
# D.Y. Patil International University, Akurdi, Pune

## School of Computer Science Engineering and Applications (SoCSEA)

### Final Year Engineering (B.Tech)

### Reinforcement Learning for Agentic AI (AIM4001)

# Lab Manual

D Y PATIL
INTERNATIONAL
UNIVERSITY
AKURDI PUNE

## *Vision of the University:*

*"To Create a vibrant learning environment – fostering innovation and creativity, experiential learning, which is inspired by research, and focuses on regionally, nationally and globally relevant areas."*

## *Mission of the School:*

*To provide a diverse, vibrant and inspirational learning environment.*

*To establish the university as a leading experiential learning and research oriented center.*

*To become a responsive university serving the needs of industry and society.*

*To embed internationalization, employability and value thinking*

# Reinforcement Learning for Agentic AI

## Course Objectives:

To introduce the foundational concepts of Reinforcement Learning (RL) and the agent-environment.
To develop skills in implementing model-free prediction and control algorithms.
To explore deep reinforcement learning techniques and function approximation approaches.
To expose students to cutting-edge research in multi-agent systems, RL with human feedback, and the responsible design of agentic AI systems.

## Course Outcomes:

On completion of the course, learner will be able to–

**CO1:** Explain the principles of reinforcement learning and understand different agent architectures and learning paradigms.
**CO2:** Implement and analyze model-free RL algorithms.
**CO3:** Apply function approximation techniques and deep learning models.
**CO4:** Critically study modern reinforcement learning systems and apply RL techniques to real-world agentic AI applications.

## Program Outcomes:

| | |
|---|---|
| **PO1** | **Engineering knowledge:**<br>Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems. |
| **PO2** | **Problem analysis**:<br>Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences. |
| **PO3** | **Design/development of solutions**:<br>Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations. |
| **PO4** | **Conduct investigations of complex problems**:<br>Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. |
| **PO5** | **Modern tool usage**:<br>Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations. |
| **PO6** | **The engineer and society**:<br>Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice. |
| **PO7** | **Environment and sustainability**:<br>Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development. |
| **PO8** | **Ethics**:<br>Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. |
| **PO9** | **Individual and team work**:<br>Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings. |
| **PO10** | **Communication**:<br>Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions. |
| **PO11** | **Project management and finance**: |

| | |
|---|---|
| | Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments. |
| **PO12** | **Life-long learning**:<br>Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change. |

## Rules and Regulations for Laboratory:

- Students should be regular and punctual to all the Lab practical
- Lab assignments and practicals should be submitted within a given time.
- Mobile phones are strictly prohibited in the Lab.
- Please shut down the Computers before leaving the Lab.
- Please switch off the fans and lights and keep the chair in proper position before leaving the Lab
- Maintain proper discipline in the Lab.

# D Y Patil International University, Akurdi, Pune
## School of Computer Science Engineering and Applications

Index

| Sr. No. | Name of the Practical | Date of Conducting | Page No. | | Sign of Teacher | Remarks* |
|---|---|---|---|---|---|---|
| | | | From | To | | |
| 1. | Policy Evaluation using Monte Carlo (MC) methods. | | | | | |
| 2. | Implement Temporal Difference (TD(0), TD(ʎ)) Learning. | | | | | |
| 3. | SARSA vs. Q-Learning with €-greedy and Softmax Exploration Strategies. | | | | | |
| 4. | Implementation of Deep Q-Network (DQN). | | | | | |
| 5. | Implement REINFORCE with Baseline and Advantage Functions using PyTorch/ TensorFlow. | | | | | |
| 6. | Implement Actor-Critic method for continuous control tasks. | | | | | |
| 7. | Multi-agent RL scenario in GridWorld. | | | | | |

***Absent/Attended/Late/Partially Completed/Completed**

**CERTIFICATE**

This is to certify that **Mr. Kiran Biradar PRN: 20220802048** of class: B.Tech(CSE) has completed practical/term work in the course of RL for Agentic AI of Final Year Engineering (B.Tech) within SoCSEA, as prescribed D Y Patil International University, Pune during the academic year 2024 -2025.

**Date:** **Teaching Assistant** **Faculty I/C** **Director** (SCSEA)

# Practical 01

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

**Aim:**

Policy Evaluation using Monte Carlo (MC) methods.

**Objectives:**

- Implement a **Grid World Environment:** Create a simulation of a grid-based world with obstacles, a starting point, and a goal, where an agent can move and receive rewards or penalties.

- Develop a **Q-Learning Agent:** Build an agent that learns to navigate the grid world using the Q-learning algorithm, learning action-values (Q-values) to maximize cumulative rewards.

- **Train the Agent:** Train the Q-learning agent within the Grid World environment for a specified number of episodes to allow it to learn an optimal policy.

- **Analyze Training Progress:** Visualize and evaluate the agent's learning progress by tracking metrics like rewards per episode and steps taken to reach the goal.

- **Test the Trained Agent:** Evaluate the performance of the trained agent by having it navigate the grid world using its learned policy and visualizing the path taken.

- **Visualize Learned Q-values:** Display the learned Q-values for different states in the grid world to understand what the agent has learned about the value of different actions in different locations.

**Software/Tool:** Google Colab / Jupyter

**Theory:**

This experiment demonstrates **Reinforcement Learning (RL)** — a branch of machine learning where an agent learns to make sequential decisions by interacting with an environment to maximize a **cumulative reward**.

The environment is modeled as a **Markov Decision Process (MDP)** defined as:

$$\text{MDP} = (S, A, P, R, \gamma)$$

where:

- $S$: Set of states
- $A$: Set of actions
- $P(s'|s, a)$: Transition probability
- $R(s, a)$: Reward function
- $\gamma$: Discount factor

The goal is to learn a **policy** $\pi(a|s)$ that maximizes the **expected discounted return**:

### 1. Monte Carlo (MC) Policy Evaluation

Monte Carlo methods estimate the **value function** $V^\pi(s)$ or **action-value function** $Q^\pi(s, a)$ by averaging sample returns from complete episodes:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

**Incremental MC update rule:**

$$Q(s, a) \leftarrow Q(s, a) + \alpha(G_t - Q(s, a))$$

Here $\alpha$ is the learning rate controlling how much new experiences affect existing estimates.

### 2. Q-Learning (Off-Policy Temporal Difference Control)

Q-Learning is a model-free, off-policy RL algorithm that updates Q-values based on the **Bellman Equation**:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$$

- $R_{t+1}$: Reward after action $a_t$
- $\gamma$: Discount factor
- $\alpha$: Learning rate
- $\max_{a'} Q(s_{t+1}, a')$: Estimated best next action value

### 3. Exploration vs. Exploitation

The agent uses an **ε-greedy policy** to balance exploration and exploitation:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s,a), & \text{with probability } 1-\epsilon \end{cases}$$

As training progresses, $\epsilon$ gradually decreases to favor exploitation of learned knowledge.

### 4. Grid World Environment

- Each **cell** in the grid = a **state**.
- **Start (S)** → Beginning point
- **Goal (G)** → Rewarding state
- **Obstacles (X)** → Penalizing or forbidden states
- **Actions:** Up, Down, Left, Right

Rewards:

- ( +1 ) for reaching the goal
- ( -1 ) for hitting obstacles or invalid moves
- ( 0 ) for regular transitions

### 5. Learning Process and Convergence

During training, the agent interacts with the environment and updates its Q-values using the Bellman equation.
Over many episodes, the values converge towards the **optimal Q-function**:

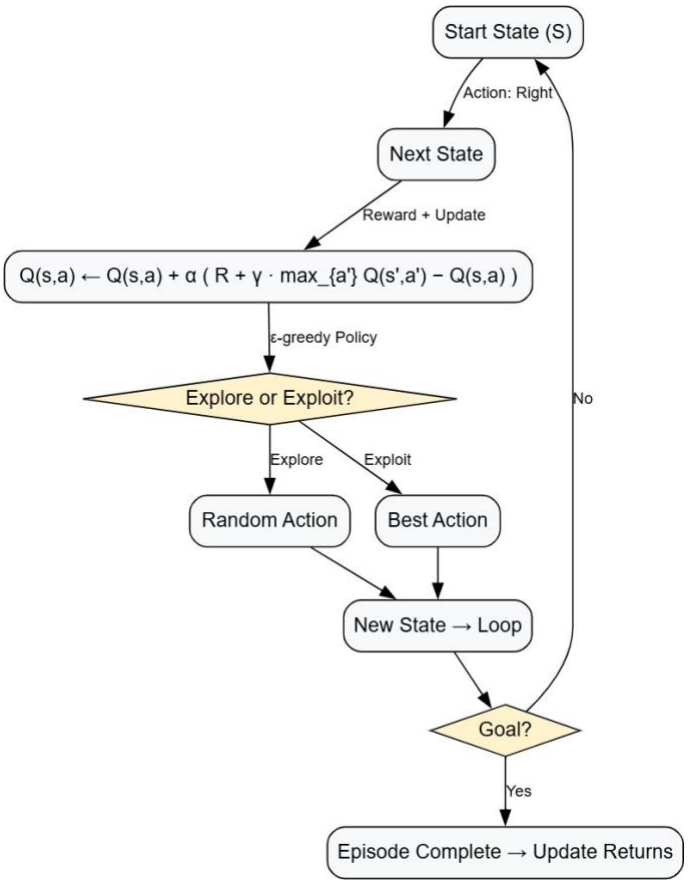
Over many episodes, the values converge towards the **optimal Q-function**:

$$\lim_{t\to\infty} Q(s,a) = Q^*(s,a)$$

and the agent learns the **optimal policy**:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

**7. Diagram: Q-Learning in Grid World**




**Attachment:** Code and Result

```
# Name – Kiran Biradar
# PRN – 20220802048
#
# Simple MDP simulation in a 4x4 GridWorld for Markov Decesion Process(MDP)
#
# - Terminal states: +1 goal at (3,3), -1 pit at (3,0)
# - Step cost: -0.04, slip probability: 0.1 to a random perpendicular action
# - Actions: 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT

from __future__ import annotations
import numpy as np
from dataclasses import dataclass
from typing import Tuple, List, Dict, Callable

UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
ACTIONS = [UP, RIGHT, DOWN, LEFT]
ACTION_TO_STR = {UP:"↑", RIGHT:"→", DOWN:"↓", LEFT:"←"}
```

```
w
@dataclass(frozen=True)
class GridWorldConfig:
    rows: int = 4
    cols: int = 4
    terminal_states: Dict[Tuple[int,int], float] = None
    step_cost: float = -0.04
    slip_prob: float = 0.1   # slip to a perpendicular action with this prob
    gamma: float = 0.99

    def __post_init__(self):
        if self.terminal_states is None:
            object.__setattr__(self, "terminal_states", {(3,3): +1.0, (3,0): -1.0})

class GridWorldMDP:
    def __init__(self, cfg: GridWorldConfig):
        self.cfg = cfg
        self.rows = cfg.rows
        self.cols = cfg.cols
        self.S = [(r,c) for r in range(self.rows) for c in range(self.cols)]
        self.terminal = set(cfg.terminal_states.keys())

    def is_terminal(self, s: Tuple[int,int]) -> bool:
        return s in self.terminal

    def in_bounds(self, r, c) -> Tuple[int,int]:
        r = min(max(r, 0), self.rows-1)
        c = min(max(c, 0), self.cols-1)
        return (r, c)

    def move(self, s: Tuple[int,int], a: int) -> Tuple[int,int]:
        r, c = s
        if a == UP:    r -= 1
        if a == DOWN:  r += 1
        if a == LEFT:  c -= 1
        if a == RIGHT: c += 1
        return self.in_bounds(r, c)

    def perpendicular_actions(self, a: int) -> List[int]:
        if a in [UP, DOWN]:
            return [LEFT, RIGHT]
        else:
            return [UP, DOWN]

    # Stochastic transition model
    def transitions(self, s: Tuple[int,int], a: int):
        """Return list of (prob, next_state, reward) tuples."""
        if self.is_terminal(s):
            # absorbing terminal with zero further reward
            return [(1.0, s, 0.0)]
```

```
        p_main = 1.0 - self.cfg.slip_prob
        slip_each = self.cfg.slip_prob / 2.0
        outcomes = []
        # intended move
        s1 = self.move(s, a)
        outcomes.append((p_main, s1, self.reward(s, a, s1)))
        # slips
        for a_perp in self.perpendicular_actions(a):
            sp = self.move(s, a_perp)
            outcomes.append((slip_each, sp, self.reward(s, a_perp, sp)))
        return outcomes

    def reward(self, s: Tuple[int,int], a: int, s1: Tuple[int,int]) -> float:
        if s1 in self.cfg.terminal_states:
            return self.cfg.terminal_states[s1]
        return self.cfg.step_cost

    # One step environment interaction (sampled)
    def step(self, s: Tuple[int,int], a: int) -> Tuple[Tuple[int,int], float, bool]:
        probs, next_states, rewards = zip(*self.transitions(s, a))
        idx = np.random.choice(len(probs), p=np.array(probs))
        ns, r = next_states[idx], rewards[idx]
        done = self.is_terminal(ns)
        return ns, r, done

    def reset(self, start: Tuple[int,int] = (0,0)) -> Tuple[int,int]:
        return start

# ----- Planning: Value Iteration -----
def value_iteration(env: GridWorldMDP, theta: float = 1e-6, max_iters: int = 10_000):
    V = {s: 0.0 for s in env.S}
    for s in env.terminal:
        V[s] = env.cfg.terminal_states[s]

    def q_value(s, a):
        return sum(p * (r + env.cfg.gamma * V[s1]) for p, s1, r in env.transitions(s, a))

    iters = 0
    while True:
        delta = 0.0
        iters += 1
        for s in env.S:
            if env.is_terminal(s):
                continue
            v = V[s]
            V[s] = max(q_value(s, a) for a in ACTIONS)
            delta = max(delta, abs(v - V[s]))
        if delta < theta or iters >= max_iters:
            break
```

```
    # Derive greedy policy
    Pi = {}
    for s in env.S:
        if env.is_terminal(s):
            Pi[s] = None
            continue
        qs = [sum(p * (r + env.cfg.gamma * V[s1]) for p, s1, r in env.transitions(s, a)) for
a in ACTIONS]
        Pi[s] = int(np.argmax(qs))
    return V, Pi, iters

# ----- Control: ε-greedy policy + Q-learning -----
def epsilon_greedy(Q: Dict[Tuple[Tuple[int,int], int], float], s, eps: float):
    if np.random.rand() < eps:
        return np.random.choice(ACTIONS)
    qvals = [Q.get((s, a), 0.0) for a in ACTIONS]
    return int(np.argmax(qvals))

def q_learning(env: GridWorldMDP, episodes=500, alpha=0.5, gamma=None, eps_start=1.0,
eps_end=0.05):
    if gamma is None:
        gamma = env.cfg.gamma
    Q = {}
    returns = []

    for ep in range(episodes):
        s = env.reset()
        done = env.is_terminal(s)
        eps = eps_end + (eps_start - eps_end) * np.exp(-3.0 * ep / episodes)  # smooth decay

        total = 0.0
        steps = 0
        while not done and steps < 200:
            a = epsilon_greedy(Q, s, eps)
            s1, r, done = env.step(s, a)
            total += r

            # Q-learning update
            q_sa = Q.get((s, a), 0.0)
            max_q_next = max(Q.get((s1, ap), 0.0) for ap in ACTIONS) if not
env.is_terminal(s1) else 0.0
            target = r + gamma * max_q_next
            Q[(s, a)] = q_sa + alpha * (target - q_sa)

            s = s1
            steps += 1
        returns.append(total)
    return Q, returns
```

```python
# ----- Utility: pretty printing -----
def print_values(V, env: GridWorldMDP):
    print("State values (higher is better):")
    for r in range(env.rows):
        row = []
        for c in range(env.cols):
            v = V[(r,c)]
            row.append(f"{v:6.2f}")
        print(" ".join(row))
    print()

def print_policy(Pi, env: GridWorldMDP):
    print("Policy (arrows), T=terminal:")
    for r in range(env.rows):
        row = []
        for c in range(env.cols):
            s = (r,c)
            if s in env.terminal:
                row.append(" T ")
            else:
                a = Pi[s]
                row.append(f" {ACTION_TO_STR[a]} ")
        print(" ".join(row))
    print()

def greedy_policy_from_Q(Q, env: GridWorldMDP):
    Pi = {}
    for s in env.S:
        if env.is_terminal(s):
            Pi[s] = None
        else:
            q = [Q.get((s, a), 0.0) for a in ACTIONS]
            Pi[s] = int(np.argmax(q))
    return Pi

def simulate_episode(env: GridWorldMDP, policy: Callable[[Tuple[int,int]], int], start=(0,0),
max_steps=50):
    s = env.reset(start)
    traj = [s]
    total = 0.0
    for _ in range(max_steps):
        if env.is_terminal(s):
            break
        a = policy(s)
        s, r, done = env.step(s, a)
        total += r
        traj.append(s)
        if done:
            break
    return traj, total
```

```python
# ----- Run demo -----
if __name__ == "__main__":
    np.random.seed(0)
    cfg = GridWorldConfig()
    env = GridWorldMDP(cfg)

    # 1) Planning with Value Iteration (knows the model)
    V_opt, Pi_opt, iters = value_iteration(env)
    print(f"Value Iteration converged in {iters} iterations.\n")
    print_values(V_opt, env)
    print_policy(Pi_opt, env)

    # 2) Roll out the optimal policy from (0,0)
    traj, ret = simulate_episode(env, lambda s: Pi_opt[s])
    print("Rollout under optimal policy:")
    print("Trajectory:", traj)
    print(f"Return: {ret:.2f}\n")

    # 3) Learn from interaction with Q-learning (unknown model)
    Q, returns = q_learning(env, episodes=1000, alpha=0.5)
    Pi_learned = greedy_policy_from_Q(Q, env)
    print("Learned policy via Q-learning (approximate):")
    print_policy(Pi_learned, env)

    # 4) Compare a single rollout with learned policy
    traj2, ret2 = simulate_episode(env, lambda s: Pi_learned[s])
    print("Rollout under learned policy:")
    print("Trajectory:", traj2)
    print(f"Return: {ret2:.2f}")

    # 5) (Optional) Quick stats on learning
    print("\nAverage return over last 100 episodes:", np.mean(returns[-100:]).round(3))
```

```
Value Iteration converged in 15 iterations.

State values (higher is better):
  1.66   1.72   1.78   1.84
  1.71   1.78   1.85   1.91
  1.66   1.85   1.91   1.98
 -1.00   1.91   1.98   1.00

Policy (arrows), T=terminal:
  →     →     ↓     ↓
  →     →     ↓     ↓
  ↑     →     →     ↓
  T     →     →     T

Rollout under optimal policy:
Trajectory: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
Return: 0.80

Learned policy via Q-learning (approximate):
Policy (arrows), T=terminal:
  ↓     →     ↓     ↓
  →     →     ↓     ↓
  ↑     ↓     →     ↓
  T     →     →     T
...
Trajectory: [(0, 0), (1, 0), (1, 1), (1, 2), (1, 1), (1, 2), (2, 2), (1, 2), (2, 2), (2, 3), (3, 3)]
Return: 0.64
```

## Conclusion:

# Practical 02

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

## Aim:

To implement and analyze Temporal Difference (TD) Learning algorithms, specifically TD(0) and TD(λ), for value function estimation in reinforcement learning.

## Objectives:

- To understand the concept of **Temporal Difference (TD)** learning.
- To implement **TD(0)** learning for value prediction.
- To extend the implementation to **TD(λ)** learning using eligibility traces.
- To compare the performance of **TD(0)** and **TD(λ)** in terms of convergence.
- To observe the role of parameter **λ** in balancing the bias–variance trade-off.

## Software/Tool:

Google Colab

## Theory:

**Temporal Difference (TD) Learning** is a foundational concept in Reinforcement Learning (RL) that combines ideas from **Monte Carlo methods** and **Dynamic Programming (DP)**. Like Monte Carlo methods, TD learning learns directly from **experience** without requiring a model of the environment, and like DP, it performs **bootstrapping** updating estimates based partly on other learned estimates without waiting for the final outcome of an episode.

### 1. TD Learning Concept

In TD learning, the agent estimates the **value function** $V(s)$, which represents the expected return from state $s$ while following a policy $\pi$:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

Instead of waiting until the end of an episode (as in Monte Carlo), TD learning updates its estimate **after each time step**, using the **TD target** and **TD error**.

**2. TD(0): One-Step Temporal Difference Learning**

**TD(0)** is the simplest form of TD learning. It updates the value of the current state immediately after observing the next state and reward.

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$$

Where:

- $V(s)$: Estimated value of current state $s$
- $\alpha$: Learning rate ($0 < \alpha \leq 1$)
- $r$: Immediate reward after taking an action
- $\gamma$: Discount factor ($0 \leq \gamma \leq 1$)
- $V(s')$: Estimated value of the next state

The **TD Target** is:

$$\text{Target} = r + \gamma V(s')$$

The **TD Error** is:

$$\delta = r + \gamma V(s') - V(s)$$

The update rule can thus be expressed as:

$$V(s) \leftarrow V(s) + \alpha\delta$$

**Intuition:**

- The agent compares the predicted value $V(s)$ with a better estimate $r + \gamma V(s')$.
- The error $\delta$ measures how much correction is needed.
- $V(s)$ moves a small step (scaled by $\alpha$) towards the TD target.

**3. TD(λ): Multi-Step Temporal Difference Learning**

**TD(λ)** generalizes TD(0) by incorporating information from **multiple future steps** using **eligibility traces**.

This allows learning to consider not only the immediate next state but also future rewards and states.

The update rule for TD(λ) is:

$$V(s) \leftarrow V(s) + \alpha \delta e(s)$$

where

$$\delta = r + \gamma V(s') - V(s)$$

Here, $e(s)$ is the **eligibility trace**, representing how recently and frequently the state $s$ has been visited.

Here, ( e(s) ) is the **eligibility trace**, representing how recently and frequently the state ( s ) has been visited.

**4. Eligibility Traces**

Eligibility traces keep a decaying memory of previously visited states.

When a state $s$ is visited, its trace increases, and with each step, traces decay exponentially by a factor of $\gamma\lambda$:

$$e(s) \leftarrow \gamma\lambda e(s) + 1(s = \text{current state})$$

Where $1(s = \text{current state}) = 1$ if the state $s$ is currently visited, otherwise 0.

This mechanism allows TD(λ) to distribute the TD error back to previously visited states, making learning faster and smoother.

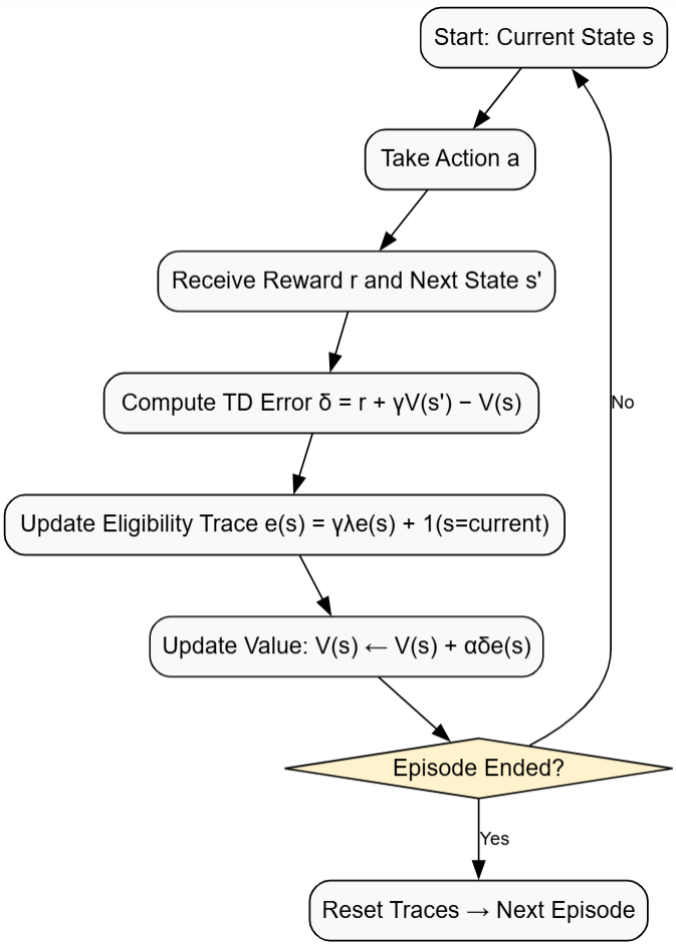
**5. Bias–Variance Trade-off**

The **parameter λ** determines the balance between **bias** and **variance**:

- Small λ (close to 0) → More bias, less variance (faster updates, short-term corrections).
- Large λ (close to 1) → Less bias, more variance (more accurate but slower updates).

This makes TD(λ) a flexible framework bridging the extremes of TD(0) and Monte Carlo methods.

### 8. Diagram: Flow of TD(λ) Learning




**Attachment:** Code and Results

```
#
# Name: Kiran Biradar
# PRN: 20220802048
#
#Implement Temporal Difference (TD(0), TD()) Learning.
import numpy as np
from collections import defaultdict

# --- Simple 4x4 GridWorld ---
class GridWorld:
    def __init__(self, rows=4, cols=4, terminal_states=[(0,0), (3,3)]):
        self.rows = rows
        self.cols = cols
        self.terminal_states = set(terminal_states)
        self.reset()
```

```
    def reset(self):
        while True:
            r, c = np.random.randint(self.rows), np.random.randint(self.cols)
            if (r,c) not in self.terminal_states:
                self.state = (r,c)
                break
        return self.state

    def step(self, action):
        """Actions: 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT"""
        r, c = self.state
        if self.state in self.terminal_states:
            return self.state, 0, True

        if action == 0:   r = max(r-1, 0)
        elif action == 1: c = min(c+1, self.cols-1)
        elif action == 2: r = min(r+1, self.rows-1)
        elif action == 3: c = max(c-1, 0)

        self.state = (r, c)
        reward = 0 if self.state in self.terminal_states else -1
        done = self.state in self.terminal_states
        return self.state, reward, done

# --- Policy: Random uniform ---
def random_policy(state):
    return np.random.choice([0,1,2,3])

# --- TD(0) Policy Evaluation ---
def td0_policy_evaluation(env, policy, episodes=5000, alpha=0.1, gamma=1.0):
    V = defaultdict(float)
    for _ in range(episodes):
        s = env.reset()
        done = False
        while not done:
            a = policy(s)
            s_next, r, done = env.step(a)
            # TD(0) update
            V[s] += alpha * (r + gamma * V[s_next] - V[s])
            s = s_next
    return V

# --- TD(λ) with eligibility traces ---
def td_lambda_policy_evaluation(env, policy, episodes=5000, alpha=0.1, gamma=1.0, lam=0.9):
    V = defaultdict(float)
    for _ in range(episodes):
        s = env.reset()
        E = defaultdict(float)  # eligibility trace
        done = False
```

```
        while not done:
            a = policy(s)
            s_next, r, done = env.step(a)
            delta = r + gamma * V[s_next] - V[s]
            E[s] += 1  # increment eligibility for current state
            for s_ in list(V.keys()):
                V[s_] += alpha * delta * E[s_]
                E[s_] *= gamma * lam  # decay
            s = s_next
    return V

# --- Run Example ---
if __name__ == "__main__":
    env = GridWorld()

    # TD(0)
    V_td0 = td0_policy_evaluation(env, random_policy, episodes=5000)
    print("\nValue Function from TD(0):\n")
    for r in range(env.rows):
        row = []
        for c in range(env.cols):
            row.append(f"{V_td0[(r,c)]:6.2f}")
        print(" ".join(row))

    # TD(λ)
    V_tdl = td_lambda_policy_evaluation(env, random_policy, episodes=5000, lam=0.9)
    print("\nValue Function from TD(λ=0.9):\n")
    for r in range(env.rows):
        row = []
        for c in range(env.cols):
            row.append(f"{V_tdl[(r,c)]:6.2f}")
        print(" ".join(row))
```

```
Value Function from TD(0):

  0.00  -9.54 -18.07 -19.06
 -9.55 -14.80 -17.11 -16.13
-16.36 -16.86 -14.28  -8.53
-19.19 -16.85  -7.67   0.00

Value Function from TD(λ=0.9):

  0.00 -10.49 -16.27 -14.67
 -6.32  -9.51 -17.77 -14.92
-12.75 -13.62 -15.01 -13.39
-18.21 -19.91 -14.45   0.00
```

**Conclusion:**

# Practical 03

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

**Aim:**

Implement, compare, and visually analyze two fundamental reinforcement learning algorithms — **SARSA (On-Policy)** and **Q-Learning (Off-Policy)** — for solving a navigation problem in a simulated GridWorld environment.

**Objectives:**

- **Implement the Environment:** Create a GridWorld class that defines the states, actions, rewards, and rules of the agent's world.
- **Implement Exploration Strategies:** Code two functions, `epsilon_greedy` and `softmax`, that an agent can use to select actions.
- **Implement SARSA:** Develop the SARSA function that updates action-values based on the ((S, A, R, S', A')) tuple, following an on-policy approach.
- **Implement Q-Learning:** Develop the Q-learning function that updates action-values based on the ((S, A, R, S')) tuple and the maximum possible value of the next state, following an off-policy approach.
- **Train and Compare Agents:** Run a comparative experiment by training agents with all four combinations — (SARSA/ε-greedy, Q-Learning/ε-greedy, SARSA/Softmax, Q-Learning/Softmax).
- **Visualize State-Value Function:** Display the state-value function as a heatmap to show which states the agent considers most valuable.

**Software/Tool:** Google Colab

## Theory:

This experiment explores two cornerstone Temporal Difference (TD) reinforcement learning algorithms SARSA and Q-Learning within a GridWorld environment. The objective is to analyze how these algorithms learn policies under different exploration strategies and how they balance exploration and exploitation to achieve optimal navigation.

**1. The GridWorld Problem**

GridWorld is a simple discrete environment consisting of a grid of cells:

- Each **cell** represents a **state (s)**.
- The **agent** starts at a random position and must reach a **goal state (G)** while avoiding obstacles.
- Possible **actions (a)** are typically **Up, Down, Left, Right**.
- The agent receives a **reward (r)** at each step, e.g.,
    - ( +1 ) for reaching the goal,
    - ( -1 ) for hitting a wall or invalid move,
    - ( 0 ) for normal transitions.

The goal of the agent is to learn an **optimal policy π*** that maximizes the **expected return**:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

where $\gamma$ is the discount factor that determines the importance of future rewards.

**2. Exploration–Exploitation Trade-off**

A fundamental challenge in RL is the **exploration–exploitation dilemma**:

- **Exploitation:** The agent chooses the action that it currently believes yields the highest reward (based on its Q-values).
- **Exploration:** The agent occasionally chooses random or probabilistic actions to discover new states and rewards.

Two exploration strategies are commonly used in this lab:

**a) ε-Greedy Policy**

With probability ( **ε** ), the agent explores randomly; otherwise, it exploits the best-known action:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s,a), & \text{with probability } 1-\epsilon \end{cases}$$

As training progresses, ( **ε** ) is often reduced gradually to encourage exploitation.

**b) Softmax (Boltzmann Exploration)**

In **Softmax exploration**, actions are chosen probabilistically based on their Q-values. The probability of choosing an action ( a ) in state ( s ) is given by:

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_b e^{Q(s,b)/\tau}}$$

where $\tau$ (temperature parameter) controls randomness:

- High $\tau$: More exploration (nearly uniform probability).
- Low $\tau$: More exploitation (favoring higher Q-values).

**3. SARSA (State–Action–Reward–State–Action)**

**SARSA** is an **on-policy TD control algorithm**, meaning it learns the value of the policy it is currently following, including its exploration behavior.

At each time step, the agent observes the transition $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ and updates its Q-value as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Where:

- $\alpha$: Learning rate
- $\gamma$: Discount factor
- $Q(S_t, A_t)$: Estimated action value
- The term in brackets is the **TD error**

SARSA learns by considering **what the agent actually does**, making it a *safer* approach when exploration leads to risky states.

**4. Q-Learning**

**Q-Learning** is an **off-policy TD control algorithm**, meaning it learns the value of the optimal policy, regardless of the agent's actual exploratory actions.

It uses the **maximum** estimated future Q-value as its target:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)]$$

Here, the agent's learning target assumes **optimal future actions**, even if the current behavior is exploratory. Thus, Q-Learning typically converges faster to the **optimal policy** but may take more risks during training.

**5. Key Comparison: SARSA vs Q-Learning**

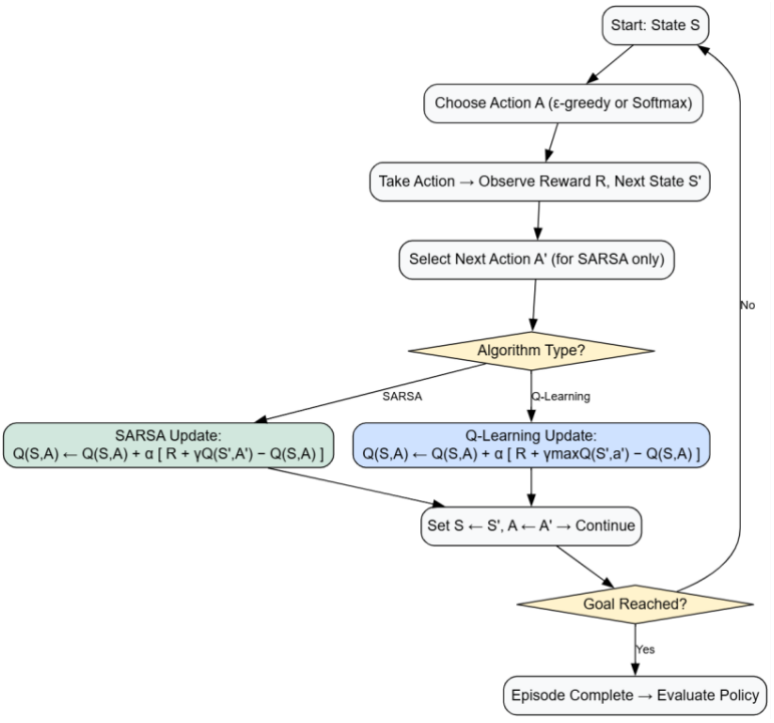
| Feature | SARSA (On-Policy) | Q-Learning (Off-Policy) |
|---|---|---|
| Update Target | $R + \gamma Q(S', A')$ | $R + \gamma \max_{a'} Q(S', a')$ |
| Policy Type | Learns the current (ε-greedy/softmax) policy | Learns the optimal policy |
| Exploration Handling | Includes exploratory actions in learning | Ignores exploration in update |
| Convergence | Safer, slower | Faster, but more risk-prone |
| Behavior | Learns from what it *does* | Learns from what it *could have done best* |

**6. Integration of Exploration Strategies**

Each algorithm can be paired with either ε-greedy or softmax exploration:These four combinations are compared to study their convergence behavior and policy quality.

| Combination | Description |
|---|---|
| **SARSA + ε-Greedy** | Conservative learning using stochastic exploration. |
| **SARSA + Softmax** | Smooth probabilistic exploration using temperature parameter. |
| **Q-Learning + ε-Greedy** | Fast off-policy learning with occasional random exploration. |
| **Q-Learning + Softmax** | Boltzmann-weighted selection focusing on high-reward actions. |

**7. Diagram: SARSA vs Q-Learning Process.**



**Attachment:** Code and Results

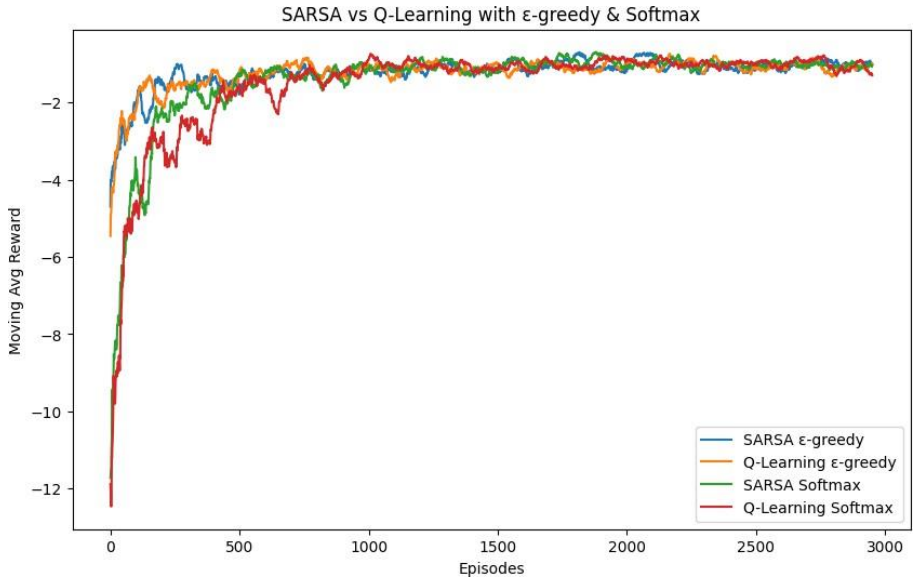
```
#
# Name: Kiran Biradar
# PRN: 20220802048
# ---------------------------
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict

# ============================================================
# GRIDWORLD ENVIRONMENT
# ============================================================
# A simple 4x4 GridWorld:
# - States: positions (row, col)
# - Actions: 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT
# - Rewards: -1 per step, 0 at terminal states
# - Goal: reach terminal state with maximum cumulative reward
# ============================================================

class GridWorld:
    def __init__(self, rows=4, cols=4, terminal_states=[(0,0), (3,3)]):
        self.rows = rows
        self.cols = cols
        self.terminal_states = set(terminal_states)
        self.reset()

    def reset(self):
        # Start from a random non-terminal state
        while True:
            r, c = np.random.randint(self.rows), np.random.randint(self.cols)
            if (r,c) not in self.terminal_states:
                self.state = (r,c)
                break
        return self.state

    def step(self, action):
        """Actions: 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT"""
        r, c = self.state
        if self.state in self.terminal_states:
            return self.state, 0, True  # episode ends at terminal

        # Move within grid boundaries
        if action == 0:   r = max(r-1, 0)
        elif action == 1: c = min(c+1, self.cols-1)
        elif action == 2: r = min(r+1, self.rows-1)
        elif action == 3: c = max(c-1, 0)
```

```
        self.state = (r, c)
        reward = 0 if self.state in self.terminal_states else -1
        done = self.state in self.terminal_states
        return self.state, reward, done

# ============================================================
# EXPLORATION STRATEGIES
# ============================================================
# 1. ε-greedy:
#    - With probability ε → choose random action
#    - With probability 1-ε → choose best known action
#    - Balances exploration vs exploitation
#
# 2. Softmax (Boltzmann Exploration):
#    - Converts Q-values into probabilities using temperature τ
#    - High τ → more exploration (uniform distribution)
#    - Low τ → greedy (deterministic)
# ============================================================

def epsilon_greedy(Q, state, epsilon, nA=4):
    if np.random.rand() < epsilon:
        return np.random.choice(nA)  # explore
    return np.argmax([Q[(state,a)] for a in range(nA)])  # exploit

def softmax(Q, state, tau=1.0, nA=4):
    q_values = np.array([Q[(state,a)] for a in range(nA)])
    prefs = np.exp(q_values / tau)  # convert to preferences
    probs = prefs / np.sum(prefs)   # normalize to probabilities
    return np.random.choice(nA, p=probs)

# ============================================================
# SARSA (On-policy TD Control)
# ============================================================
# Update rule:
#   Q(s,a) ← Q(s,a) + α [ r + γ Q(s',a') - Q(s,a) ]
#
# - Uses action a' from policy (on-policy)
# - Learns values of policy it actually follows
# - More conservative, tends to avoid risky actions
# ============================================================

def sarsa(env, episodes=5000, alpha=0.1, gamma=0.99,
          epsilon=0.1, tau=1.0, strategy="epsilon",
          epsilon_decay=0.999, tau_decay=0.999):
    Q = defaultdict(float)
    rewards = []
    for ep in range(episodes):
        s = env.reset()
        # Choose initial action using exploration strategy
```

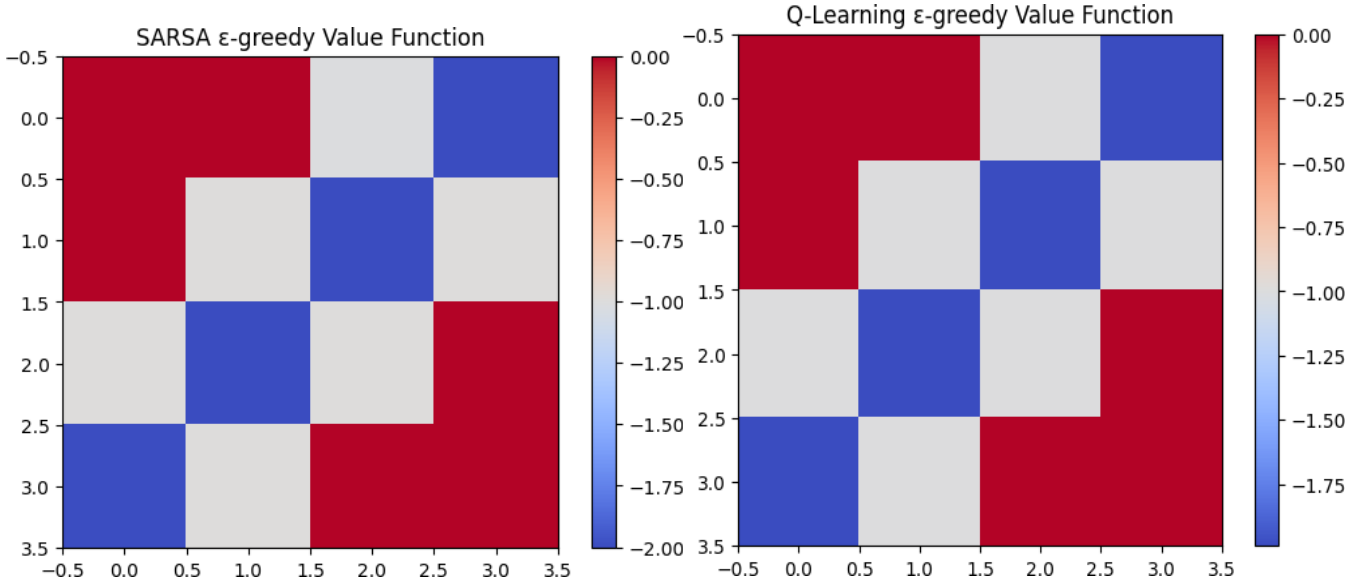
```
        if strategy=="epsilon":
            a = epsilon_greedy(Q, s, epsilon)
        else:
            a = softmax(Q, s, tau)

        total_reward, done = 0, False
        while not done:
            s_next, r, done = env.step(a)
            total_reward += r

            # Next action (a') chosen from same policy → On-policy
            if strategy=="epsilon":
                a_next = epsilon_greedy(Q, s_next, epsilon)
            else:
                a_next = softmax(Q, s_next, tau)

            # SARSA update
            Q[(s,a)] += alpha * (r + gamma * Q[(s_next,a_next)] - Q[(s,a)])
            s, a = s_next, a_next

        rewards.append(total_reward)

        # Decay exploration parameters over time
        epsilon *= epsilon_decay
        tau *= tau_decay
    return Q, rewards

# ============================================================
# Q-Learning (Off-policy TD Control)
# ============================================================
# Update rule:
#   Q(s,a) ← Q(s,a) + α [ r + γ max_a' Q(s',a') - Q(s,a) ]
#
# - Uses best next action (max) regardless of current policy
# - Off-policy: learns optimal policy while exploring with another
# - More aggressive than SARSA, converges faster but riskier
# ============================================================

def q_learning(env, episodes=5000, alpha=0.1, gamma=0.99,
               epsilon=0.1, tau=1.0, strategy="epsilon",
               epsilon_decay=0.999, tau_decay=0.999):
    Q = defaultdict(float)
    rewards = []
    for ep in range(episodes):
        s = env.reset()
        total_reward, done = 0, False
        while not done:
            # Choose action from policy (exploration)
            if strategy=="epsilon":
                a = epsilon_greedy(Q, s, epsilon)
```

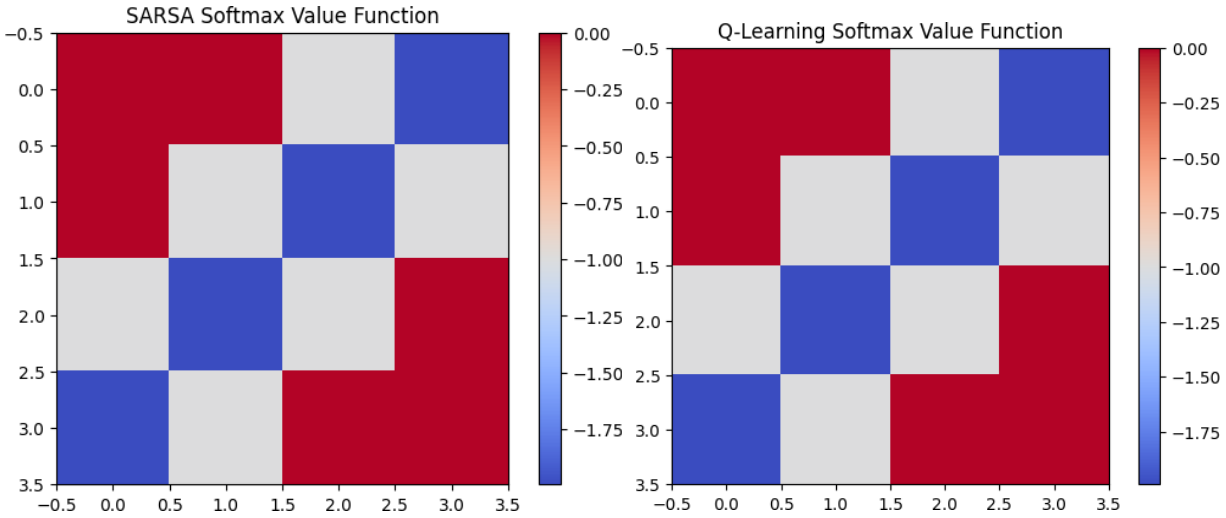
```
            else:
                a = softmax(Q, s, tau)

            s_next, r, done = env.step(a)
            total_reward += r

            # Q-Learning update (max over next actions)
            best_next = max([Q[(s_next,a_)] for a_ in range(4)])
            Q[(s,a)] += alpha * (r + gamma * best_next - Q[(s,a)])
            s = s_next

        rewards.append(total_reward)

        # Decay exploration
        epsilon *= epsilon_decay
        tau *= tau_decay
    return Q, rewards

# ============================================================
# VISUALIZATION HELPERS
# ============================================================
# - Reward plots (learning curves)
# - Policy visualization with arrows
# - Value heatmap (state values)
# ============================================================

def plot_rewards(results, window=50):
    plt.figure(figsize=(10,6))
    for label, rewards in results.items():
        avg = np.convolve(rewards, np.ones(window)/window, mode='valid')
        plt.plot(avg, label=label)
    plt.xlabel("Episodes")
    plt.ylabel("Moving Avg Reward")
    plt.title("SARSA vs Q-Learning with ε-greedy & Softmax")
    plt.legend()
    plt.show()

def extract_policy(Q, env):
    actions = ["↑","→","↓","←"]
    policy = np.full((env.rows, env.cols), " ")
    for r in range(env.rows):
        for c in range(env.cols):
            if (r,c) in env.terminal_states:
                policy[r,c] = "T"
            else:
                best_a = np.argmax([Q[((r,c), a)] for a in range(4)])
                policy[r,c] = actions[best_a]
    return policy

def plot_policy(policy, title="Policy"):
```

```
    print(f"\n{title}")
    for row in policy:
        print(" ".join(row))

def plot_value(Q, env, title="Value Function"):
    V = np.zeros((env.rows, env.cols))
    for r in range(env.rows):
        for c in range(env.cols):
            V[r,c] = max([Q[((r,c),a)] for a in range(4)])
    plt.imshow(V, cmap="coolwarm", interpolation="nearest")
    plt.colorbar()
    plt.title(title)
    plt.show()

# ============================================================
# MAIN EXECUTION
# ============================================================
if __name__== "__main__":
    env = GridWorld()
    episodes = 3000

    # Train all algorithms with both strategies
    Q_sarsa_eps, R_sarsa_eps = sarsa(env, episodes, strategy="epsilon", epsilon=0.2)
    Q_q_eps, R_q_eps = q_learning(env, episodes, strategy="epsilon", epsilon=0.2)

    Q_sarsa_soft, R_sarsa_soft = sarsa(env, episodes, strategy="softmax", tau=1.0)
    Q_q_soft, R_q_soft = q_learning(env, episodes, strategy="softmax", tau=1.0)

    # Compare rewards
    results = {
        "SARSA ε-greedy": R_sarsa_eps,
        "Q-Learning ε-greedy": R_q_eps,
        "SARSA Softmax": R_sarsa_soft,
        "Q-Learning Softmax": R_q_soft
    }
    plot_rewards(results)

    # Show policies
    plot_policy(extract_policy(Q_sarsa_eps, env), "SARSA ε-greedy Policy")
    plot_policy(extract_policy(Q_q_eps, env), "Q-Learning ε-greedy Policy")
    plot_policy(extract_policy(Q_sarsa_soft, env), "SARSA Softmax Policy")
    plot_policy(extract_policy(Q_q_soft, env), "Q-Learning Softmax Policy")

    # Show value functions
    plot_value(Q_sarsa_eps, env, "SARSA ε-greedy Value Function")
    plot_value(Q_q_eps, env, "Q-Learning ε-greedy Value Function")
    plot_value(Q_sarsa_soft, env, "SARSA Softmax Value Function")
    plot_value(Q_q_soft, env, "Q-Learning Softmax Value Function")
```




```
SARSA ε-greedy Policy
T ← ← ↓
↑ ↑ → ↓
↑ → ↓ ↓
↑ → → T

Q-Learning ε-greedy Policy
T ← ← ←
↑ ← ↑ ↓
↑ → ↓ ↓
↑ → → T

SARSA Softmax Policy
T ← ← ↓
↑ ↑ ← ↓
↑ ↓ → ↓
→ → → T

Q-Learning Softmax Policy
T ← ← ←
↑ ← ↓ ↓
↑ → ↓ ↓
↑ → → T
```




SARSA Softmax Value Function
Q-Learning Softmax Value Function
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75

## Conclusion:

# Practical 04

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

## Aim:

Implementation of Deep Q-Network (DQN).

## Objectives:

- Build a Function Approximator: Design and implement a multi-layer neural network (QNetwork) using PyTorch to approximate the action-value (Q) function.
- Implement Experience Replay: Create a ReplayBuffer to store the agent's experiences ((state, action, reward, next_state, done)) and sample random mini-batches for training to stabilize learning.
- Develop the DQN Agent: Construct a DQNAgent class that integrates:
    - An online network for action selection and learning.
    - A target network to provide stable Q-value targets.
    - An ε-greedy policy with decay for exploration–exploitation management.
- Create the Training Loop: Implement a function (`train_dqn`) that allows the agent to interact with the CartPole-v1 environment, collecting experiences and updating its network over many episodes.
- Evaluate Performance: Visualize and analyze the agent's learning progress by plotting the total reward achieved in each episode to confirm the learned balancing policy.

**Software/Tool:** Google Colab

## Theory:

Deep Q-Network (DQN) is a significant advancement in value-based reinforcement learning that combines the Q-Learning algorithm with deep neural networks.
It enables agents to learn effective policies from high-dimensional or continuous state spaces, such as images or sensor data, where tabular Q-learning is infeasible.

### 1. Q-Learning Recap

Traditional Q-learning updates the Q-value for each state-action pair using the Bellman optimality equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

However, when the state space is large or continuous, maintaining a tabular Q-table becomes impossible.

To overcome this, DQN replaces the table with a deep neural network that approximates the Q-function.

**2. Function Approximation using Neural Networks**

A neural network $Q_\theta(s,a)$ parameterized by weights $\theta$ is used to approximate the optimal action-value function $Q^*(s,a)$:

$$Q^*(s,a) \approx Q_\theta(s,a)$$

The network takes a **state vector** $s$ as input and outputs Q-values for all possible actions.

Training adjusts $\theta$ to minimize the difference between predicted and target Q-values.

**3. Experience Replay**

A major innovation in DQN is the Experience Replay Buffer, which improves data efficiency and stability.

It stores tuples of experience:

$$(s_t, a_t, r_{t+1}, s_{t+1}, done)$$

Instead of learning from consecutive transitions, which are highly correlated, DQN samples random mini-batches from the buffer, ensuring uncorrelated and diverse training samples.

**4. Target Network**

Another key innovation is the introduction of a **Target Network** $Q_\theta$ — a periodically updated copy of the online Q-network $Q_\theta$.

**Reason:**

Without it, the network's target keeps changing as it learns, creating instability.

The **target Q-value** for training becomes:

$$y = r + \gamma \max_{a'} Q_\theta (s',a')$$

The **online network** learns to minimize the error between its current Q-value and this stable target.

**5. Exploration vs. Exploitation (ε-Greedy Policy)**

To balance exploration and exploitation, DQN uses an ε-greedy policy:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q_\theta(s,a), & \text{with probability } 1-\epsilon \end{cases}$$

The value of ε decays over time from a high exploration phase to a low exploration (exploitation) phase, allowing the agent to shift from discovering new strategies to refining its learned policy.

**6. Loss Function**

The DQN minimizes the Mean Squared Error (MSE) between the predicted Q-value and the target Q-value:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}\left[\left(y - Q_\theta(s,a)\right)^2\right]$$

where

$$y = r + \gamma(1 - done)\max_{a'} Q_\theta \ (s', a')$$

This loss encourages the network to predict values close to the stable target provided by the target network

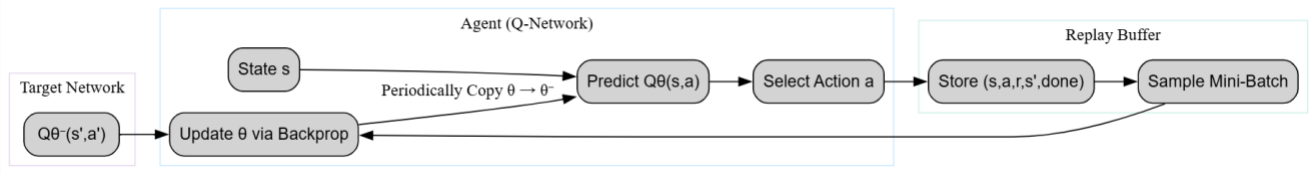
**7. Training Loop**

The learning process involves the following iterative steps:

1. **Interaction:** The agent observes state $s_t$ and selects an action $a_t$ via ε-greedy policy.
2. **Transition:** It receives reward $r_t$ and next state $s_{t+1}$.
3. **Storage:** The transition tuple $(s_t, a_t, r_t, s_{t+1}, done)$ is stored in the replay buffer.
4. **Sampling:** Random mini-batches are drawn from the buffer.
5. **Target Computation:** The target value $y$ is computed using the target network.
6. **Update:** The online network minimizes loss $L(\theta)$.
7. **Synchronization:** Periodically copy weights from the online network to the target network.

This process repeats until convergence.

**8. Diagram: DQN Learning Architecture**



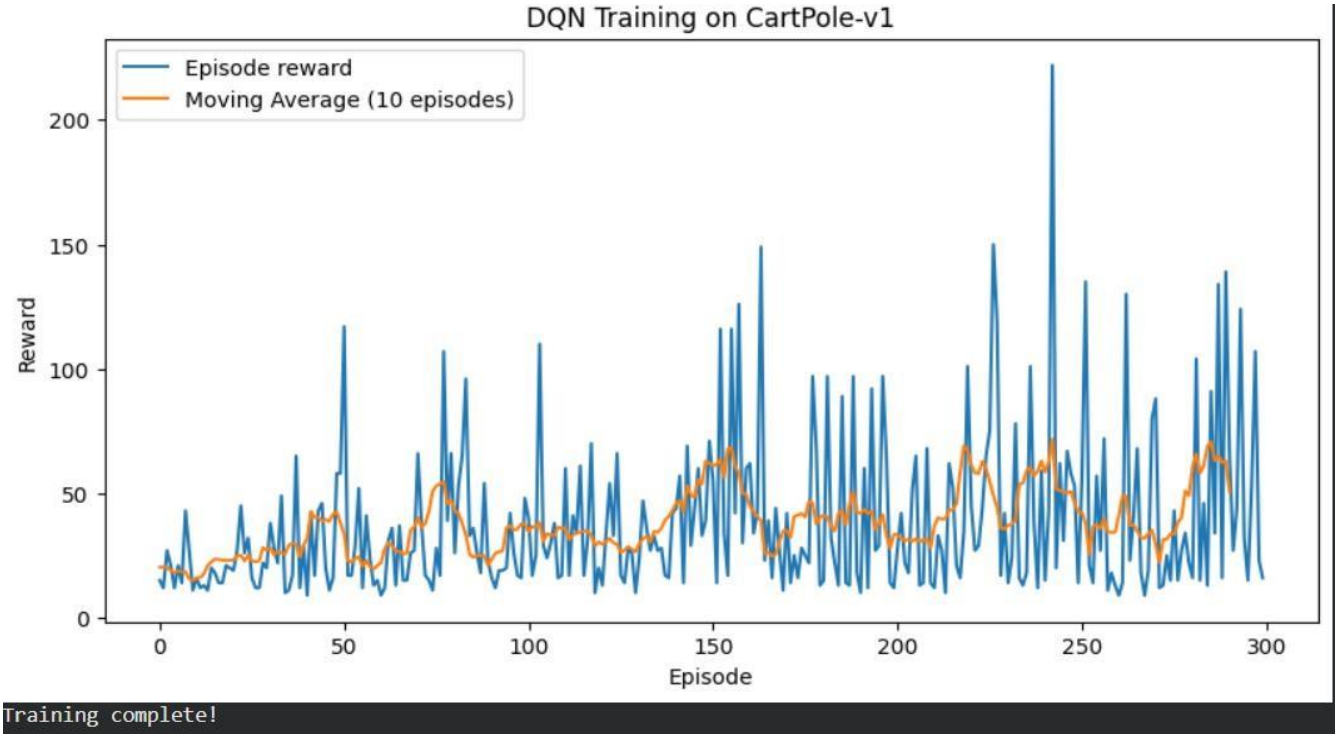
**Attachment:** Code and Results

```
# Name: Kiran Biradar
# PRN: 20220802048
# Advanced Deep Q-Network (DQN) with Gymnasium + Visualization

"""
Theory Recap:

DQN combines Q-Learning with deep neural networks to handle
large/continuous state spaces. Key innovations are:

1. Experience Replay:
   - Store (state, action, reward, next_state, done) in a buffer.
   - Sample random batches to reduce correlation between updates.


2. Target Network:
   - Maintain two networks: Q (online) and Q_target (frozen copy).
   - Update target network slowly to stabilize training.

3. Exploration vs. Exploitation:
   - Use ε-greedy policy.
   - Start with high ε (explore more), decay over time to ε_min
     (exploit learned policy).
  ------------------------------------------------------------
4. Loss Function:
   - Mean Squared Error (MSE) between predicted Q(s,a) and target:
       target = r + γ * max_a' Q_target(s', a')
"""

import random
import numpy as np
import gymnasium as gym
import torch
import torch.nn as nn
import torch.optim as optim
from collections import deque
import matplotlib.pyplot as plt
  ------------------------------------------------------------
#
# Neural Network for Q-function------------------------------
#
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.fc3 = nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)  # returns Q-values for all actions


#
# Replay Buffer
#
class ReplayBuffer:
    def __init__(self, capacity=10000):
  ------------------------------------------------------------
        self.buffer = deque(maxlen=capacity)

  ------------------------------------------------------------
    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (np.array(states), np.array(actions), np.array(rewards),
                np.array(next_states), np.array(dones))

    def __len__(self):
        return len(self.buffer)


#
# DQN Agent
#
class DQNAgent:
    def __init__(self, state_dim, action_dim, gamma=0.99, lr=1e-3,
                 batch_size=64, buffer_capacity=10000, target_update=100):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.gamma  = gamma
        self.batch_size = batch_size
        self.target_update = target_update

        # Online Q-network & Target Q-network
        self.q_net = QNetwork(state_dim, action_dim)
        self.target_net = QNetwork(state_dim, action_dim)
        self.target_net.load_state_dict(self.q_net.state_dict())
        self.target_net.eval()

        # Optimizer & Replay Buffer
        self.optimizer = optim.Adam(self.q_net.parameters(), lr=lr)
        self.replay_buffer = ReplayBuffer(buffer_capacity)

        self.steps = 0

    def select_action(self, state, epsilon):
        # ε-greedy policy
        if random.random() < epsilon:
```

```
            return random.randint(0, self.action_dim - 1)
        else:
            state = torch.FloatTensor(state).unsqueeze(0)
            q_values = self.q_net(state)
            return q_values.argmax().item()

    def update(self):
        if len(self.replay_buffer) < self.batch_size:
            return

        # Sample batch
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size)
        states = torch.FloatTensor(states)
        actions = torch.LongTensor(actions).unsqueeze(1)
        rewards = torch.FloatTensor(rewards).unsqueeze(1)
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones).unsqueeze(1)

        # Q(s,a)
        q_values = self.q_net(states).gather(1, actions)

        # Target Q-values
        with torch.no_grad():
            next_q_values = self.target_net(next_states).max(1)[0].unsqueeze(1)
            target_q = rewards + self.gamma * next_q_values * (1 - dones)

        # Loss = MSE
        loss = nn.MSELoss()(q_values, target_q)

        # Gradient descent
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update target network
        self.steps += 1
        if self.steps % self.target_update == 0:
            self.target_net.load_state_dict(self.q_net.state_dict())


#
# Training Loop
#
def train_dqn(env_name="CartPole-v1", episodes=300):
    env = gym.make(env_name)
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    agent = DQNAgent(state_dim, action_dim)
```

```
    # Exploration schedule
    epsilon_start, epsilon_end, epsilon_decay = 1.0, 0.01, 500
    epsilon = epsilon_start
    rewards_history = []

    for ep in range(episodes):
        state, _ = env.reset()
        total_reward = 0

        for t in range(500):
            action = agent.select_action(state, epsilon)
            # Gymnasium API: step returns 5 values
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            agent.replay_buffer.push(state, action, reward, next_state, done)
            agent.update()

            state = next_state
            total_reward += reward

            if done:
                break

        # Decay epsilon
        epsilon = max(epsilon_end, epsilon * np.exp(-1.0 / epsilon_decay))

        rewards_history.append(total_reward)
        print(f"Episode {ep+1}/{episodes}, Reward: {total_reward}, Epsilon: {epsilon:.3f}")

    env.close()
    return rewards_history


# ------------------------------------------------------------
# Run Training + Plot Rewards
# ------------------------------------------------------------
if __name__== "__main__":
    rewards = train_dqn("CartPole-v1", episodes=300)

    # Plot rewards per episode
    plt.figure(figsize=(10,5))
    plt.plot(rewards, label="Episode reward")
    plt.plot(np.convolve(rewards, np.ones(10)/10, mode='valid'),
             label="Moving Average (10 episodes)")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.title("DQN Training on CartPole-v1")
    plt.legend()
    plt.show()
```




```
Training complete!
```

**Conclusion:**

# Practical 05

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

## Aim:

Implementation and Application of REINFORCE Algorithm with Baseline and Advantage Function.

## Objectives:

- To understand and implement the **REINFORCE algorithm** (a Monte Carlo Policy Gradient method) for reinforcement learning.
- To integrate a **value function baseline (Critic)** to reduce variance and stabilize training in policy gradient methods.
- To compare and implement the **REINFORCE with Baseline** algorithm using both PyTorch and TensorFlow frameworks.
- To observe the learning behavior and performance of an agent on the **CartPole-v1 environment** using the Gymnasium API.

## Software/Tool:

Google Colab

## Theory:

The **REINFORCE algorithm** is one of the earliest and most fundamental **policy gradient** methods in reinforcement learning.
It aims to **directly optimize the policy** by adjusting the parameters of the policy network to maximize the **expected cumulative reward**.

## 1. Policy Gradient Framework

In **policy-based reinforcement learning**, the policy is represented by a parameterized probability distribution $\pi_\theta(a|s)$, where $\theta$ denotes the learnable parameters (e.g., weights of a neural network).
The goal is to maximize the expected return $J(\theta)$:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t]$$

The gradient of the objective with respect to policy parameters $\theta$ is derived using the **Policy Gradient Theorem**:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a_t|s_t)\, G_t\Big]$$

Here, $G_t$ is the **total discounted return** from time $t$:

$$G_t = \sum_{k=0}^{T-t} \gamma^k R_{t+k+1}$$

## 2. REINFORCE Algorithm

The **REINFORCE algorithm** estimates this expectation using **Monte Carlo sampling** over multiple episodes.
It updates the policy parameters in the direction of actions that led to higher returns.

The update rule is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\, G_{i,t}$$

Where:

- $N$: Number of sampled episodes
- $T_i$: Length of episode $i$
- $G_{i,t}$: Discounted return at time $t$
- $\nabla_\theta \log \pi_\theta(a|s)$: Gradient of log-policy

This method directly adjusts the policy to **increase the probability** of actions yielding high returns.

## 3. Variance Reduction using a Baseline

A limitation of the basic REINFORCE algorithm is **high variance** in gradient estimates, which can cause unstable or slow learning.
To mitigate this, a **baseline function** ( b(s) ) is subtracted from the return without introducing bias.

The modified update rule becomes:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a_t|s_t)\,(G_t - b(s_t))\Big]$$

A common choice for the baseline is the **state-value function** $V(s_t)$, which represents the expected return from state $s_t$.

Hence, the REINFORCE with baseline update rule is:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T_i}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\,(G_{i,t} - V(s_{i,t}))$$

## 4. Advantage Function

The term $(G_t - V(s_t))$ is known as the **Advantage Function** $A(s_t, a_t)$.
It measures how much better (or worse) an action $a_t$ is compared to the average performance from that state $s_t$.

$$A(s,a) = Q(s,a) - V(s)$$

Since $Q(s,a)$ (the action-value function) represents the expected return for taking action $a$ in state $s$, the advantage can be estimated empirically using the observed return:

$$A(s_t, a_t) \approx G_t - V(s_t)$$

**Interpretation:**

- If ( A(s,a) > 0 ): The action performed better than expected — increase its probability.
- If ( A(s,a) < 0 ): The action performed worse than expected — decrease its probability.

## 5. REINFORCE with Baseline Algorithm Steps

1. **Initialize** policy network $\pi_\theta(a|s)$ and value function $V_\phi(s)$.
2. For each episode:
   a. Generate a trajectory: $(s_0, a_0, r_1, \ldots, s_T)$.
   b. Compute discounted returns $G_t$ for each step.
   c. Estimate advantages $A_t = G_t - V_\phi(s_t)$.
   d. Update policy parameters $\theta$ using gradient ascent:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_t$$

   e. Update value network parameters $\phi$ by minimizing prediction error:

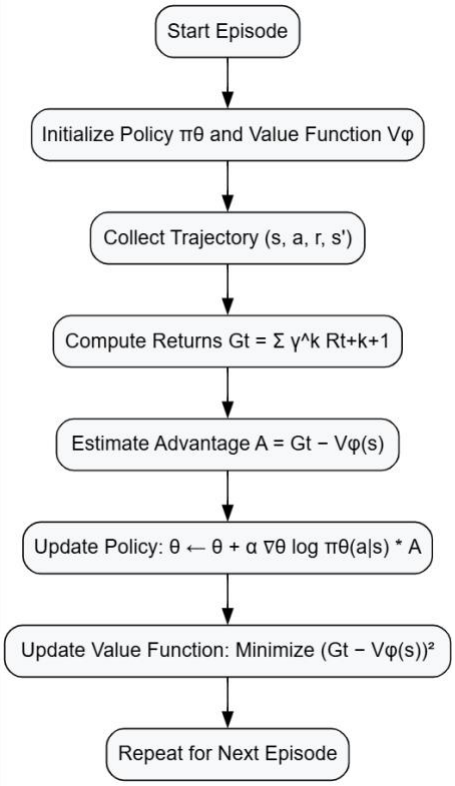
$$L_V(\phi) = (G_t - V_\phi(s_t))^2$$

3. Repeat until convergence.

## 6. Advantages of REINFORCE with Baseline

- **Variance Reduction:** The baseline reduces fluctuations in policy gradients, improving learning stability.
- **Efficient Updates:** By considering the advantage, updates focus only on deviations from the expected return.
- **Improved Convergence:** Helps faster and smoother convergence compared to vanilla REINFORCE.

## 7. Diagram: REINFORCE with Baseline Flow



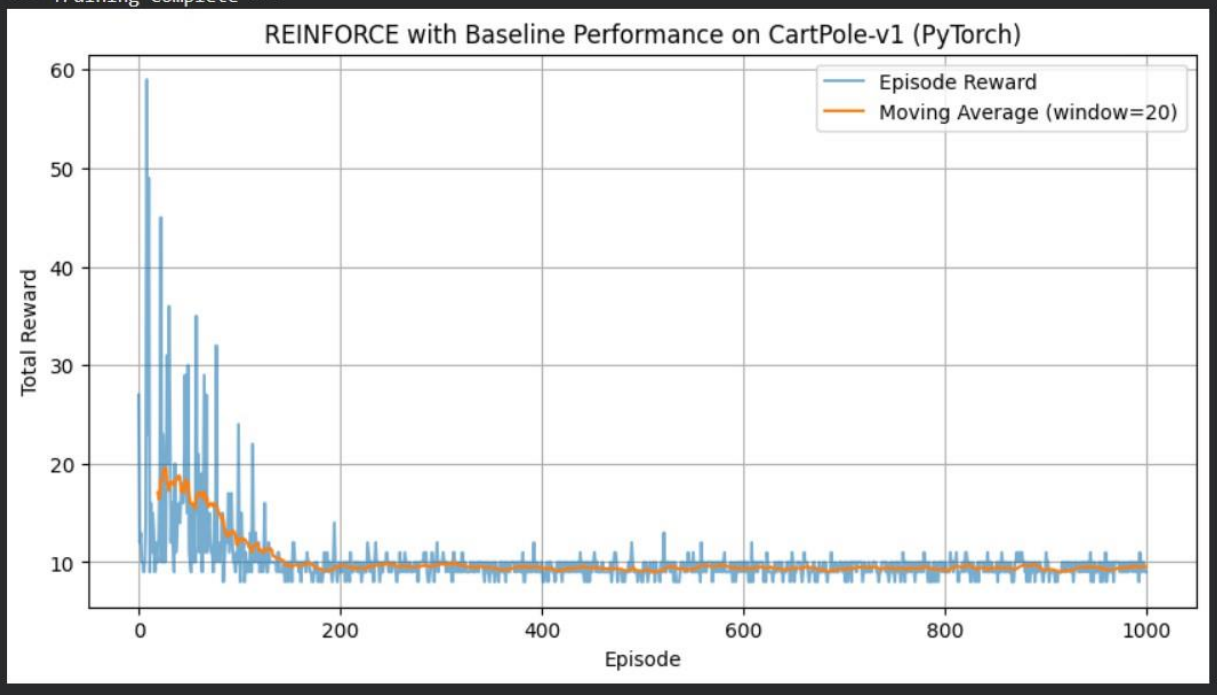
**Attachment:** Code and Results

```
# Name: Kiran Biradar
# PRN: 20220802048

# REINFORCE with Baseline (PyTorch)
#
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

#
# Policy Network (Actor)
#
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=128):
        super(PolicyNetwork, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, action_dim),
            nn.Softmax(dim=1)
        )
    def forward(self, x):
        return self.fc(x)

#
# Value Network (Critic)
#
class ValueNetwork(nn.Module):
    def __init__(self, state_dim, hidden=128):
        super(ValueNetwork, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 1)
        )
    def forward(self, x):
        return self.fc(x)

#
# REINFORCE with Baseline Training Loop
#
def reinforce(env, policy_net, value_net, policy_optimizer, value_optimizer, episodes=1000,
gamma=0.99):
    reward_history = []
    for episode in range(episodes):
        state, info = env.reset()
```

```
        log_probs, values, rewards = [], [], []
        done = False
        total_reward = 0

        while not done:
            state_t = torch.FloatTensor(state).unsqueeze(0)
            # Actor: sample action
            probs = policy_net(state_t)
            dist = torch.distributions.Categorical(probs)
            action = dist.sample()

            log_probs.append(dist.log_prob(action))
            values.append(value_net(state_t))

            # Step in environment (Gymnasium API)
            next_state, reward, terminated, truncated, info = env.step(action.item())
            rewards.append(reward)
            total_reward += reward

            state = next_state
            done = terminated or truncated

        reward_history.append(total_reward)

        # Compute discounted returns (Monte Carlo Return Gt)
        returns, G = [], 0
        for r in reversed(rewards):
            G = r + gamma * G
            returns.insert(0, G)
        returns = torch.tensor(returns, dtype=torch.float32)

        # Normalize returns for stability
        returns = (returns - returns.mean()) / (returns.std() + 1e-8)

        log_probs = torch.stack(log_probs)
        values = torch.cat(values).squeeze()

        # Advantage = Return - Value (Baseline)
        advantages = returns - values.detach()

        # Policy (Actor) loss: J(θ) = E[log π(a|s) * Advantage]
        policy_loss = -(log_probs * advantages).mean()

        # Value (Critic) loss: L(φ) = MSE(V(s), Gt)
        value_loss = nn.MSELoss()(values, returns)

        # Update actor
        policy_optimizer.zero_grad()
        policy_loss.backward()
        policy_optimizer.step()
```

```
        # Update critic
        value_optimizer.zero_grad()
        value_loss.backward()
        value_optimizer.step()

        if (episode+1) % 50 == 0:
            print(f"Episode: {(episode+1)}/{episodes} | Reward: {total_reward} "
                  f"| Policy Loss: {policy_loss.item():.4f} | Value Loss:
{value_loss.item():.4f}")

    return reward_history

#
# Main Execution and Visualization
#
if __name__== "__main__":
    # Ensure reproducibility
    torch.manual_seed(42)
    np.random.seed(42)

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

    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    # Initialize PyTorch components
    policy_net = PolicyNetwork(state_dim, action_dim)
    value_net = ValueNetwork(state_dim)

    policy_optimizer = optim.Adam(policy_net.parameters(), lr=1e-3)
    value_optimizer = optim.Adam(value_net.parameters(), lr=1e-3)

    print("--- Starting REINFORCE with Baseline (PyTorch) ---")
    rewards = reinforce(env, policy_net, value_net, policy_optimizer, value_optimizer,
episodes=1000)
    print("--- Training Complete ---")

    # Visualize Performance (Objective 4)
    plt.figure(figsize=(10,5))
    plt.plot(rewards, label="Episode Reward", alpha=0.6)
    # Moving Average for smoothing
    window_size = 20
    if len(rewards) >= window_size:
        avg_rewards = np.convolve(rewards, np.ones(window_size)/window_size, mode='valid')
        plt.plot(np.arange(window_size - 1, len(rewards)), avg_rewards, label=f"Moving
Average (window={window_size})")

    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("REINFORCE with Baseline Performance on CartPole-v1 (PyTorch)")
    plt.legend()
```

```
Episode: 900/1000 | Reward: 9.0 | Policy Loss: 0.0000 | Value Loss: 0.0008
Episode: 950/1000 | Reward: 9.0 | Policy Loss: 0.0000 | Value Loss: 0.0018
Episode: 1000/1000 | Reward: 9.0 | Policy Loss: 0.0000 | Value Loss: 0.0011
--- Training Complete ---
```




## Conclusion:

# Practical 06

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

**Aim:**

Implement the **Actor–Critic (A2C)** method for continuous control tasks.

**Objectives:**

- Implement the **Actor–Critic (A2C)** reinforcement learning algorithm using **PyTorch**.
- Apply the A2C algorithm to solve a **continuous control task** using the **Pendulum-v1** environment from **Gymnasium**.
- Analyze the training process by plotting the **rewards obtained over episodes** and observing the agent's learning progress.
- Understand the **interplay between the Actor (policy)** and the **Critic (value function)** in guiding the learning process.

**Software/Tool:**

Google Colab

**Theory:**

The **Actor–Critic (A2C)** method is a class of reinforcement learning algorithms that combines the strengths of both **value-based** and **policy-based** methods.
It is particularly suited for **continuous control tasks**, where the action space is continuous rather than discrete.

**1. Conceptual Overview**

In reinforcement learning, two main approaches exist:

- **Value-Based Methods (e.g., Q-Learning):** Learn a value function $V(s)$ or $Q(s,a)$ to estimate expected rewards.
- **Policy-Based Methods (e.g., REINFORCE):** Learn a parameterized policy $\pi_\theta(a|s)$ that directly outputs actions.

**Actor–Critic** methods combine both:

- The **Actor** updates the policy parameters to select actions.
- The **Critic** evaluates these actions using a value function to provide feedback (advantage signal).

## 2. Actor–Critic Framework

The **Actor** represents the **policy** $\pi_\theta(a|s)$, parameterized by $\theta$.
The **Critic** represents the **value function** $V_w(s)$, parameterized by $w$.

At each time step $t$:

1. The Actor selects an action $a_t \sim \pi_\theta(a_t|s_t)$.
2. The environment returns a reward $r_t$ and a new state $s_{t+1}$.
3. The Critic computes the **Advantage Function** $A(s_t, a_t)$, which measures how good the action was compared to the expected value.

## 3. Advantage Function

The **Advantage Function** represents how much better an action is compared to the average action in that state:

$$A(s,a) = Q(s,a) - V(s)$$

Since $Q(s,a)$ is often hard to compute directly, it is estimated using the observed **return** $R$:

$$A(s,a) \approx R - V(s)$$

Where:

- $R$ is the **discounted sum of future rewards**,

$$R_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k+1}$$

- $\gamma$ is the **discount factor** ($0 < \gamma \leq 1$).

## 4. Actor Update Rule (Policy Gradient)

The Actor updates its policy parameters $\theta$ to **increase the probability of actions** that lead to higher advantage values:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{s_t, a_t \sim \pi_\theta} \Big[ \nabla_\theta \log \pi_\theta(a_t|s_t)\, A(s_t, a_t) \Big]$$

This encourages actions with **positive advantage** and discourages those with **negative advantage**.

## 5. Critic Update Rule (Value Function Learning)

The Critic learns to predict the value function $V(s_t)$ that estimates the expected return from a state.
Its parameters $w$ are updated to minimize the **Mean Squared Error (MSE)** between the estimated value and the actual return:

$$L_{critic}(w) = \big(R_t - V_w(s_t)\big)^2$$

This helps the Critic provide more accurate evaluations for future updates of the Actor.

## 6. Combined Loss Function

The total loss function for the Actor–Critic model combines both objectives:

$$L_{total} = L_{actor} + \beta L_{critic}$$

Where:

- $L_{actor} = -\log \pi_\theta(a_t|s_t)\, A(s_t, a_t)$
- $L_{critic} = (R_t - V_w(s_t))^2$
- $\beta$ is a scaling factor that balances policy and value updates.

### 7. Continuous Action Space (Pendulum-v1)

In **continuous control tasks**, such as the **Pendulum-v1** environment, actions are real-valued (e.g., torque values).
Therefore, the Actor outputs parameters of a **Gaussian distribution** instead of discrete probabilities:

$$a_t \sim \mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))$$

Where:

- $\mu_\theta(s_t)$: Mean of the action distribution (predicted by Actor).
- $\sigma_\theta(s_t)$: Standard deviation of the distribution (learned or fixed).

The Actor thus learns to produce **continuous control signals** to balance the pendulum.

### 8. Network Architecture

The Actor–Critic network typically has:

1. **Shared Layers:** Process the state input $s_t$ to extract common features.
2. **Actor Head:** Outputs mean $\mu$ and log standard deviation $\log \sigma$ for the Gaussian action distribution.
3. **Critic Head:** Outputs a scalar value $V(s_t)$ estimating the state's value.

### 9. Training Algorithm (A2C)

**Algorithm Steps:**

1. Initialize shared network parameters $\theta, w$.
2. For each episode:
   a. Observe initial state $s_0$.
   b. For each step $t$:
   - Sample action $a_t \sim \pi_\theta(a_t|s_t)$.
   - Execute $a_t$, observe $r_t, s_{t+1}$.
   - Store transition $(s_t, a_t, r_t, s_{t+1})$.

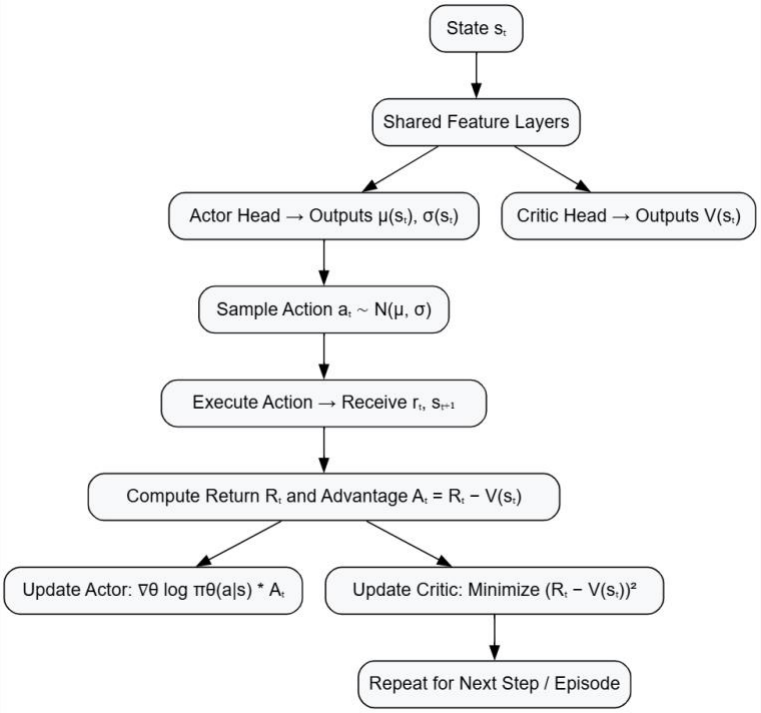
   c. Compute return $R_t = \sum_k \gamma^k r_{t+k+1}$.
   d. Compute advantage $A_t = R_t - V_w(s_t)$.
   e. Update Actor and Critic using gradients:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_\theta(a_t|s_t) A_t$$

$$w \leftarrow w - \alpha_w \nabla_w (R_t - V_w(s_t))^2$$

3. Repeat until convergence.

## 10. Diagram: Actor–Critic (A2C) Architecture



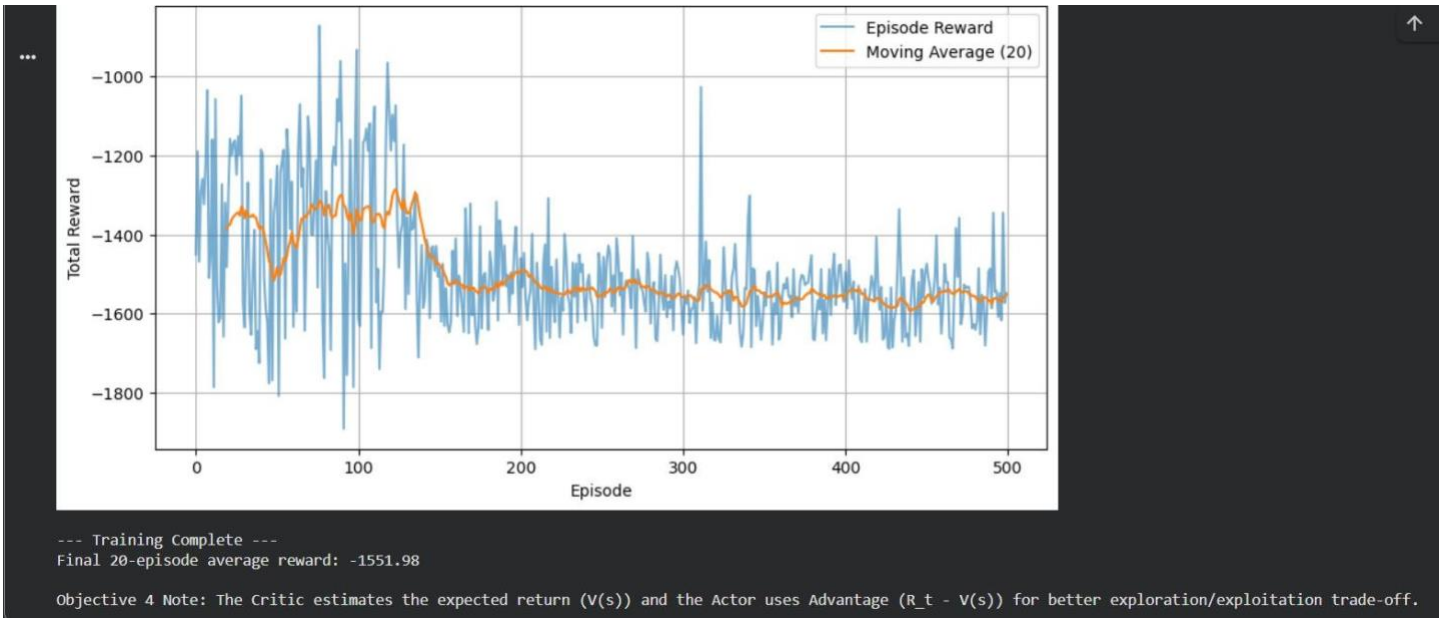

### Code:

```
# Name -Kiran Biradar
# PRN: 20220802048

import gymnasium as gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# Actor Network (Gaussian Policy)
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, hidden=128):
        super(Actor, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU()
        )
        self.mean = nn.Linear(hidden, action_dim)
```

```
        self.log_std = nn.Parameter(torch.zeros(action_dim))

    def forward(self, x):
        x = self.fc(x)
        mean = self.mean(x)
        std = torch.exp(self.log_std)
        return mean, std

# Critic Network (Value Function)
class Critic(nn.Module):
    def __init__(self, state_dim, hidden=128):
        super(Critic, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(state_dim, hidden),
            nn.ReLU(),
            nn.Linear(hidden, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 1)
        )

    def forward(self, x):
        return self.fc(x)

# Function to select action from Actor
def select_action(actor, state):
    state_t = torch.FloatTensor(state).unsqueeze(0)
    mean, std = actor(state_t)
    dist = torch.distributions.Normal(mean, std)
    action = dist.sample()
    action_clipped = torch.clamp(action, -2.0, 2.0)
    log_prob = dist.log_prob(action).sum(dim=-1)
    return action_clipped.detach().numpy()[0], log_prob

# Actor-Critic Training Loop
def train_actor_critic(env_name='Pendulum-v1', episodes=500, gamma=0.99, lr=3e-4):
    env = gym.make(env_name)
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.shape[0]
    actor = Actor(state_dim, action_dim)
    critic = Critic(state_dim)
    actor_optimizer = optim.Adam(actor.parameters(), lr=lr)
    critic_optimizer = optim.Adam(critic.parameters(), lr=lr)
    reward_history = []

    print("--- Starting A2C Training on Pendulum-v1 ---")
    for ep in range(episodes):
        state, _ = env.reset()
        log_probs = []
        values = []
        rewards = []
```

```
        done = False

        while not done:
            value = critic(torch.FloatTensor(state).unsqueeze(0))
            action, log_prob = select_action(actor, state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            log_probs.append(log_prob)
            values.append(value)
            rewards.append(reward)
            state = next_state

        total_reward = sum(rewards)
        reward_history.append(total_reward)

        # Compute discounted returns (Monte Carlo)
        returns = []
        G = 0
        for r in reversed(rewards):
            G = r + gamma * G
            returns.insert(0, G)
        returns = torch.tensor(returns, dtype=torch.float32).unsqueeze(1)
        values = torch.cat(values)
        log_probs = torch.stack(log_probs).unsqueeze(1)
        advantages = returns - values.detach()
        actor_loss = -(log_probs * advantages).mean()
        critic_loss = nn.MSELoss()(values, returns)

        actor_optimizer.zero_grad()
        actor_loss.backward()
        actor_optimizer.step()

        critic_optimizer.zero_grad()
        critic_loss.backward()
        critic_optimizer.step()

        if (ep + 1) % 20 == 0:
            print(f"Episode {ep+1}/{episodes} | Total Reward: {total_reward:.2f} | "
                  f"Actor Loss: {actor_loss.item():.4f} | Critic Loss:
{critic_loss.item():.4f}")

    env.close()
    return actor, critic, reward_history

# Main Execution and Visualization
if __name__ == "__main__":
    torch.manual_seed(42)
    np.random.seed(42)
    actor, critic, rewards = train_actor_critic(episodes=500)
    plt.figure(figsize=(10,5))
```

```
    plt.plot(rewards, label="Episode Reward", alpha=0.6)
    window_size = 20
    if len(rewards) >= window_size:
        avg_rewards = np.convolve(rewards, np.ones(window_size)/window_size, mode='valid')
        plt.plot(np.arange(window_size - 1, len(rewards)), avg_rewards,
                 label=f"Moving Average ({window_size})")
    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("Actor-Critic (A2C) Performance on Pendulum-v1")
    plt.legend()
    plt.grid(True)
    plt.show()
    print("\n--- Training Complete ---")
    print(f"Final 20-episode average reward: {np.mean(rewards[-20:]):.2f}")
    print("\nObjective 4 Note: The Critic estimates the expected return (V(s)) and the Actor uses Advantage (R_t - V(s)) for better exploration/exploitation trade-off.")
```




```
--- Training Complete ---
Final 20-episode average reward: -1551.98

Objective 4 Note: The Critic estimates the expected return (V(s)) and the Actor uses Advantage (R_t - V(s)) for better exploration/exploitation trade-off.
```

## Conclusion:

# Practical 07

| |
|---|
| **Student Name: Kiran Biradar** |
| **Date of Experiment:** |
| **Date of Submission:** |
| **PRN No: 20220802048** |

**Aim:**

To implement and analyze a **Multi-Agent Reinforcement Learning (MARL)** scenario in a **GridWorld** environment using **Independent Q-Learning (IQL)**.

**Objectives:**

- Implement a **Multi-Agent GridWorld environment** with two agents — a **Tagger** and a **Runner** — that interact within a grid-based world.
- Apply the **Independent Q-Learning (IQL)** algorithm for both agents, enabling them to learn policies independently based on their individual reward signals.
- Train and evaluate both agents in the Multi-Agent GridWorld environment by tracking their **total rewards over time**.
- **Visualize the learned policies**, particularly how the runner's escape behavior evolves relative to the tagger's position.
- **Observe emergent competitive behaviors** that arise naturally, where one agent (the tagger) learns to catch, and the other (the runner) learns to evade.

## Software/Tool:

Google Colab

## Theory:

**Multi-Agent Reinforcement Learning (MARL)** extends the classical reinforcement learning framework to settings where **multiple agents** interact within a shared environment. Each agent simultaneously **learns its own policy** while adapting to the dynamic strategies of other agents.

In this experiment, we explore a **competitive multi-agent scenario**:

- The **Tagger** aims to catch the **Runner**.
- The **Runner** aims to escape the Tagger for as long as possible.

The environment is a **GridWorld**, where agents move in discrete directions (up, down, left, right) on a grid.

**1. Multi-Agent Reinforcement Learning Setup**

Each agent $i$ in the environment has its own components:

- **State space:** $S_i$ — the observable part of the environment (e.g., positions of both agents).
- **Action space:** $A_i$ — possible moves (up, down, left, right, stay).
- **Reward function:** $R_i(s, a_1, a_2)$ — depends on both agents' actions and positions.
- **Policy:** $\pi_i(a_i|s_i)$ — the strategy used by agent $i$ to select actions.

At each time step $t$:

1. Both agents observe the current joint state $s_t$.
2. Each agent selects an action $a_t^i$ according to its policy $\pi_i$.
3. The environment transitions to a new state $s_{t+1}$.
4. Each agent receives its own reward $r_t^i$.

## 2. Independent Q-Learning (IQL)

**Independent Q-Learning (IQL)** is one of the simplest and most widely used approaches in **multi-agent reinforcement learning**.
Each agent independently applies the standard **Q-learning** algorithm to learn its own action-value function, **treating other agents as part of the environment**.
The update rule for agent $i$ is given by the **Bellman Equation**:

$$Q_i(s_t, a_t^i) \leftarrow Q_i(s_t, a_t^i) + \alpha \left[ r_t^i + \gamma \max_{a'} Q_i(s_{t+1}, a') - Q_i(s_t, a_t^i) \right]$$

Where:

- $Q_i(s_t, a_t^i)$: Estimated value of taking action $a_t^i$ in state $s_t$.
- $\alpha$: Learning rate.
- $\gamma$: Discount factor.
- $r_t^i$: Reward received by agent $i$.
- $\max_{a'} Q_i(s_{t+1}, a')$: Estimated optimal future value.

Each agent updates its Q-table **independently**, even though the environment changes dynamically as other agents learn simultaneously.

## 3. Agent-Specific Reward Design

To promote **competitive learning**, distinct reward signals are assigned:

| Agent | Objective | Reward Function |
|---|---|---|
| **Tagger** | Catch the runner | +10 for catching the runner, -1 per step otherwise |
| **Runner** | Avoid capture | +1 per step alive, -10 if caught |

This creates a **zero-sum interaction**, where the success of one agent implies the failure of the other.

**4. Non-Stationarity Challenge**

In multi-agent settings, the environment is **non-stationary** from the perspective of any single agent, because the policies of other agents are changing over time.

Thus, the transition dynamics $P(s_{t+1}|s_t, a_t)$ become **non-stationary**, violating the Markov property assumed in classical Q-learning. Despite this, IQL can still work in simpler or well-structured environments like GridWorld.

**5. Learning Policies Independently**

Each agent follows an **ε-greedy policy** to balance exploration and exploitation:

$$a_t^i = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q_i(s_t, a), & \text{with probability } 1 - \epsilon \end{cases}$$

The ε-value typically decays over time, allowing the agent to explore early and exploit later as its Q-values converge.

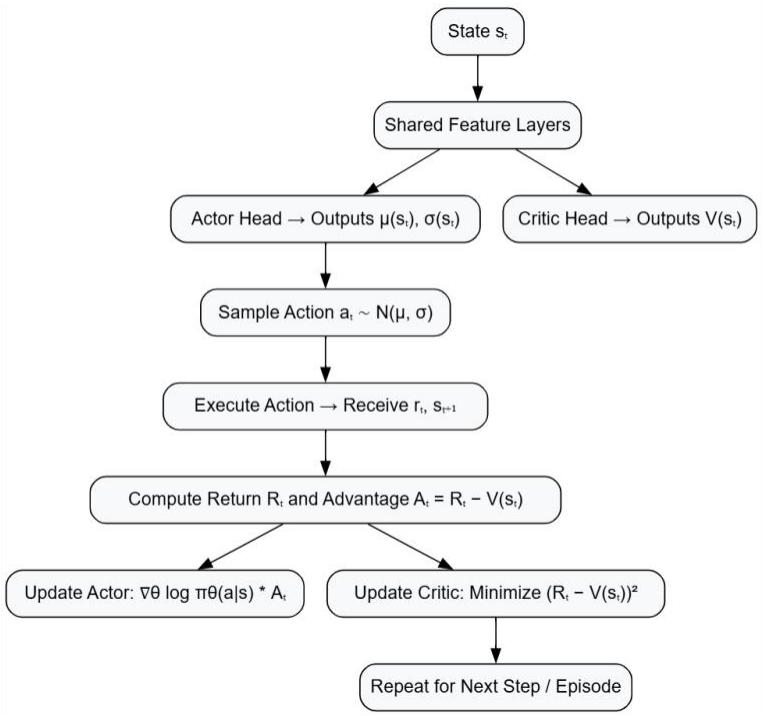
**6. Emergent Behavior**

As both agents independently optimize their own objectives, **emergent behaviors** arise:

- The **Tagger** learns to predict and intercept the Runner's movement path.
- The **Runner** learns evasive maneuvers, maintaining distance or moving to corner traps strategically.

**7. Diagram: Multi-Agent GridWorld and IQL Flow**

**Output and result:**



**Attachment:** Code and Results

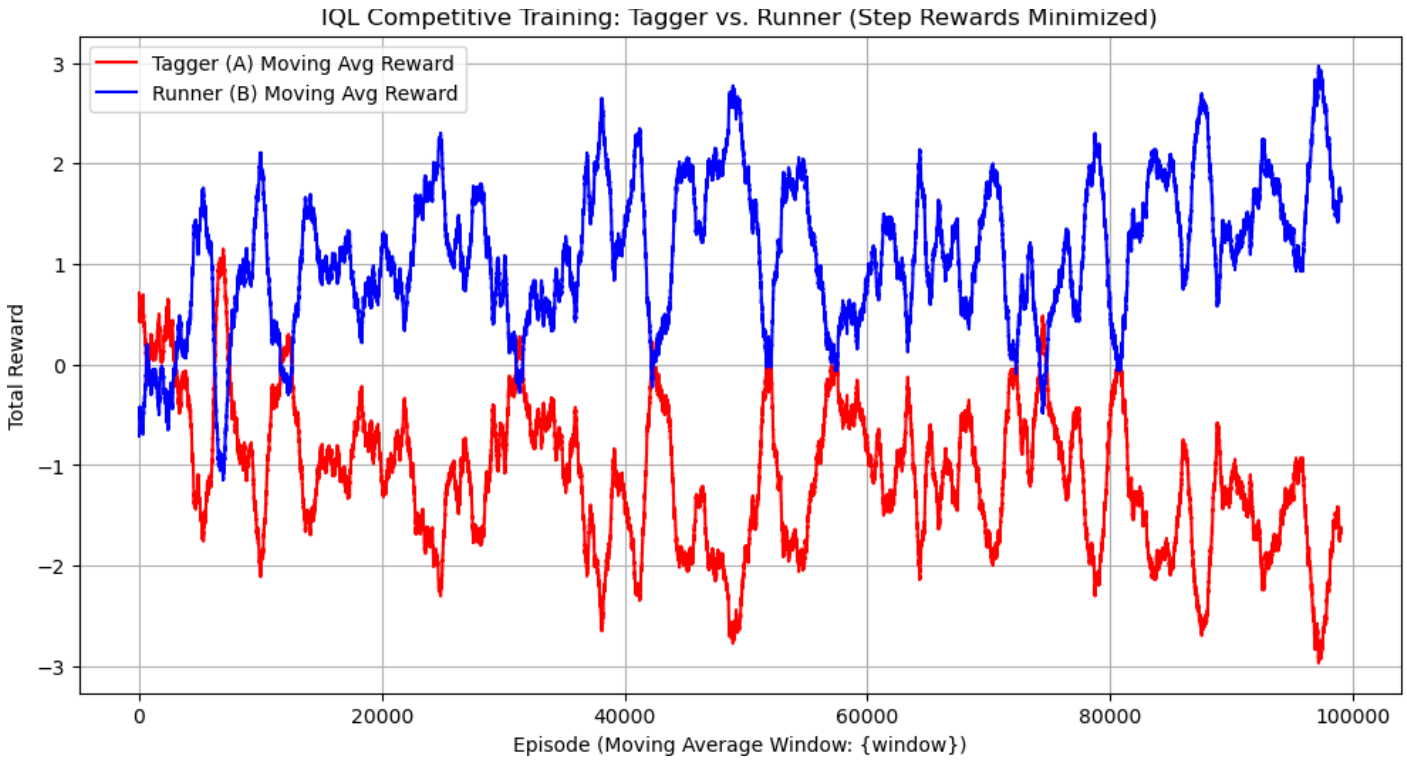
```
# Name: Kiran Biradar
# PRN: 2022802048

# Independent Q-Learning (IQL) for Tagger/Runner Competition
import numpy as np
import matplotlib.pyplot as plt from
collections import defaultdict
from typing import Dict, Tuple, List

# Define actions
UP, RIGHT, DOWN, LEFT = 0, 1, 2, 3
ACTION_TO_STR = {UP: "↑", RIGHT: "→", DOWN: "↓", LEFT: "←"}

# Multi-Agent GridWorld Environment (Tagger vs Runner)
class TaggerRunnerGridWorld:
    """
    GridWorld environment for two agents: Tagger (A) and Runner (B).

    NEW REWARDS (fixed for competitive learning):
    - Tagger (A): Rewards +10 (catch), -0.01 (step), -5 (time out).
    - Runner (B): Rewards -10 (caught), +0.01 (step), +5 (time out).
    """

    def __init__(self, rows=4, cols=4, max_steps=50):
        self.rows = rows
        self.cols = cols
        self.max_steps = max_steps
        self.current_step = 0
        self.reset()

    def reset(self):
        """Randomly initialize Tagger (A) and Runner (B) positions.
        Ensure agents start in different, non-terminal locations."""
        while True:
            r_a, c_a = np.random.randint(self.rows), np.random.randint(self.cols)
            r_b, c_b = np.random.randint(self.rows), np.random.randint(self.cols)
            if (r_a, c_a) != (r_b, c_b):
                self.agent_states = {'A': (r_a, c_a), 'B': (r_b, c_b)}
                break
        self.current_step  =  0
        return self._get_state()

    def _get_state(self):
        """Returns the combined state observation (Tagger_pos, Runner_pos)."""
        ra, ca = self.agent_states['A']
        rb, cb = self.agent_states['B']
        return (ra, ca, rb, cb)

    def _move(self, pos, action):
```

```
        """Calculates next position based on action, respecting boundaries."""
        r, c = pos
        if action == UP:
            r = max(r - 1, 0)
        elif action == RIGHT:
            c = min(c + 1, self.cols - 1)
        elif action == DOWN:
            r = min(r + 1, self.rows - 1)
        elif action == LEFT:
            c = max(c - 1, 0)
        return (r, c)

    def step(self, actions):
        """Apply actions for both agents and calculate competitive rewards."""
        self.current_step += 1

        # Calculate next positions
        next_states = {
            'A': self._move(self.agent_states['A'], actions['A']),
            'B': self._move(self.agent_states['B'], actions['B'])
        }

        # Check for termination and reward condition
        is_catch = (next_states['A'] == next_states['B'])
        max_steps_reached = (self.current_step >= self.max_steps)
        done = is_catch or max_steps_reached

        rewards = {}

        if is_catch:
            # Tagger (A) wins: Terminal, high positive reward
            rewards['A'] = +10
            rewards['B'] = -10
        elif max_steps_reached:
            # Runner (B) wins: Terminal, low magnitude reward
            rewards['A'] = -5
            rewards['B'] = +5
        else:
            # Step rewards (Low magnitude to prioritize terminal states)
            rewards['A'] = -0.01  # Step penalty for Tagger (pursuit focus)
            rewards['B'] = +0.01  # Evasion/Survival bonus for Runner

        self.agent_states = next_states.copy()
        return self._get_state(), rewards, done

# ϵ-greedy policy for state observation
def epsilon_greedy(Q: dict, state: tuple, epsilon: float = 0.1):
    """Select action using ε-greedy based on the full state tuple."""
    if np.random.rand() < epsilon:
        return np.random.choice([0, 1, 2, 3])
    else:
```

```
        # If state is unvisited, Q[state] will return zeros (via defaultdict setup)
        return np.argmax(Q[state])

# Independent Q-Learning (IQL) for Competition
def independent_q_learning(env: TaggerRunnerGridWorld, episodes=100000, alpha=0.1,
gamma=0.99, epsilon=0.1):
    """Independent Q-learning where each agent observes the full state."""
    # Q-tables mapping combined state (Ra, Ca, Rb, Cb) -> [Q(Up), Q(Right), ...]
    Q = {agent: defaultdict(lambda: np.zeros(4)) for agent in ['A', 'B']}
    rewards_history = {agent: [] for agent in ['A', 'B']}

    for ep in range(episodes):
        full_state = env.reset()
        done = False
        total_reward = {'A': 0, 'B': 0}

        while not done:
            # Agent A (Tagger) and Agent B (Runner) select actions independently
            actions = {
                'A': epsilon_greedy(Q['A'], full_state, epsilon),
                'B': epsilon_greedy(Q['B'], full_state, epsilon)
            }

            next_state, rewards, done = env.step(actions)

            # Q-learning update for each agent (IQL)
            for agent in ['A', 'B']:
                s = full_state
                a = actions[agent]
                r = rewards[agent]
                s_next = next_state

                # IQL Update: Q(s, a) <- Q(s, a) + alpha * [r + gamma * max_a' Q(s', a') -
Q(s, a)]
                Q[agent][s][a] += alpha * (r + gamma * np.max(Q[agent][s_next]) -
Q[agent][s][a])
                total_reward[agent] += r
```

```
            full_state = next_state

        rewards_history['A'].append(total_reward['A'])
        rewards_history['B'].append(total_reward['B'])

        # Log progress
        if (ep + 1) % 10000 == 0:
            print(f"Episode {ep + 1}/{episodes} | Avg R(Tagger):
{np.mean(rewards_history['A'][-10000:]):.2f} | Avg R(Runner): {np.mean(rewards_history['B'][-
10000:]):.2f}")

    return Q, rewards_history

#
# Visualization and Analysis
#

def visualize_policy_slice(Q_tables: dict, agent: str, env: TaggerRunnerGridWorld,
opponent_pos: tuple):
    """
    Visualizes the greedy policy for a fixed opponent position.
    """
    R_opp, C_opp = opponent_pos

    policy_map = np.full((env.rows, env.cols), " ")

    # Select the agent's specific Q-table
    Q = Q_tables[agent]

    # The agent's position (R_self, C_self) varies across the grid
    for R_self in range(env.rows):
        for C_self in range(env.cols):
            if agent == 'A':
                # Tagger observes its pos and Runner's fixed pos
                state = (R_self, C_self, R_opp, C_opp)
            else:
                # Runner observes its pos and Tagger's fixed pos
                state = (R_opp, C_opp, R_self, C_self)

            if (R_self, C_self) == (R_opp, C_opp):
                policy_map[R_self, C_self] = 'x' # Mark collision state
                continue

            Q_values = Q[state]
            if np.all(Q_values == 0):
                policy_map[R_self, C_self] = '?' # Unvisited
            else:
                best_a = np.argmax(Q_values)
                policy_map[R_self, C_self] = ACTION_TO_STR[best_a]

    # Mark the opponent's position
```

```
    policy_map[R_opp, C_opp] = 'O'

    print(f"\n--- Greedy Policy for {agent} when Opponent is at {(R_opp, C_opp)} ---")
    for row in policy_map:
        print(" ".join([f"{c:3}" for c in row]))

def plot_learning_progress(rewards_history: Dict[str, List[float]]):
    """Plots the moving average reward for both agents."""
    plt.figure(figsize=(12, 6))
    window = 1000

    # Tagger (Agent A)
    tagger_rewards = rewards_history['A']
    tagger_avg = np.convolve(tagger_rewards, np.ones(window)/window, mode='valid')
    plt.plot(tagger_avg, label='Tagger (A) Moving Avg Reward', color='red')

    # Runner (Agent B)
    runner_rewards = rewards_history['B']
    runner_avg = np.convolve(runner_rewards, np.ones(window)/window, mode='valid')
    plt.plot(runner_avg, label='Runner (B) Moving Avg Reward', color='blue')

    plt.xlabel('Episode (Moving Average Window: {window})')
    plt.ylabel('Total Reward')

    plt.title('IQL Competitive Training: Tagger vs. Runner (Step Rewards Minimized)')
    plt.legend()
    plt.grid(True)
    plt.show()

#
# Main Execution
#

if __name__ == "__main__":
    np.random.seed(42)
    env = TaggerRunnerGridWorld(rows=4, cols=4, max_steps=30)

    # Train the agents
    Q_tables, rewards_history = independent_q_learning(env, episodes=100000)

    # 1. Visualize Training Progress
    plot_learning_progress(rewards_history)

    # 2. Visualize Emergent Competitive Policies
    # Scenario 1: Runner (B) policy when Tagger (A) is near the middle (3, 2)
    # Observe Runner's evasion tactics (moving away from Tagger's fixed position)
    visualize_policy_slice(Q_tables, 'B', env, opponent_pos=(3,2))

    # Scenario 2: Tagger (A) policy when Runner (B) is at the opposite corner (0, 0)
    # Observe Tagger's pursuit behavior (moving towards Runner's fixed position)
    visualize_policy_slice(Q_tables, 'A', env, opponent_pos=(0,0))
```




```
--- Greedy Policy for B when Opponent is at (3, 2) ---
→   →   ↓   ←
↑   →   ↑   ↑
→   ↑   ↑   →
←   ↑   O   ↑

--- Greedy Policy for A when Opponent is at (0, 0) ---
O   ↑   →   ←
→   →   →   ↓
↑   ↑   ←   ←
↑   ←   ←   →

--- Emergent Behavior Observation (Competitive Check) ---
Tagger Final Avg Reward: -1.62
Runner Final Avg Reward: 1.62

⚠️ Agents failed to learn clear competitive strategies. (The competitive dynamics might not have emerged clearly.)
```

## Conclusion: