```
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
memory_size = 10000

# Initialize the SARSA model and optimizer
print("Initializing SARSA model and optimizer...")
model = SARSANetwork(input_size, output_size)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()
print("Model and optimizer initialized.")

# Replay memory
memory = deque(maxlen=memory_size)


# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Epsilon-greedy action selection
def select_action(state, epsilon):
    if np.random.rand() < epsilon:
        action = np.random.randint(0, n_actions)  # Exploration
    else:
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        q_values = model(state_tensor)
        action = torch.argmax(q_values).item()  # Exploitation
    return action


# Train the SARSA model
print("Starting training...")
n_episodes = 5
for episode in range(n_episodes):
    print(f"Episode {episode + 1}/{n_episodes}")
    for i, state in enumerate(X_train):
        action = select_action(state, epsilon)
        next_state = X_train[i]
        reward = -np.abs(
            loan_amount_bins[action] - y_train.iloc[i]
        )  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)
```

```
        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(np.array(states))
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
            next_states = torch.FloatTensor(np.array(next_states))

            # Compute target Q-values
            q_targets_next = model(next_states).max(1)[0].detach()
            q_targets = rewards + (gamma * q_targets_next)

            # Compute predicted Q-values
            q_values = model(states)
            q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze()

            # Compute loss and update the model
            loss = loss_fn(q_values, q_targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # Decay epsilon (reduce exploration)
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay
    print(f"Epsilon after episode {episode + 1}: {epsilon}")
print("Training completed.")

# Testing
print("Starting testing...")
y_pred_discretized = []
for state in X_test:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_pred_discretized.append(predicted_loan_amount)
    print(f"Test state: Predicted discretized loan amount:
{predicted_loan_amount}")

# Inverse scaling the predicted values
y_pred = y_scaler.inverse_transform(
    np.array(y_pred_discretized).reshape(-1, 1)
).flatten()
print("Testing completed. Predicted loan amounts:")
print(y_pred)

# Testing on custom input (as per template)
print("Testing on custom input...")
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
```

```
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
print("Preprocessing custom input...")
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)
print("Custom input preprocessing completed.")

# Predicting using SARSA
print("Predicting on custom input...")
y_custom_pred_discretized = []
for state in X_custom:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_custom_pred_discretized.append(predicted_loan_amount)
    print(f"Custom state: Predicted discretized loan amount:
{predicted_loan_amount}")

# Inverse scaling custom predictions
y_custom_pred = y_scaler.inverse_transform(
    np.array(y_custom_pred_discretized).reshape(-1, 1)
).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")

# Loan approval predictions
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
import numpy as np
```

```python
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, classification_report
from collections import deque
import random
from termcolor import colored

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Discretize loan_amount (actions) into bins (for SARSA)
n_actions = 10  # Define the number of discrete actions (loan amount bins)
loan_amount_bins = np.linspace(min(y_train), max(y_train), n_actions + 1)
y_train_discretized = (
    np.digitize(y_train, loan_amount_bins) - 1
)  # Discretize the loan amount


# Define the SARSA Network architecture
class SARSANetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(SARSANetwork, self).__init__()
```

```
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_size)

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


# Parameters for SARSA
input_size = X_train.shape[1]  # Number of features (states)
output_size = n_actions  # Number of actions (discretized loan amounts)
learning_rate = 0.001
gamma = 0.9  # Discount factor
epsilon = 1.0  # Exploration-exploitation tradeoff
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
memory_size = 10000

# Initialize the SARSA model and optimizer
model = SARSANetwork(input_size, output_size)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_fn = nn.MSELoss()

# Replay memory
memory = deque(maxlen=memory_size)


# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Epsilon-greedy action selection
def select_action(state, epsilon):
    if np.random.rand() < epsilon:
        action = np.random.randint(0, n_actions)  # Exploration
    else:
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        q_values = model(state_tensor)
        action = torch.argmax(q_values).item()  # Exploitation
    return action


# Train the SARSA model
```

```python
n_episodes = 5
for episode in range(n_episodes):
    for i, state in enumerate(X_train):
        action = select_action(state, epsilon)
        next_state = X_train[i]
        reward = -np.abs(
            loan_amount_bins[action] - y_train.iloc[i]
        )  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)

        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(np.array(states))
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
            next_states = torch.FloatTensor(np.array(next_states))

            # Compute target Q-values
            q_targets_next = model(next_states).max(1)[0].detach()
            q_targets = rewards + (gamma * q_targets_next)

            # Compute predicted Q-values
            q_values = model(states)
            q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze()

            # Compute loss and update the model
            loss = loss_fn(q_values, q_targets)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # Decay epsilon (reduce exploration)
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

# Testing
y_pred_discretized = []
for state in X_test:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_pred_discretized.append(predicted_loan_amount)

# Inverse scaling the predicted values
y_pred = y_scaler.inverse_transform(
    np.array(y_pred_discretized).reshape(-1, 1)
).flatten()
```

```
# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using SARSA
y_custom_pred_discretized = []
for state in X_custom:
    action = select_action(state, epsilon=0.0)  # Pure exploitation
    predicted_loan_amount = loan_amount_bins[action]
    y_custom_pred_discretized.append(predicted_loan_amount)

# Inverse scaling custom predictions
y_custom_pred = y_scaler.inverse_transform(
    np.array(y_custom_pred_discretized).reshape(-1, 1)
).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")
```

```
# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
                precision    recall  f1-score   support

     Approved       0.63      1.00      0.77       536
     Rejected       0.00      0.00      0.00       318

     accuracy                           0.63       854
    macro avg       0.31      0.50      0.39       854
 weighted avg       0.39      0.63      0.48       854


Predicted loan amounts:
[1.09307742e+14 1.09307742e+14 1.09307742e+14 1.09307742e+14]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will be approved
```

Practical 4

❖ **Using DDPG.**

DDPG is a model-free, actor-critic algorithm designed for continuous action spaces, utilizing deterministic policies. It consists of two networks: the actor, which selects actions, and the critic, which evaluates them. Similar to DQN, DDPG employs experience replay and target networks. Its ability to manage continuous action spaces makes it particularly suitable for robotics control tasks.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, classification_report
from collections import deque
import random
from termcolor import colored

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```python
# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for DDPG
input_size = X_train.shape[1]  # Number of features (states)
output_size = 1  # Continuous output (loan amount prediction)
learning_rate_actor = 0.001
learning_rate_critic = 0.001
gamma = 0.99  # Discount factor
tau = 0.001  # For soft target updates
batch_size = 64
memory_size = 10000

# Replay memory
memory = deque(maxlen=memory_size)


# Define Actor and Critic networks
class ActorNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, output_size)

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


class CriticNetwork(nn.Module):
    def __init__(self, input_size, action_size):
        super(CriticNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size + action_size, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 1)

    def forward(self, x, a):
        a = (
            a if a.dim() == 2 else a.unsqueeze(1)
        )  # Ensure action tensor has correct dimensions
        x = torch.cat([x, a], dim=1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)


# Initialize Actor, Critic, and their target networks
```

```
actor = ActorNetwork(input_size, output_size)
target_actor = ActorNetwork(input_size, output_size)
critic = CriticNetwork(input_size, output_size)
target_critic = CriticNetwork(input_size, output_size)

# Copy weights from the original networks to the target networks
target_actor.load_state_dict(actor.state_dict())
target_critic.load_state_dict(critic.state_dict())

# Optimizers
actor_optimizer = optim.Adam(actor.parameters(), lr=learning_rate_actor)
critic_optimizer = optim.Adam(critic.parameters(), lr=learning_rate_critic)
loss_fn = nn.MSELoss()


# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Soft update target networks
def soft_update(target, source, tau):
    for target_param, source_param in zip(target.parameters(),
source.parameters()):
        target_param.data.copy_(
            tau * source_param.data + (1.0 - tau) * target_param.data
        )


# Train the DDPG model
n_episodes = 5
for episode in range(n_episodes):
    for i, state in enumerate(X_train):
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        action = actor(state_tensor).detach().numpy().flatten()
        next_state = X_train[i]
        reward = -np.abs(y_train_scaled[i] - action[0])  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)

        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(np.array(states))
            actions = (
```

```
                torch.FloatTensor(np.array(actions)).unsqueeze(1)
                if actions[0].ndim == 0
                else torch.FloatTensor(np.array(actions))
            )
            rewards = torch.FloatTensor(rewards).unsqueeze(1)
            next_states = torch.FloatTensor(np.array(next_states))

            # Compute target Q-values
            next_actions = target_actor(next_states)
            target_q_values = target_critic(next_states, next_actions).detach()
            q_targets = rewards + (gamma * target_q_values)

            # Compute predicted Q-values and update Critic network
            q_values = critic(states, actions)
            critic_loss = loss_fn(q_values, q_targets)
            critic_optimizer.zero_grad()
            critic_loss.backward()
            critic_optimizer.step()

            # Update Actor network
            actor_loss = -critic(states, actor(states)).mean()
            actor_optimizer.zero_grad()
            actor_loss.backward()
            actor_optimizer.step()

            # Soft update target networks
            soft_update(target_actor, actor, tau)
            soft_update(target_critic, critic, tau)

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
```

```
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using DDPG
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
Classification Report:
              precision    recall  f1-score   support

    Approved       0.60      0.54      0.57       536
    Rejected       0.34      0.39      0.36       318

    accuracy                           0.49       854
   macro avg       0.47      0.47      0.47       854
weighted avg       0.50      0.49      0.49       854


Predicted loan amounts:
[9751720.296795422, 6109957.480170436, 12331793.704231085, 6109957.480170436]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 5

❖ Using TD3

TD3 is an enhanced version of DDPG designed to counter its tendency to overestimate Q-values, which can degrade performance. It introduces two critics (hence the "twin") to reduce Q-value overestimation. Additionally, it employs target smoothing to lower variance in updates and delays the actor network's updates to promote more stable learning.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score, classification_report
from collections import deque
import random
from termcolor import colored

# Preprocessing steps (from your template)
print("Reading dataset...")
data = pd.read_csv("loan_approval_dataset.csv")

loan_status = data["loan_status"]  # Save loan_status before dropping
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns

data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})


# Separate features and target variable

X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]


# Split the data into training and testing sets

X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)
```

```python
# Standardize the features

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)


# Scale the target variable as well

y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()


# Parameters for TD3
input_size = X_train.shape[1]  # Number of features (states)
output_size = 1  # Continuous output (loan amount prediction)
learning_rate_actor = 0.001
learning_rate_critic = 0.001
gamma = 0.99  # Discount factor
tau = 0.005  # For soft target updates
batch_size = 64
memory_size = 10000
policy_noise = 0.2
noise_clip = 0.5
policy_freq = 2

# Replay memory
memory = deque(maxlen=memory_size)


# Define Actor and Critic networks
class ActorNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, output_size)

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


class CriticNetwork(nn.Module):
    def __init__(self, input_size, action_size):
        super(CriticNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size + action_size, 400)
        self.fc2 = nn.Linear(400, 300)
```

```
        self.fc3 = nn.Linear(300, 1)

    def forward(self, x, a):
        a = (
            a if a.dim() == 2 else a.unsqueeze(1)
        )  # Ensure action tensor has correct dimensions
        x = torch.cat([x, a], dim=1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)


# Initialize Actor, Critic, and their target networks

actor = ActorNetwork(input_size, output_size)
target_actor = ActorNetwork(input_size, output_size)
critic_1 = CriticNetwork(input_size, output_size)
target_critic_1 = CriticNetwork(input_size, output_size)
critic_2 = CriticNetwork(input_size, output_size)
target_critic_2 = CriticNetwork(input_size, output_size)

# Copy weights from the original networks to the target networks
target_actor.load_state_dict(actor.state_dict())
target_critic_1.load_state_dict(critic_1.state_dict())
target_critic_2.load_state_dict(critic_2.state_dict())


# Optimizers
actor_optimizer = optim.Adam(actor.parameters(), lr=learning_rate_actor)
critic_1_optimizer = optim.Adam(critic_1.parameters(), lr=learning_rate_critic)
critic_2_optimizer = optim.Adam(critic_2.parameters(), lr=learning_rate_critic)
loss_fn = nn.MSELoss()


# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Soft update target networks
def soft_update(target, source, tau):
    for target_param, source_param in zip(target.parameters(),
source.parameters()):
        target_param.data.copy_(
            tau * source_param.data + (1.0 - tau) * target_param.data
        )
```

```python
# Train the TD3 model
print("Starting training...")
n_episodes = 5
for episode in range(n_episodes):
    print(f"Episode {episode + 1}/{n_episodes}")
    for i, state in enumerate(X_train):
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        action = actor(state_tensor).detach().numpy().flatten()
        next_state = X_train[i]
        reward = -np.abs(y_train_scaled[i] - action[0])  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)

        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(np.array(states))
            actions = (
                torch.FloatTensor(np.array(actions)).unsqueeze(1)
                if actions[0].ndim == 0
                else torch.FloatTensor(np.array(actions))
            )
            rewards = torch.FloatTensor(rewards).unsqueeze(1)
            next_states = torch.FloatTensor(np.array(next_states))

            # Add noise to target actions
            noise = torch.clamp(
                torch.normal(0, policy_noise, size=actions.shape),
                -noise_clip,
                noise_clip,
            )
            next_actions = target_actor(next_states) + noise
            next_actions = torch.clamp(next_actions, -1, 1)

            # Compute target Q-values
            target_q1_values = target_critic_1(next_states, next_actions).detach()
            target_q2_values = target_critic_2(next_states, next_actions).detach()
            q_targets = rewards + (
                gamma * torch.min(target_q1_values, target_q2_values)
            )

            # Compute predicted Q-values and update Critic networks
            q1_values = critic_1(states, actions)
            critic_1_loss = loss_fn(q1_values, q_targets)
            critic_1_optimizer.zero_grad()
            critic_1_loss.backward()
            critic_1_optimizer.step()

            q2_values = critic_2(states, actions)
```

```
            critic_2_loss = loss_fn(q2_values, q_targets)
            critic_2_optimizer.zero_grad()
            critic_2_loss.backward()
            critic_2_optimizer.step()

            # Update Actor network
            if i % policy_freq == 0:
                actor_loss = -critic_1(states, actor(states)).mean()
                actor_optimizer.zero_grad()
                actor_loss.backward()
                actor_optimizer.step()

                # Soft update target networks
                soft_update(target_actor, actor, tau)
                soft_update(target_critic_1, critic_1, tau)
                soft_update(target_critic_2, critic_2, tau)

print("Training completed.")

# Testing
print("Starting testing...")
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input (as per template)
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
```

```
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using TD3
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
apple@MacBook-Air-2 Lab % /Users/apple/Coded/College/RL/Lab/env/bin/python
/Users/apple/Coded/College/RL/Lab/td3.py
Reading dataset...
Starting training...
Episode 1/5
Episode 2/5
Episode 3/5
Episode 4/5
Episode 5/5
Training completed.
Starting testing...

Classification Report:
              precision    recall  f1-score   support

    Approved       0.60      0.50      0.54       536
    Rejected       0.34      0.43      0.38       318

    accuracy                           0.47       854
   macro avg       0.47      0.46      0.46       854
weighted avg       0.50      0.47      0.48       854



Predicted loan amounts:
[10217652.384126294, 6109957.480170436, 11055592.718949573, 6109957.4801704
36]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 6

❖ **Using Asynchronous Advantage Actor-Critic (A3C).**

A3C is a policy gradient algorithm that accelerates training by having multiple agents learn in parallel asynchronously, enabling them to explore different areas of the environment at the same time. It integrates actor-critic methods with asynchronous learning and uses an advantage function to stabilize training. This function estimates how much better an action is compared to the average action.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, classification_report
from collections import deque
import random
from termcolor import colored

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```python
# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for DDPG
input_size = X_train.shape[1]  # Number of features (states)
output_size = 1  # Continuous output (loan amount prediction)
learning_rate_actor = 0.001
learning_rate_critic = 0.001
gamma = 0.99  # Discount factor
tau = 0.001  # For soft target updates
batch_size = 64
memory_size = 10000

# Replay memory
memory = deque(maxlen=memory_size)


# Define Actor and Critic networks
class ActorNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(ActorNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, output_size)

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


class CriticNetwork(nn.Module):
    def __init__(self, input_size, action_size):
        super(CriticNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size + action_size, 400)
        self.fc2 = nn.Linear(400, 300)
        self.fc3 = nn.Linear(300, 1)

    def forward(self, x, a):
        a = (
            a if a.dim() == 2 else a.unsqueeze(1)
        )  # Ensure action tensor has correct dimensions
        x = torch.cat([x, a], dim=1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)


# Initialize Actor, Critic, and their target networks
```

```
actor = ActorNetwork(input_size, output_size)
target_actor = ActorNetwork(input_size, output_size)
critic = CriticNetwork(input_size, output_size)
target_critic = CriticNetwork(input_size, output_size)

# Copy weights from the original networks to the target networks
target_actor.load_state_dict(actor.state_dict())
target_critic.load_state_dict(critic.state_dict())

# Optimizers
actor_optimizer = optim.Adam(actor.parameters(), lr=learning_rate_actor)
critic_optimizer = optim.Adam(critic.parameters(), lr=learning_rate_critic)
loss_fn = nn.MSELoss()


# Store experiences in the replay buffer
def store_experience(state, action, reward, next_state, done):
    memory.append((state, action, reward, next_state, done))


# Sample a batch of experiences
def sample_experiences(batch_size):
    return random.sample(memory, batch_size)


# Soft update target networks
def soft_update(target, source, tau):
    for target_param, source_param in zip(target.parameters(),
source.parameters()):
        target_param.data.copy_(
            tau * source_param.data + (1.0 - tau) * target_param.data
        )


# Train the DDPG model
n_episodes = 5
for episode in range(n_episodes):
    for i, state in enumerate(X_train):
        state_tensor = torch.FloatTensor(state).unsqueeze(0)
        action = actor(state_tensor).detach().numpy().flatten()
        next_state = X_train[i]
        reward = -np.abs(y_train_scaled[i] - action[0])  # Reward is negative error

        # Store experience in the replay buffer
        store_experience(state, action, reward, next_state, False)

        # Sample a batch of experiences from the memory
        if len(memory) > batch_size:
            experiences = sample_experiences(batch_size)
            states, actions, rewards, next_states, dones = zip(*experiences)
            states = torch.FloatTensor(np.array(states))
            actions = (
```

```
                torch.FloatTensor(np.array(actions)).unsqueeze(1)
                if actions[0].ndim == 0
                else torch.FloatTensor(np.array(actions))
            )
            rewards = torch.FloatTensor(rewards).unsqueeze(1)
            next_states = torch.FloatTensor(np.array(next_states))

            # Compute target Q-values
            next_actions = target_actor(next_states)
            target_q_values = target_critic(next_states, next_actions).detach()
            q_targets = rewards + (gamma * target_q_values)

            # Compute predicted Q-values and update Critic network
            q_values = critic(states, actions)
            critic_loss = loss_fn(q_values, q_targets)
            critic_optimizer.zero_grad()
            critic_loss.backward()
            critic_optimizer.step()

            # Update Actor network
            actor_loss = -critic(states, actor(states)).mean()
            actor_optimizer.zero_grad()
            actor_loss.backward()
            actor_optimizer.step()

            # Soft update target networks
            soft_update(target_actor, actor, tau)
            soft_update(target_critic, critic, tau)

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
```

```
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using DDPG
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    action = actor(state_tensor).detach().numpy().flatten()[0]
    predicted_loan_amount = y_scaler.inverse_transform([[action]])[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
Classification Report:
              precision    recall  f1-score   support

    Approved       0.59      0.53      0.56       536
    Rejected       0.33      0.39      0.35       318

    accuracy                           0.48       854
   macro avg       0.46      0.46      0.46       854
weighted avg       0.49      0.48      0.48       854


Predicted loan amounts:
[10982776.176398007, 6109957.480170436, 13228874.908135025, 6109957.480170436]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 7

❖ Using Trust Region Policy Optimization (TRPO)

TRPO is a policy gradient method designed to prevent large policy updates from causing significant performance declines by keeping updates within a "trust region." It optimizes policies by limiting how much the new policy can differ from the old one, ensuring stable learning and preventing disruptive changes during training.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from torch.distributions import MultivariateNormal
from termcolor import colored

# Preprocessing steps (from your template)
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status before dropping
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
```

```
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for TRPO
input_size = X_train.shape[1]
output_size = 1
learning_rate = 0.001
max_kl_divergence = 0.01
n_episodes = 5


# Define the Policy Network
class PolicyNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.mean_layer = nn.Linear(128, output_size)
        self.log_std = nn.Parameter(torch.zeros(output_size))

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        mean = self.mean_layer(x)
        std = torch.exp(self.log_std)
        return mean, std


# Define the Value Network
class ValueNetwork(nn.Module):
    def __init__(self, input_size):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.value_layer = nn.Linear(128, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        value = self.value_layer(x)
        return value


# Initialize networks
policy_network = PolicyNetwork(input_size, output_size)
value_network = ValueNetwork(input_size)
value_optimizer = optim.Adam(value_network.parameters(), lr=learning_rate)


# Compute surrogate loss
def surrogate_loss(old_log_probs, new_log_probs, advantages):
```

```
    return torch.mean(torch.exp(new_log_probs - old_log_probs) * advantages)


# Train the TRPO model
for episode in range(n_episodes):
    states = []
    actions = []
    rewards = []
    log_probs = []

    # Collect trajectories
    for i in range(len(X_train)):
        state = torch.FloatTensor(X_train[i]).unsqueeze(0)
        mean, std = policy_network(state)
        dist = MultivariateNormal(mean, torch.diag(std))
        action = dist.sample()
        log_prob = dist.log_prob(action)

        reward = -np.abs(y_train_scaled[i] - action.item())  # Reward is negative
error

        states.append(state)
        actions.append(action)
        rewards.append(reward)
        log_probs.append(log_prob)

    # Compute value targets
    values = torch.cat([value_network(state) for state in states])
    rewards = torch.FloatTensor(rewards).unsqueeze(1)
    advantages = rewards - values.detach()

    # Update value network
    value_loss = torch.mean((values - rewards) ** 2)
    value_optimizer.zero_grad()
    value_loss.backward()
    value_optimizer.step()

    # Update policy network using TRPO
    old_log_probs = torch.cat(log_probs)
    for _ in range(10):  # Iterate for policy optimization
        new_log_probs = torch.cat(
            [
                MultivariateNormal(
                    policy_network(state)[0],
                    torch.diag(torch.exp(policy_network(state)[1])),
                ).log_prob(action)
                for state, action in zip(states, actions)
            ]
        )
        loss = surrogate_loss(old_log_probs, new_log_probs, advantages)
        kl_div = torch.mean(old_log_probs - new_log_probs).abs()
```

```
        if kl_div > max_kl_divergence:
            break

        value_optimizer.zero_grad()
        loss.backward()
        value_optimizer.step()

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    mean, _ = policy_network(state_tensor)
    predicted_loan_amount = y_scaler.inverse_transform(mean.detach().numpy())[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input (as per template)
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
```

```
X_custom = scaler.transform(X_custom)

# Predicting using TRPO
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    mean, _ = policy_network(state_tensor)
    predicted_loan_amount = y_scaler.inverse_transform(mean.detach().numpy())[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

Output:

```
Classification Report:
              precision    recall  f1-score   support

    Approved       0.61      0.49      0.54       536
    Rejected       0.36      0.48      0.41       318

    accuracy                           0.48       854
   macro avg       0.49      0.48      0.48       854
weighted avg       0.52      0.48      0.49       854


Predicted loan amounts:
[13536135.0, 13220936.0, 12435057.0, 13228838.0]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will be approved
```

Practical 8

❖ Using Proximal Policy Optimization (PPO).

PPO is an enhanced version of TRPO that improves the efficiency of policy updates by simplifying the optimization process, while still maintaining stable updates. It restricts policy changes by clipping the objective function to keep the new policy close to the current one. PPO is widely adopted due to its simplicity compared to TRPO and its ability to deliver strong results in practice.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from torch.distributions import MultivariateNormal

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
```

```python
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for PPO
input_size = X_train.shape[1]
output_size = 1
learning_rate = 0.001
gamma = 0.99
eps_clip = 0.2
k_epochs = 10
n_episodes = 5


# Define the Policy Network
class PolicyNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.mean_layer = nn.Linear(128, output_size)
        self.log_std = nn.Parameter(torch.zeros(output_size))

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        mean = self.mean_layer(x)
        std = torch.exp(self.log_std)
        return mean, std


# Define the Value Network
class ValueNetwork(nn.Module):
    def __init__(self, input_size):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.value_layer = nn.Linear(128, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        value = self.value_layer(x)
        return value


# Initialize networks
policy_network = PolicyNetwork(input_size, output_size)
value_network = ValueNetwork(input_size)
policy_optimizer = optim.Adam(policy_network.parameters(), lr=learning_rate)
value_optimizer = optim.Adam(value_network.parameters(), lr=learning_rate)


# Compute PPO loss
```

```
def ppo_loss(old_log_probs, new_log_probs, advantages, eps_clip):
    ratio = torch.exp(new_log_probs - old_log_probs)
    surr1 = ratio * advantages
    surr2 = torch.clamp(ratio, 1 - eps_clip, 1 + eps_clip) * advantages
    return -torch.min(surr1, surr2).mean()


# Train the PPO model
for episode in range(n_episodes):
    states = []
    actions = []
    rewards = []
    log_probs = []

    # Collect trajectories
    for i in range(len(X_train)):
        state = torch.FloatTensor(X_train[i]).unsqueeze(0)
        mean, std = policy_network(state)
        dist = MultivariateNormal(mean, torch.diag(std + 1e-6))
        action = dist.sample()
        log_prob = dist.log_prob(action)

        reward = -np.abs(y_train_scaled[i] - action.item())  # Reward is negative
error

        states.append(state)
        actions.append(action)
        rewards.append(reward)
        log_probs.append(log_prob)

    # Compute value targets
    values = torch.cat([value_network(state) for state in states])
    rewards = torch.FloatTensor(rewards).unsqueeze(1)
    advantages = rewards - values.detach()

    # Update value network
    value_loss = torch.mean((values - rewards) ** 2)
    value_optimizer.zero_grad()
    value_loss.backward(retain_graph=True)
    value_optimizer.step()

    # Update policy network using PPO
    old_log_probs = torch.cat(log_probs).detach()
    for _ in range(k_epochs):
        new_log_probs = torch.cat(
            [
                MultivariateNormal(
                    policy_network(state)[0],
                    torch.diag(torch.exp(policy_network(state)[1]) + 1e-6),
                ).log_prob(action)
                for state, action in zip(states, actions)
```

```
            ]
        )
        loss = ppo_loss(old_log_probs, new_log_probs, advantages, eps_clip)

        policy_optimizer.zero_grad()
        loss.backward()
        policy_optimizer.step()

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    mean, _ = policy_network(state_tensor)
    predicted_loan_amount = y_scaler.inverse_transform(mean.detach().numpy())[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(
    classification_report(
        y_test.apply(lambda x: "Approved" if x >= 0 else "Rejected"),
y_pred_loan_status
    )
)

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
```

```
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using PPO
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    mean, _ = policy_network(state_tensor)
    predicted_loan_amount = y_scaler.inverse_transform(mean.detach().numpy())[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(f"Test Case {i+1}: Loan will be approved")
    else:
        print(f"Test Case {i+1}: Loan will not be approved")
```

Output:

```
              precision    recall  f1-score   support

    Approved       1.00      0.69      0.81       854
    Rejected       0.00      0.00      0.00         0

    accuracy                           0.69       854
   macro avg       0.50      0.34      0.41       854
weighted avg       1.00      0.69      0.81       854


Predicted loan amounts:
[22224086.0, 24802280.0, 23093658.0, 23719078.0]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will be approved
```

Practical 9

❖ Using C51.

C51 is a distributional reinforcement learning algorithm that models the full distribution of potential future rewards, rather than only predicting the expected reward. Unlike DQN, which predicts a single Q-value, C51 predicts a distribution over possible rewards. It categorizes future rewards into 51 discrete bins, which gives the algorithm its name, “C51”.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
```

```
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for C51
input_size = X_train.shape[1]
output_size = 51  # Number of atoms in C51
learning_rate = 0.001
gamma = 0.99
v_min = -10
v_max = 10
n_atoms = 51
delta_z = (v_max - v_min) / (n_atoms - 1)
z = torch.linspace(v_min, v_max, n_atoms)


# Define the C51 Network
class C51Network(nn.Module):
    def __init__(self, input_size, output_size):
        super(C51Network, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.output_layer = nn.Linear(128, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        logits = self.output_layer(x)
        return torch.softmax(logits.view(-1, n_atoms), dim=1)


# Initialize network
c51_network = C51Network(input_size, output_size)
optimizer = optim.Adam(c51_network.parameters(), lr=learning_rate)

# Train the C51 model
n_episodes = 5
for episode in range(n_episodes):
    for i in range(len(X_train)):
        state = torch.FloatTensor(X_train[i]).unsqueeze(0)
        target_distribution = torch.zeros((1, n_atoms))

        with torch.no_grad():
            mean = y_train_scaled[i]
            b = torch.tensor((mean - v_min) / delta_z)
            l = int(torch.floor(b).item())
            u = int(torch.ceil(b).item())

            target_distribution[0, l] += u - b
            if u < n_atoms:
                target_distribution[0, u] += b - l

        # Forward pass
```

```
        predicted_distribution = c51_network(state)

        # Compute loss
        loss = -(target_distribution * torch.log(predicted_distribution)).sum()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    predicted_distribution = c51_network(state_tensor)
    expected_value = torch.sum(predicted_distribution * z, dim=1).item()
    predicted_loan_amount = y_scaler.inverse_transform([[expected_value]])[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
```

```
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using C51
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    predicted_distribution = c51_network(state_tensor)
    expected_value = torch.sum(predicted_distribution * z, dim=1).item()
    predicted_loan_amount = y_scaler.inverse_transform([[expected_value]])[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(f"Test Case {i+1}: Loan will be approved")
    else:
        print(f"Test Case {i+1}: Loan will not be approved")
```

Output:

```
Classification Report:
              precision    recall  f1-score   support

    Approved       0.59      0.56      0.57       536
    Rejected       0.32      0.35      0.33       318

    accuracy                           0.48       854
   macro avg       0.45      0.45      0.45       854
weighted avg       0.49      0.48      0.48       854


Predicted loan amounts:
[11894880.943017442, 3128505.3456145246, 16626297.14359054, 3547012.260510236]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 10

❖ **Using Distributional Reinforcement Learning with Quantile Regression (QR-DQN).**

QR-DQN is a distributional reinforcement learning algorithm that uses quantile regression to approximate the reward distribution. Unlike C51, which relies on fixed categories, QR-DQN predicts the quantiles of the reward distribution. This method provides a more flexible and accurate estimation of future rewards, allowing for a finer representation of the reward distribution.

Code:

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report

# Preprocessing steps
data = pd.read_csv("loan_approval_dataset.csv")
loan_status = data["loan_status"]  # Save loan_status for evaluation
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test, loan_status_train, loan_status_test = (
    train_test_split(X, y, loan_status, test_size=0.2, random_state=42)
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Scale the target variable as well
y_scaler = StandardScaler()
y_train_scaled = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test_scaled = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Parameters for QR-DQN
input_size = X_train.shape[1]
n_quantiles = 51
learning_rate = 0.001
gamma = 0.99
tau = torch.linspace(0.0, 1.0, n_quantiles + 1)[1:]  # Quantiles to estimate


# Define the QR-DQN Network
class QRDQNNetwork(nn.Module):
    def __init__(self, input_size, n_quantiles):
        super(QRDQNNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.output_layer = nn.Linear(128, n_quantiles)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        quantiles = self.output_layer(x)
        return quantiles


# Initialize network
qr_dqn_network = QRDQNNetwork(input_size, n_quantiles)
optimizer = optim.Adam(qr_dqn_network.parameters(), lr=learning_rate)

# Train the QR-DQN model
n_episodes = 5
for episode in range(n_episodes):
    for i in range(len(X_train)):
        state = torch.FloatTensor(X_train[i]).unsqueeze(0)
        target_quantiles = torch.zeros((1, n_quantiles))

        with torch.no_grad():
            target = y_train_scaled[i]
            td_error = target - qr_dqn_network(state)
            target_quantiles = td_error * (tau - (td_error < 0).float())

        # Forward pass
        predicted_quantiles = qr_dqn_network(state)

        # Compute loss
        loss = torch.mean(target_quantiles * predicted_quantiles)

        # Backpropagation
```

```
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# Testing
y_pred = []
for state in X_test:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    predicted_quantiles = qr_dqn_network(state_tensor)
    expected_value = torch.mean(predicted_quantiles).item()
    predicted_loan_amount = y_scaler.inverse_transform([[expected_value]])[0][0]
    y_pred.append(predicted_loan_amount)

# Generate predicted loan status based on predicted loan amount
y_pred_loan_status = [
    "Approved" if pred >= actual else "Rejected" for pred, actual in zip(y_pred,
y_test)
]

# Generate classification report
print("\nClassification Report:")
print(classification_report(loan_status_test, y_pred_loan_status))

# Testing on custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Preprocessing custom input
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]
X_custom = scaler.transform(X_custom)

# Predicting using QR-DQN
```

```
y_custom_pred = []
for state in X_custom:
    state_tensor = torch.FloatTensor(state).unsqueeze(0)
    predicted_quantiles = qr_dqn_network(state_tensor)
    expected_value = torch.mean(predicted_quantiles).item()
    predicted_loan_amount = y_scaler.inverse_transform([[expected_value]])[0][0]
    y_custom_pred.append(predicted_loan_amount)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.tolist()}")

# Loan approval predictions
print("\n\nLoan Approval Predictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.iloc[i]:
        print(f"Test Case {i+1}: Loan will be approved")
    else:
        print(f"Test Case {i+1}: Loan will not be approved")
```

Output:

```
              precision    recall  f1-score   support

    Approved       0.00      0.00      0.00       536
    Rejected       0.37      1.00      0.54       318

    accuracy                           0.37       854
   macro avg       0.19      0.50      0.27       854
weighted avg       0.14      0.37      0.20       854


Predicted loan amounts:
[-2145579777806496.2, -3479820559521446.0, -2214916188455581.0, -2792260965220275.5]

Actual applied loan amounts:
[12300000, 5000000, 1500000, 10000000]


Loan Approval Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will not be approved
Test Case 4: Loan will not be approved
apple@MacBook-Air-2 Lab %
```