Aim - Develop a model-based RL algorithm, such as Monte Carlo Tree Search (MCTS), to solve a complex environment like Atari games.

Objective –

The objective of this experiment is to implement and train a Deep Q-Network (DQN) using PyTorch on the CartPole environment from OpenAI Gym. The goal is to learn an optimal policy to balance the pole on a cart by moving left or right.

Software Requirements –

1) Python (3.x recommended)

2)Jupyter Notebook or any Python IDE

Hardware Requirements - A machine with sufficient RAM and processing power for model training (8GB RAM recommended)

Prerequisites –

1) Basic understanding of Python programming

2) Familiarity with the concepts of Neural Networks

Libraries or Modules Used -

1) TensorFlow or PyTorch

2) NumPy

3) Matplotlib

Theory –

Methodology :

Environment: Utilize the CartPole environment provided by OpenAI Gym.

DQN Architecture: Design a neural network model to serve as the Q-network, which takes the state of the environment as input and outputs Q-values for each action.

Replay Buffer: Implement a replay buffer to store and sample experiences for training the DQN.

Training: Train the DQN using a combination of Q-learning and experience replay to optimize the Q-values.

Evaluation: Assess the performance of the trained DQN by measuring the average reward obtained over a specified number of episodes.

Hypothesis:

We hypothesize that the DQN will be able to learn an effective policy to balance the pole on the cart by maximizing the cumulative reward over episodes through reinforcement learning.

Experiment Setup:

Environment: CartPole-v1 environment from OpenAI Gym.

Neural Network: PyTorch-based DQN architecture.

Training Parameters: Learning rate, discount factor, exploration parameters.

Evaluation Metrics: Average reward per episode, stability of training.

Results and Analysis:

Analyze the performance of the trained DQN by examining the average reward obtained per episode and assessing the stability of training. Evaluate whether the DQN is able to learn an effective policy for balancing the pole on the cart.

Code -

```
import gym

import random

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim


# Define Deep Q-Network (DQN) model

class DQN(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):

        super(DQN, self).__init__()

        self.fc1 = nn.Linear(input_size, hidden_size)

        self.fc2 = nn.Linear(hidden_size, hidden_size)

        self.fc3 = nn.Linear(hidden_size, output_size)


    def forward(self, x):
```

```python
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# Define replay buffer
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []

    def add(self, state, action, reward, next_state, done):
        experience = (state, action, reward, next_state, done)
        self.buffer.append(experience)
        if len(self.buffer) > self.capacity:
            self.buffer.pop(0)

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

# Define Deep Q-Learning agent
class DQNAgent:
    def __init__(self, input_size, output_size, hidden_size=64, learning_rate=0.001, gamma=0.99, epsilon_start=1.0, epsilon_decay=0.995, epsilon_min=0.01, replay_buffer_capacity=10000, batch_size=64):
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate
        self.gamma = gamma
```

```
        self.epsilon = epsilon_start
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.replay_buffer = ReplayBuffer(replay_buffer_capacity)
        self.batch_size = batch_size

        self.q_network = DQN(input_size, hidden_size, output_size)
        self.target_network = DQN(input_size, hidden_size, output_size)
        self.target_network.load_state_dict(self.q_network.state_dict())
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=learning_rate)
        self.loss_function = nn.MSELoss()

    def epsilon_greedy_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(range(self.output_size))
        else:
            with torch.no_grad():
                q_values = self.q_network(torch.FloatTensor(state))
                return torch.argmax(q_values).item()

    def train(self, state, action, reward, next_state, done):
        self.replay_buffer.add(state, action, reward, next_state, done)
        if len(self.replay_buffer.buffer) > self.batch_size:
            batch = self.replay_buffer.sample(self.batch_size)
            states, actions, rewards, next_states, dones = zip(*batch)

            states = torch.FloatTensor(states)
            actions = torch.LongTensor(actions)
            rewards = torch.FloatTensor(rewards)
```

```
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones)

        q_values = self.q_network(states)
        next_q_values = self.target_network(next_states).max(1)[0]
        target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

        q_values = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

        loss = self.loss_function(q_values, target_q_values.detach())

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

        self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)

    def update_target_network(self):
        self.target_network.load_state_dict(self.q_network.state_dict())

# Initialize environment and agent
env = gym.make('CartPole-v1')
input_size = env.observation_space.shape[0]
output_size = env.action_space.n
agent = DQNAgent(input_size, output_size)

# Training
num_episodes = 1000
for episode in range(num_episodes):
```

```
    state = env.reset()
    total_reward = 0
    done = False

    while not done:
        action = agent.epsilon_greedy_action(state)
        next_state, reward, done, _ = env.step(action)
        agent.train(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward

    if episode % 100 == 0:
        agent.update_target_network()
        print(f"Episode {episode}, Total Reward: {total_reward}")

# Close the environment
env.close()
```

Conclusion : This code sets up a DQN agent using PyTorch and trains it on the CartPole environment. It includes the necessary components such as the DQN model, replay buffer, and training algorithm. The agent learns to balance the pole on the cart by maximizing cumulative rewards over episodes.