Aim - 3. Implement a Deep Q-Network (DQN) using a deep neural network library (e.g., TensorFlow or PyTorch) and train it on a simple environment like CartPole or Mountain Car.

Objective –

The objective of this experiment is to implement and train a Deep Q-Network (DQN) using PyTorch on the CartPole environment provided by the OpenAI Gym. The goal is to train an agent to balance a pole on a cart by making continuous movements.

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 Setup:

The CartPole environment from the OpenAI Gym is used for this experiment.

The environment provides observations consisting of the cart's position, cart's velocity, pole's angle, and pole's angular velocity.

Neural Network Architecture:

A simple feedforward neural network is used as the Q-network.

The network architecture consists of fully connected layers with ReLU activation functions.

The final layer outputs the Q-values for each action.

Replay Memory:

Experience replay is employed to break the correlation between consecutive samples.

A replay memory buffer stores experiences consisting of (state, action, next_state, reward) tuples.

Agent Implementation:

The DQNAgent class is implemented, which includes methods for selecting actions, optimizing the model, and updating target network.

Training Process:

The agent interacts with the environment by selecting actions based on an epsilon-greedy policy.

During each episode, the agent collects experiences and stores them in the replay memory.

After collecting a batch of experiences, the Q-network is trained using stochastic gradient descent to minimize the TD error.

The target network is periodically updated to stabilize training.

Hyperparameters:

Hyperparameters such as batch size, discount factor (gamma), epsilon decay, and target update frequency are set empirically.

Evaluation:

The performance of the trained agent is evaluated by measuring the average reward obtained over multiple episodes.

Additionally, the training process can be visualized by plotting the rewards obtained per episode over the training duration.

Results

Training Progress:

Initially, the agent's performance is poor as it explores the environment randomly.

As training progresses, the agent learns to balance the pole more effectively, leading to an increase in the average reward per episode.

Over time, the agent's performance stabilizes, indicating successful learning.

Performance Evaluation:

The trained agent achieves a high average reward, indicating successful learning and effective policy optimization.

The agent is capable of balancing the pole on the cart for extended periods, demonstrating its ability to generalize well.

Conclusion:

The experiment demonstrates the effectiveness of using a Deep Q-Network (DQN) to solve the CartPole environment.

By implementing techniques such as experience replay and target network updates, the agent learns to balance the pole effectively through interactions with the environment.

Further experimentation with different neural network architectures, hyperparameters, and reinforcement learning algorithms could potentially improve the agent's performance and robustness in more complex environments.

Code -

```
import gym

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

from collections import namedtuple


# Hyperparameters

BATCH_SIZE = 64

GAMMA = 0.99

EPS_START = 0.9

EPS_END = 0.05

EPS_DECAY = 200

TARGET_UPDATE = 10


# Define the neural network architecture

class DQN(nn.Module):

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

```
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, output_size)

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

# Define the experience replay memory
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))

class ReplayMemory(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)
```

```
    def __len__(self):
        return len(self.memory)

# Define the agent
class DQNAgent(object):
    def __init__(self, input_size, output_size):
        self.policy_net = DQN(input_size, output_size)
        self.target_net = DQN(input_size, output_size)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()
        self.optimizer = optim.Adam(self.policy_net.parameters())
        self.memory = ReplayMemory(10000)
        self.steps_done = 0
        self.input_size = input_size
        self.output_size = output_size

    def select_action(self, state):
        sample = np.random.rand()
        eps_threshold = EPS_END + (EPS_START - EPS_END) * np.exp(-1.0 * self.steps_done / EPS_DECAY)
        self.steps_done += 1
        if sample > eps_threshold:
            with torch.no_grad():
                return self.policy_net(state).max(1)[1].view(1, 1)
        else:
            return torch.tensor([[np.random.randint(self.output_size)]], dtype=torch.long)

    def optimize_model(self):
        if len(self.memory) < BATCH_SIZE:
            return
```

```
        transitions = self.memory.sample(BATCH_SIZE)
        batch = Transition(*zip(*transitions))

        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)),
dtype=torch.bool)
        non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])

        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

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

        next_state_values = torch.zeros(BATCH_SIZE)
        next_state_values[non_final_mask] = self.target_net(non_final_next_states).max(1)[0].detach()

        expected_state_action_values = (next_state_values * GAMMA) + reward_batch

        loss = nn.functional.smooth_l1_loss(state_action_values,
expected_state_action_values.unsqueeze(1))

        self.optimizer.zero_grad()
        loss.backward()
        for param in self.policy_net.parameters():
            param.grad.data.clamp_(-1, 1)
        self.optimizer.step()

# Training loop
env = gym.make('CartPole-v1')
```

```
input_size = env.observation_space.shape[0]
output_size = env.action_space.n
agent = DQNAgent(input_size, output_size)

num_episodes = 1000
for i_episode in range(num_episodes):
    state = env.reset()
    state = torch.tensor(state, dtype=torch.float32).view(1, -1)
    for t in range(1000):  # Break out of the loop if the episode is done
        action = agent.select_action(state)
        next_state, reward, done, _ = env.step(action.item())
        reward = torch.tensor([reward], dtype=torch.float32)
        next_state = torch.tensor(next_state, dtype=torch.float32).view(1, -1)

        if done:
            next_state = None

        agent.memory.push(state, action, next_state, reward)

        state = next_state

        agent.optimize_model()

        if done:
            break

    if i_episode % TARGET_UPDATE == 0:
        agent.target_net.load_state_dict(agent.policy_net.state_dict())
```

```
print('Training finished')
```