**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management & Engineering**

**Department of Aritificial Intelligence**

Program: B.Tech/MBA Tech AI Semester: VII

**Course: Reinforcement Learning**

**List of Experiments**

**Faculty:** Dr. Ami Munshi W.E.F. July 2024

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp No.** | **Title** | **Prerequisite\*** | **CO#** |
| 1 | **Introduction to RL terminologies and Elements using OpenAI Gym**   1. Explore various environments in Open AI Gym 2. Relate RL terminologies and elements with the examples in Gym Environment 3. Implement any Gym Environment from classic control 4. Customize the above gym environment parameters 5. Demonstrate how to interpret state transitions, rewards, and termination conditions with reference to the above example | Python programming  RL terminologies | 1 |
| 2. | **Implementation of Multi-Arm Bandit (MAB) Problem**   1. To implement MAB problem using pure exploitation algorithm. 2. To implement MAB problem using pure exploration algorithm 3. To implement MAB problem using Fixed Exploration followed by Exploitation 4. To implement MAB problem using Greedy algorithm 5. To apply Upper Confidence Bound(UCB) in the implementation done in part d 6. To analyse the algorithms and compare them in terms of rewards, regrets and complexity | Concept of Multi-Arm Bandit Problem in RL | 1,2 |
| 3. | **Understand and apply the concepts of Markov Decision Processes (MDPs) by modeling a real-world decision-making scenario**   1. Model decision making process 2. Implement value iteration algorithm 3. Identify the optimal policy by analysing the results | Concept of Markov Property, Markov Chain, Markov Decision Process | 2,3 |
| 4 | **Dynamic Programming- Policy Iteration Algorithm for MDP**   1. To find the optimal policy that maximizes the expected cumulative reward for a given MDP by iteratively improving an initial policy until convergence 2. To analyse the impact of discount factors for myopic and farsighted agents | Policy Iteration |  |
| 5 | **Dynamic Programming- Value Iteration Algorithm for MDP**   1. To compute the optimal policy by iteratively improving the value function for each state and selecting the action that maximizes the expected return 2. To analyse the impact of discount factors for myopic and farsighted agents 3. To compare value iteration and policy iteration algorithms in terms of time and computational complexity | Value Iteration |  |
| 6 | **Monte-Carlo Prediction**   1. To implement First Visit Monte Carlo Prediction Algorithm 2. To implement Every Visit Monte Carlo Prediction Algorithm 3. To compare First Visit Algorithm and Every Visit Algorithm on the basis of following points    * Time complexity    * Value Table for each Episode |  |  |
| 7 | **Temporal Difference Prediction**   1. To implement Temporal Difference (0) in a complex grid world environment 2. To implement Temporal Difference (1) in a complex grid world environment 3. To compare Temporal Difference Prediction Algorithm with Monte-Carlo Prediction Algorithm |  |  |
| 8 | **Q Learning and SARSA**   1. To implement Q Learning Algorithm in a complex grid world environment 2. To implement SARSA Algorithm in a complex grid world environment 3. To compare Q learning and SARSA Algorithm | On policy and Off Policy |  |
| 9 | To Implement a simple **linear value function approximator** for a given state in an RL environment. The linear approximator will estimate the state value by computing a weighted sum of the features of the state |  |  |
| 10 |  |  |  |
| 11 |  |  |  |

\* Students are expected to be ready with the prerequisite before attending the lab

**Experiment No.09**

PART A

(PART A: TO BE REFFERED BY STUDENTS)

**A.1 Aim:**  Linear Function Approximation

To Implement a simple **linear value function approximator** for a given state in an RL environment. The linear approximator will estimate the state value by computing a weighted sum of the features of the state.

**A.2 Prerequisite:**

Concept of Value Function Approximation

**A.3 Learning Outcome:**

After completing this experiment you will be able to-

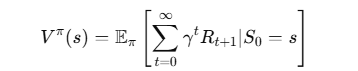
* Comprehend Value Function Approximation

**A.4 Theory:**

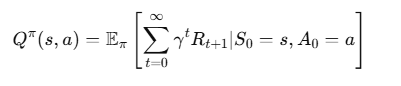
**A.4.1 Value Function Approximation**

Value function approximation is a fundamental concept in reinforcement learning (RL) that enables RL algorithms to scale to large or continuous state and action spaces. When it is infeasible to compute or store the exact value of each state (or state-action pair) in large environments, **value function approximation** provides a way to estimate values using parametric or non-parametric functions.

**State-Value Function (V)**: Estimates the expected return (cumulative future reward) when starting in state s and following a policy π

****

**Action-Value Function (Q)**: Estimates the expected return starting from state s, taking action a, and thereafter following policy π

****

**A.4.2 Steps for Linear Approximation**

1. Define a state feature vector

2. Use weights to approximate the value of state as

3. Use sample based update rule to update

**A.5 Task to be completed:**

**Implement Linear Function Approximation**:

**References**

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction.

**PART B**

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Portal/MS Teams assignment link at the end of the practical)**

|  |  |
| --- | --- |
| Roll No. C052 | Name: Drumil Kotecha |
| Program : BTI | Division: B |
| Batch: B2 | Date of Experiment: 02-04-25 |
| Date of Submission: 05-04-25 | Grade : |

**B.1 Tasks given in PART A to be completed here**

**Linear Function Approximation for Value Prediction (TD(0) style):**

import numpy as np

# Simple environment with states from 1 to 5 (states 0 and 6 are terminal)

N\_STATES = 7

TERMINAL\_LEFT = 0

TERMINAL\_RIGHT = 6

ALPHA = 0.01  # Learning rate

GAMMA = 1.0   # Discount factor

EPISODES = 100

# Feature representation for each state (one-hot encoding for simplicity)

def one\_hot(state, size=N\_STATES):

    vec = np.zeros(size)

    vec[state] = 1

    return vec

# TD(0) with linear function approximation

def td\_zero\_linear\_approximation():

    w = np.zeros(N\_STATES)  # Weight vector for linear function approximation

    for episode in range(EPISODES):

        state = 3  # Start from middle state

        while state != TERMINAL\_LEFT and state != TERMINAL\_RIGHT:

            x = one\_hot(state)

            # Take a random action (left or right)

            action = np.random.choice([-1, 1])

            next\_state = state + action

            reward = 0

            if next\_state == TERMINAL\_RIGHT:

                reward = 1

            x\_next = one\_hot(next\_state)

            # TD target and TD error

            v\_hat = np.dot(w, x)

            v\_hat\_next = np.dot(w, x\_next)

            delta = reward + GAMMA \* v\_hat\_next - v\_hat

            # Update weights

            w += ALPHA \* delta \* x

            state = next\_state

    return w

# Train and show weights

weights = td\_zero\_linear\_approximation()

print("Learned weights (approximate value function):")

for s in range(1, 6):

    print(f"V({s}) ≈ {weights[s]:.3f}")

**Semi-gradient SARSA with Linear Function Approximation on Cart Pole**

import gym

import numpy as np

import matplotlib.pyplot as plt

import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

# Hyperparameters

alpha = 0.01

gamma = 0.99

epsilon = 0.1

episodes = 500

env = gym.make("CartPole-v1")

n\_actions = env.action\_space.n

n\_features = env.observation\_space.shape[0]

# Feature vector for (state, action)

def featurize\_state\_action(state, action):

    state = np.asarray(state)

    features = np.zeros(n\_features \* n\_actions)

    start = action \* n\_features

    features[start:start + n\_features] = state

    return np.append(features, 1.0)  # Add bias

# ε-greedy action selection

def select\_action(state, w):

    if np.random.rand() < epsilon:

        return np.random.choice(n\_actions)

    q\_values = [np.dot(w, featurize\_state\_action(state, a)) for a in range(n\_actions)]

    return np.argmax(q\_values)

# Training function with reward tracking

def train():

    w = np.zeros(n\_features \* n\_actions + 1)

    reward\_per\_episode = []

    for ep in range(episodes):

        state = env.reset()[0]

        action = select\_action(state, w)

        total\_reward = 0

        done = False

        while not done:

            next\_state, reward, terminated, truncated, \_ = env.step(action)

            done = terminated or truncated

            total\_reward += reward

            next\_action = select\_action(next\_state, w)

            x = featurize\_state\_action(state, action)

            x\_next = featurize\_state\_action(next\_state, next\_action)

            q = np.dot(w, x)

            q\_next = np.dot(w, x\_next) if not done else 0

            td\_error = reward + gamma \* q\_next - q

            w += alpha \* td\_error \* x

            state = next\_state

            action = next\_action

        reward\_per\_episode.append(total\_reward)

    return w, reward\_per\_episode

# Print function for weights

def print\_weights(w, n\_features, n\_actions):

    print("\n=== Learned Weights ===")

    for a in range(n\_actions):

        start = a \* n\_features

        end = start + n\_features

        print(f"Action {a} weights: {w[start:end]}")

    print(f"Bias term: {w[-1]}")

# Run training and plot results

weights, rewards = train()

print\_weights(weights, n\_features, n\_actions)

# Plotting the reward per episode

plt.figure(figsize=(10, 5))

plt.plot(rewards, label="Total Reward per Episode")

plt.xlabel("Episode")

plt.ylabel("Total Reward")

plt.title("Learning Progress of Linear Q-Approximation on CartPole")

plt.grid(True)

plt.legend()

plt.tight\_layout()

plt.show()

**B.2 Observations and Learning:**

**Output for linear Function Approximation for Value Prediction (TD(0) style):**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Semi-gradient SARSA** **with Linear Function Approximation on Cart Pole:**

**A computer screen with numbers and letters

AI-generated content may be incorrect.**

**A screen shot of a graph

AI-generated content may be incorrect.**

**B.3 Conclusion:**

## This code demonstrates how an agent can learn to play the Cart Pole game using a simple linear model. By updating weights through trial and error, the agent gets better over time. The reward graph proves that learning is happening. Even with a basic setup, the agent learns to keep the pole balanced quite well using linear function approximation.