EXPERIMENT ASSESSMENT

ACADEMIC YEAR 2023-24

Course: Reinforcement Learning Lab

Course code: CSDOL8013

Year: BE **Sem:** VIII

Experiment No.: 8
Aim: - Deep Reinforcement Learning: Implementing a deep Q-network (DQN) to train an agent to play a popular Atari game, such as Pong or Space Invaders
Name:
Roll Number:
Date of Performance:
Date of Submission:

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission.	10	
Total	**20**	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	5	3	2
Understanding	5	3	2
Journal work and timely submission.	10	8	4

Checked by

Name of Faculty : Rujuta Vartak

Signature :

Date :

EXPERIMENT 8

Aim: Deep Reinforcement Learning: Implementing a deep Q-network (DQN) to train an agent to play a popular Atari game, such as Pong or Space Invaders.

Objective: The goal is to develop an AI agent capable of autonomously learning and executing effective strategies to achieve high scores and proficient gameplay in complex Atari game environments.

Theory:

Implementing a Deep Q-Network (DQN) to train an agent to play Atari games like Pong or Space Invaders is a popular application of reinforcement learning. Here's a high-level overview of how you can do this:

1. Environment Setup:
 Install the necessary libraries like Gym Atari, which provides an interface to Atari games.
 Set up your Python environment with libraries like TensorFlow or PyTorch for deep learning.

2. Understanding the Environment:
 Choose an Atari game (e.g., Pong, Space Invaders) and understand its state space, action space, and reward structure.

3. Deep Q-Network (DQN) Implementation:
 Define a neural network architecture for your DQN. Typically, a convolutional neural network (CNN) is used to process game frames.
 Implement the Q-network, which takes the game state as input and outputs Q-values for each action.
 Implement an experience replay buffer to store past experiences (state, action, reward, next state).
 Implement the epsilon-greedy strategy for action selection to balance exploration and exploitation.
 Define the loss function, typically the mean squared error between predicted Q-values and target Q values.
 Train the DQN by sampling batches of experiences from the replay buffer and updating the Q-network parameters using gradient descent.

4. Training Process:
 Initialize the DQN with random weights.
 Interact with the environment using the current policy (epsilon-greedy) and collect experiences.
 Store experiences (state, action, reward, next state) in the replay buffer.
 Sample batches of experiences from the replay buffer and perform a gradient descent step to update the Q-network parameters.
 Update the target Q-network periodically to stabilize training.

5. Evaluation:
 Periodically evaluate the performance of the trained agent by running it in the environment and measuring its average reward or other performance metrics.
 Adjust hyperparameters and tweak the algorithm as needed based on performance.

6. Fine-tuning:
 Experiment with different hyperparameters, neural network architectures, and training strategies to improve the agent's performance.
 Consider advanced techniques like double DQN, dueling DQN, prioritized experience replay, or rainbow DQN for further improvements.

7. Deployment:
 Once the agent achieves satisfactory performance, deploy it to play the game autonomously or integrate it into a larger system.

Remember that training DQN can be computationally intensive and may require significant time and computational resources, especially for complex Atari games. Be patient and monitor the training process closely to diagnose any issues and make necessary adjustments. Additionally, consider leveraging cloud computing resources or distributed training techniques to speed up the training process

Code: # Initialization

```
def __init__(self, state_dim, action_dim):
    self.state_dim = state_dim
    self.action_dim = action_dim
    self.memory = []
    self.epsilon, self.epsilon_decay, self.epsilon_min = 1.0, 0.995, 0.01
    self.gamma, self.learning_rate = 0.99, 0.001
    self.model = self._build_model()
#Building the Q- network
def _build_model(self):
    model = Sequential()
    model.add(Dense(24, input_shape=(self.state_dim,), activation='relu'))
    model.add(Dense(24, activation='relu'))
    model.add(Dense(self.action_dim, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate))
    return model
#selecting action
def select_action(self, state):
    if np.random.rand() <= self.epsilon:
        return np.random.choice(self.action_dim)
    q_values = self.model.predict(state)
    return np.argmax(q_values[0])
#storing experiances
def store_experience(self, state, action, reward, next_state, done):
    experience = (state, action, reward, next_state, done)
    self.memory.append(experience)
```

```
def create_space_invaders_agent():
    environment = gym.make('SpaceInvaders-v4', render_mode='rgb_array')
    state_space = environment.observation_space.shape
    action_space = environment.action_space.n
    agent = QLearningAgent(state_space, action_space)
    return environment, agent

environs, agent = create_space_invaders_agent()

def train_spi_agent(agent, environs, episodes):
    batch_size = 32
    for episode in range(episodes):
        state = environs.reset()
        done = False
        total_reward = 0

        while not done:
            # Choose action using epsilon-greedy policy
            if np.random.rand() < agent.epsilon:
                action = environs.action_space.sample()  # Explore: choose a
random action
            else:
                action = tf.convert_to_tensor(action) # Exploit: choose the best
action

            next_state, reward, done, _ = environs.step(action)

            # Store the experience in memory
            agent.agent_experience(state, action, reward, next_state, done)

            # Update the state
            state = next_state
            total_reward += reward

           # Apply epsilon decay
            if agent.epsilon > agent.epsilon_min:
                agent.epsilon *= agent.epsilon_decay

        print(f"Episode: {episode + 1}, Total Reward: {total_reward}, Epsilon:
{agent.epsilon}")

    # Close the environment after training
    environs.close()

# Usage:
train_spi_agent(agent, environs, episodes=30)
```

Output:

```
Episode 1, Total Reward: 180.0, Epsilon: 0.05743227569078546
Episode 2, Total Reward: 170.0, Epsilon: 0.00998645168764533
Episode 3, Total Reward: 120.0, Epsilon: 0.00998645168764533
Episode 4, Total Reward: 220.0, Epsilon: 0.00998645168764533
Episode 5, Total Reward: 190.0, Epsilon: 0.00998645168764533
Episode 6, Total Reward: 190.0, Epsilon: 0.00998645168764533
Episode 7, Total Reward: 220.0, Epsilon: 0.00998645168764533
Episode 8, Total Reward: 240.0, Epsilon: 0.00998645168764533
Episode 9, Total Reward: 260.0, Epsilon: 0.00998645168764533
Episode 10, Total Reward: 270.0, Epsilon: 0.00998645168764533
Episode 11, Total Reward: 300.0, Epsilon: 0.00998645168764533
Episode 12, Total Reward: 230.0, Epsilon: 0.00998645168764533
Episode 13, Total Reward: 250.0, Epsilon: 0.00998645168764533
Episode 14, Total Reward: 260.0, Epsilon: 0.00998645168764533
Episode 15, Total Reward: 180.0, Epsilon: 0.00998645168764533
Episode 16, Total Reward: 280.0, Epsilon: 0.00998645168764533
Episode 17, Total Reward: 1080.0, Epsilon: 0.00998645168764533
Episode 18, Total Reward: 200.0, Epsilon: 0.00998645168764533
Episode 19, Total Reward: 230.0, Epsilon: 0.00998645168764533
Episode 20, Total Reward: 230.0, Epsilon: 0.00998645168764533
Episode 21, Total Reward: 160.0, Epsilon: 0.00998645168764533
Episode 22, Total Reward: 240.0, Epsilon: 0.00998645168764533
Episode 23, Total Reward: 220.0, Epsilon: 0.00998645168764533
Episode 24, Total Reward: 1080.0, Epsilon: 0.00998645168764533
Episode 25, Total Reward: 240.0, Epsilon: 0.00998645168764533
Episode 26, Total Reward: 200.0, Epsilon: 0.00998645168764533
Episode 27, Total Reward: 250.0, Epsilon: 0.00998645168764533
Episode 28, Total Reward: 290.0, Epsilon: 0.00998645168764533
Episode 29, Total Reward: 190.0, Epsilon: 0.00998645168764533
Episode 30, Total Reward: 230.0, Epsilon: 0.00998645168764533
```

Conclusion:

1. Explain role deep Q-network (DQN) in Reinforcement Learning

Deep Q-Network (DQN) is a type of neural network architecture used in reinforcement learning (RL) to approximate the Q-function, which represents the expected cumulative reward of taking a particular action in a given state and following a certain policy thereafter. DQN is a significant advancement in RL, particularly in solving complex tasks with high-dimensional state spaces, such as video games or robotics.

2**. Explain Working of Atari game.**

Atari games are classic arcade video games developed and released by Atari, Inc. during the 1970s and 1980s. Atari games are a classic benchmark environment in reinforcement learning (RL) research. These games provide a challenging and diverse set of tasks for RL agents to learn from.

By interacting with the game environment, receiving feedback in the form of observations and rewards, and updating its policy based on reinforcement learning algorithms, the RL agent learns to play Atari games at a human or superhuman level of performance. The working of Atari games within an RL framework demonstrates the capability of RL algorithms to learn complex behaviors and strategies in diverse and challenging environments.

the working of an Atari game involves a combination of input handling, game state management, rendering, audio output, and player interaction to create an engaging and immersive gaming experience. While the specific implementation details may vary between different games and platforms, these core components form the foundation of how Atari games operate.

Conclusion: