Experiment No 1

Code:

```
import numpy as np
import random


class GridWorld:
    def __init__(self, size=5):
        self.size = size
        self.grid = np.zeros((size, size))
        self.start = (0, 0)
        self.goal = (size - 1, size - 1)
        self.obstacles = [(1, 1), (1, 2), (2, 1)]


    def is_valid_state(self, state):
        x, y = state
        return 0 <= x < self.size and 0 <= y < self.size and state not in self.obstacles


    def get_possible_actions(self, state):
        actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        possible_actions = []
        for dx, dy in actions:
            new_state = (state[0] + dx, state[1] + dy)
            if self.is_valid_state(new_state):
                possible_actions.append((dx, dy))
        return possible_actions
```

```
    def take_action(self, state, action):
        x, y = state
        dx, dy = action
        new_state = (x + dx, y + dy)
        if self.is_valid_state(new_state):
            return new_state, -1
        else:
            return state, -1

    def is_goal(self, state):
        return state == self.goal

def q_learning(env, num_episodes, learning_rate=0.1, discount_factor=0.9,
exploration_rate=0.1):
    q_table = {}
    for i in range(env.size):
        for j in range(env.size):
            q_table[(i,j)] = {}
            for action in [(0,1),(0,-1),(1,0),(-1,0)]:
                if (i, j) not in env.obstacles:
                    q_table[(i, j)][action] = 0.0

    for episode in range(num_episodes):
        state = env.start
        while not env.is_goal(state):
            possible_actions = env.get_possible_actions(state)
            if random.uniform(0, 1) < exploration_rate:
```

```
            action = random.choice(possible_actions)
        else:
            action = max(possible_actions, key=lambda a: q_table[state][a])

        next_state, reward = env.take_action(state, action)

        max_future_q = max([q_table[next_state][a] for a in
env.get_possible_actions(next_state)], default=0)
        q_table[state][action] = q_table[state][action] + learning_rate * (reward +
discount_factor * max_future_q - q_table[state][action])

        state = next_state

    return q_table

env = GridWorld()
q_table = q_learning(env, num_episodes=1000)

state = env.start
while not env.is_goal(state):
    possible_actions = env.get_possible_actions(state)
    action = max(possible_actions, key=lambda a: q_table[state][a])
    next_state, reward = env.take_action(state,action)
    print(f"Current state:{state}, Action taken: {action}, Next state: {next_state}")
    state = next_state
print("Reached Goal!")
```

Output:

Current state:(0, 0), Action taken: (0, 1), Next state: (0, 1)

Current state:(0, 1), Action taken: (0, 1), Next state: (0, 2)

Current state:(0, 2), Action taken: (0, 1), Next state: (0, 3)

Current state:(0, 3), Action taken: (1, 0), Next state: (1, 3)

Current state:(1, 3), Action taken: (1, 0), Next state: (2, 3)

Current state:(2, 3), Action taken: (1, 0), Next state: (3, 3)

Current state:(3, 3), Action taken: (0, 1), Next state: (3, 4)

Current state:(3, 4), Action taken: (1, 0), Next state: (4, 4)

Reached Goal!

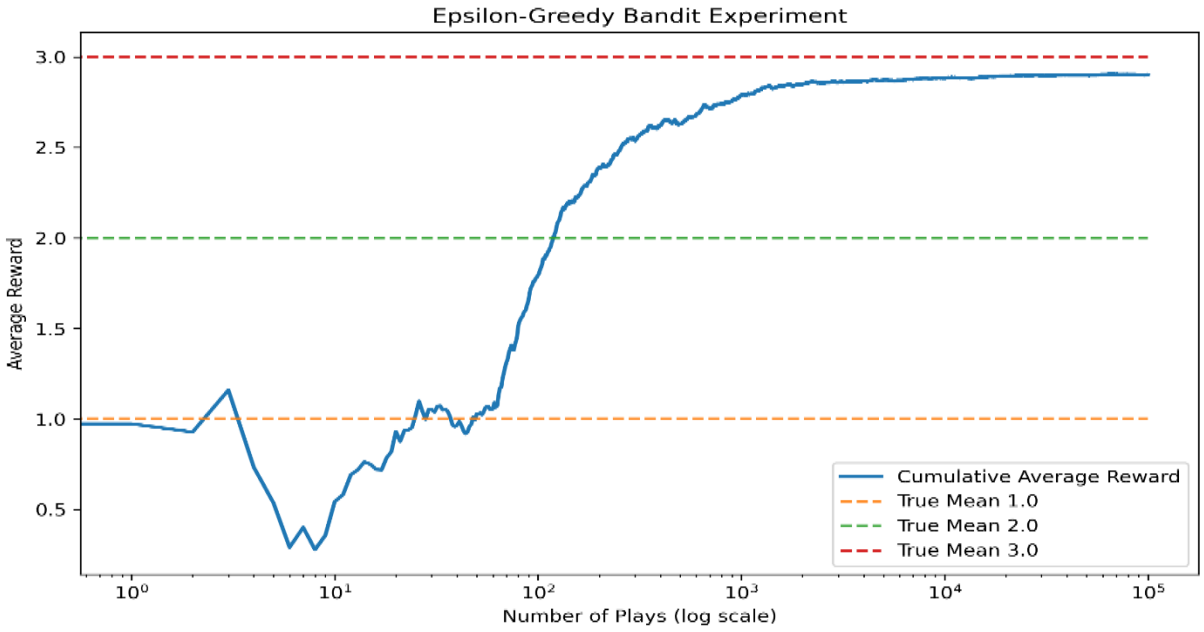
Experiment No 2

Code:

```
import numpy as np
import matplotlib.pyplot as plt
class Bandit:
    def __init__(self, m):
        self.m = m
        self.mean = 0
        self.N = 0
    def pull(self):
        return np.random.randn() + self.m
    def update(self, x):
        self.N += 1
        self.mean = (1 - 1.0 / self.N) * self.mean + 1.0 / self.N * x
def run_experiment(m1, m2, m3, eps, N):
    bandits = [Bandit(m1), Bandit(m2), Bandit(m3)]
    data = np.empty(N)
    for i in range(N):
        p = np.random.random()
        if p < eps:
            j = np.random.choice(3)
        else:
            j = np.argmax([b.mean for b in bandits])
        x = bandits[j].pull()
```

```
        bandits[j].update(x)
        data[i] = x
    cumulative_average = np.cumsum(data) / (np.arange(N) + 1)
    plt.plot(cumulative_average)
    plt.plot(np.ones(N) * m1)
    plt.plot(np.ones(N) * m2)
    plt.plot(np.ones(N) * m3)
    plt.xscale('log')
    plt.show()
    return cumulative_average
if __name__ == '__main__':
    m1 = 1.0
    m2 = 2.0
    m3 = 3.0
    eps = 0.1
    N = 100000
    cumulative_average = run_experiment(m1, m2, m3, eps, N)
    print("Estimated means:", [b.mean for b in [Bandit(m1), Bandit(m2), Bandit(m3)]])
    print("True means:", [m1, m2, m3])
    print("Number of times each bandit was played:", [b.N for b in [Bandit(m1),
Bandit(m2), Bandit(m3)]])
```

Output:



Estimated means: [0, 0, 0]

True means: [1.0, 2.0, 3.0]

Number of times each bandit was played: [0, 0, 0]

Experiment No 3

Code:

```
import numpy as np

class GridWorld:
    def __init__(self, rows, cols, start, goal, obstacles):
        self.rows = rows
        self.cols = cols
        self.start = start
        self.goal = goal
        self.obstacles = obstacles
        self.actions = ['up', 'down', 'left', 'right']
        self.rewards = {goal: 100, **{obstacle: -100 for obstacle in obstacles}}
        self.state_space = [(i, j) for i in range(rows) for j in range(cols) if (i,j) not in
obstacles]
    def get_possible_actions(self, state):
        row, col = state
        actions = []
        for action in self.actions:
            new_row, new_col = row, col
            if action == 'up':
                new_row = max(0, row - 1)
            elif action == 'down':
                new_row = min(self.rows - 1, row + 1)
            elif action == 'left':
                new_col = max(0, col - 1)
```

```
            elif action == 'right':
                new_col = min(self.cols - 1, col + 1)
            if (new_row, new_col) not in self.obstacles:
              actions.append(action)
        return actions
    def next_state(self, state, action):
        row, col = state
        if action == 'up':
            row = max(0, row - 1)
        elif action == 'down':
            row = min(self.rows - 1, row + 1)
        elif action == 'left':
            col = max(0, col - 1)
        elif action == 'right':
            col = min(self.cols - 1, col + 1)
        if (row, col) in self.obstacles:
          return state
        return (row, col)
    def get_reward(self, state):
        return self.rewards.get(state, -1)
rows, cols = 5, 5
start = (0, 0)
goal = (4, 4)
obstacles = [(1,1), (1,2)]
grid_world = GridWorld(rows, cols, start, goal, obstacles)
print(grid_world.get_possible_actions((0,0)))
print(grid_world.next_state((0,0), "right"))
```

```
print(grid_world.get_reward((4,4)))

grid_world.state_space
```

Output:

(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (1, 0) (1, 3) (1, 4) (2, 0) (2, 1) (2, 2) (2, 3)

(2, 4) (3, 0) (3, 1) (3, 2) (3, 3) (3, 4) (4, 0) (4, 1) (4, 2) (4, 3) (4, 4)

Experiment No 4

Code:

```
import numpy as np
def policy_evaluation(grid_world, policy, gamma=0.9, theta=0.0001):
    V = {state: 0 for state in grid_world.state_space}
    while True:
        delta = 0
        for state in grid_world.state_space:
            v = V[state]
            new_v = 0
            for action in grid_world.get_possible_actions(state):
                next_state = grid_world.next_state(state, action)
                reward = grid_world.get_reward(next_state)
                new_v += policy.get(state,{}).get(action,0) * (reward + gamma * V[next_state])
            V[state] = new_v
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break
    return V
def policy_improvement(grid_world, V, gamma=0.9):
    policy = {}
    for state in grid_world.state_space:
      policy[state] = {}
      best_action = None
```

```
        best_value = -np.inf
        for action in grid_world.get_possible_actions(state):
            next_state = grid_world.next_state(state, action)
            reward = grid_world.get_reward(next_state)
            value = reward + gamma * V[next_state]
            if value > best_value:
                best_value = value
                best_action = action
        if best_action:
            policy[state][best_action] = 1
    return policy
rows, cols = 5, 5
start = (0, 0)
goal = (4, 4)
obstacles = [(1,1), (1,2)]
grid_world = GridWorld(rows, cols, start, goal, obstacles)
initial_policy = {}
for state in grid_world.state_space:
    possible_actions = grid_world.get_possible_actions(state)
    if possible_actions:
        initial_policy[state] = {action: 1/len(possible_actions) for action in possible_actions}
policy = initial_policy
for i in range(10):
    V = policy_evaluation(grid_world, policy)
    policy = policy_improvement(grid_world, V)
print(V)
policy
```

Output:

{(0, 0): 473.07904816898986, (0, 1): 526.7545891689899, (0, 2): 586.39407916899, (0, 3): 652.6601791689899, (0, 4): 726.2891791689898, (1, 0): 526.7545891689899, (1, 3): 726.2891791689898, (1, 4): 808.0991791689898, (2, 0): 586.39407916899, (2, 1): 652.6601791689899, (2, 2): 726.2891791689898, (2, 3): 808.0991791689898, (2, 4): 898.9991791689898, (3, 0): 652.6601791689899, (3, 1): 726.2891791689898, (3, 2): 808.0991791689898, (3, 3): 898.9991791689898, (3, 4): 999.9991791689898, (4, 0): 726.2891791689898, (4, 1): 808.0991791689898, (4, 2): 898.9991791689898, (4, 3): 999.9991791689898, (4, 4): 999.9991791689898}

(0, 0): {'down': 1} (1, 0): {'down': 1}

(0, 1): {'right': 1} (1, 3): {'down': 1}

(0, 2): {'right': 1} (1, 4): {'down': 1}

(0, 3): {'down': 1} (2, 0): {'down': 1}

(0, 4): {'down': 1} (2, 1): {'down': 1}

(2, 2): {'down': 1} (3, 0): {'down': 1}

(2, 3): {'down': 1} (3, 1): {'down': 1}

(2, 4): {'down': 1} (3, 2): {'down': 1}

(3, 3): {'down': 1} (3, 4): {'down': 1}

(4, 0): {'right': 1} (4, 1): {'right': 1}

(4, 2): {'right': 1} (4, 3): {'right': 1}

(4, 4): {'down': 1}

Experiment No 5

Code:

```
import numpy as np
import random
class GridWorld:
    def __init__(self, size=5):
        self.size = size
        self.actions = ['up', 'down', 'left', 'right']
        self.terminal_states = [(0, 0), (size - 1, size - 1)]
    def reset(self):
        return (0, self.size-1)
    def step(self, state, action):
        x, y = state
        if action == 'up':
            y = min(self.size-1, y+1)
        elif action == 'down':
            y = max(0, y - 1)
        elif action == 'left':
            x = max(0, x-1)
        elif action == 'right':
            x = min(self.size - 1, x + 1)
        next_state = (x,y)
        reward = -1 if next_state not in self.terminal_states else 0
        done = True if next_state in self.terminal_states else False
```

```
        return next_state, reward, done
def mc_control(env, num_episodes=500, epsilon=0.1, gamma=1.0):
    Q = {}
    returns = {}
    for _ in range(num_episodes):
        episode = []
        state = env.reset()
        done = False
        while not done:
            if tuple(state) not in Q:
                Q[tuple(state)] = {a: 0 for a in env.actions}
            action = random.choice(env.actions)
            next_state, reward, done = env.step(state, action)
            episode.append((state, action, reward))
            state = next_state
        G = 0
        for t in reversed(range(len(episode))):
            state, action, reward = episode[t]
            G = gamma * G + reward
            if (tuple(state), action) not in returns:
                returns[(tuple(state), action)] = []
            returns[(tuple(state), action)].append(G)
            Q[tuple(state)][action] = np.mean(returns[(tuple(state), action)])
    return Q
env = GridWorld(size=5)
Q = mc_control(env)
print("Example Q-values:")
```

```
for s,actions in list(Q.items())[:2]:
  print(f"State: {s}, Q-values: {actions}")
```

Output:

Example Q-values:

State: (0, 4), Q-values: {'up': -39.714467005076145, 'down': -38.493688639551195, 'left': -42.13380281690141, 'right': -37.72080536912752}

State: (0, 3), Q-values: {'up': -39.85776805251641, 'down': -36.233890214797135, 'left': -39.78997613365155, 'right': -37.427947598253276}

Experiment No 6

Code:

```
import random
class Bandit:
    def __init__(self, arms):
        self.arms = arms
        self.true_rewards = [random.gauss(0, 1) for _ in range(arms)]
    def pull(self, arm):
        return random.gauss(self.true_rewards[arm], 1)
def experiment(bandit, strategy, iterations):
    rewards = []
    arm_counts = [0] * bandit.arms
    for _ in range(iterations):
        if strategy == 'epsilon-greedy':
            if random.random() < 0.1:
                arm = random.randrange(bandit.arms)
            else:
                arm = max(range(bandit.arms), key=lambda x: arm_counts[x]) if
sum(arm_counts) > 0 else 0
        elif strategy == 'greedy':
            arm = max(range(bandit.arms), key=lambda x: arm_counts[x]) if
sum(arm_counts) > 0 else 0
        elif strategy == 'random':
            arm = random.randrange(bandit.arms)
        reward = bandit.pull(arm)
```

```
        rewards.append(reward)
        arm_counts[arm] += 1
    return rewards, arm_counts
bandit = Bandit(10)
strategies = ['epsilon-greedy', 'greedy', 'random']
iterations = 1000
for strategy in strategies:
    rewards, arm_counts = experiment(bandit, strategy, iterations)
    print(f"Strategy: {strategy}")
    print(f"Total Reward: {sum(rewards)}")
    print(f"Arm counts: {arm_counts}\n")
```

Output:

Strategy: epsilon-greedy

Total Reward: -552.9616399755109

Arm counts: [913, 4, 12, 13, 13, 8, 12, 5, 9, 11]

Strategy: greedy

Total Reward: -602.0009247533192

Arm counts: [1000, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Strategy: random

Total Reward: 34.58277808080329

Arm counts: [98, 96, 95, 103, 109, 91, 122, 97, 95, 94]

Experiment No 7

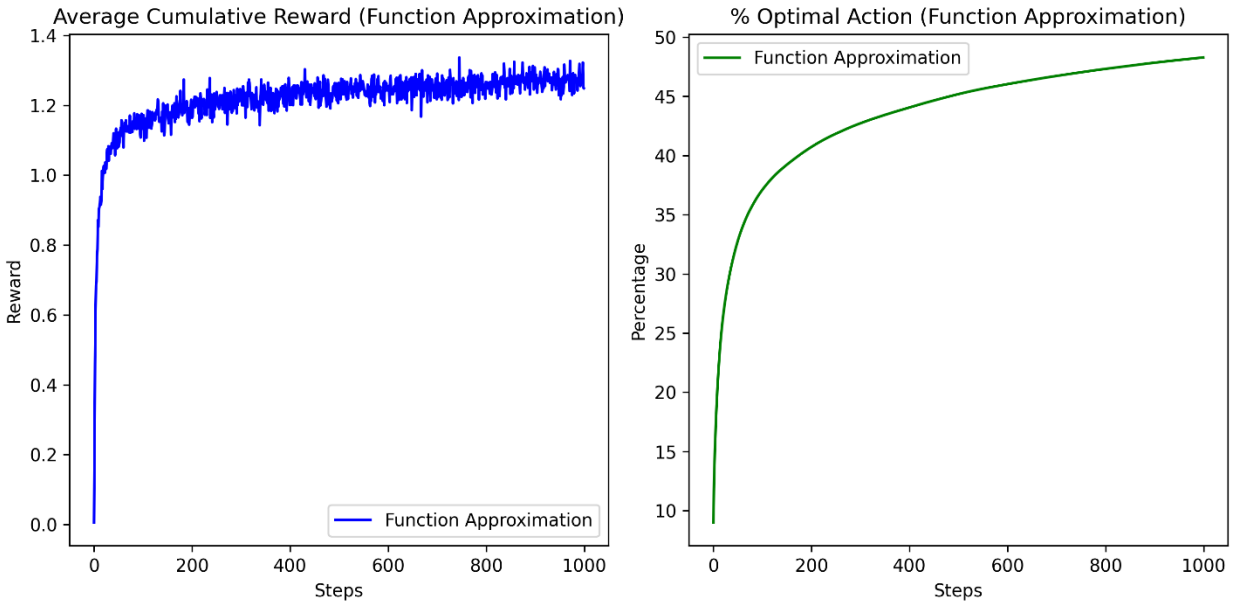
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
class Bandit:
    def __init__(self, k_arms=10):
        self.k = k_arms
        self.means = np.random.randn(k_arms)
        self.optimal_arm = np.argmax(self.means)
    def pull(self, arm):
        """Return a random reward based on the selected arm's mean."""
        return np.random.randn() + self.means[arm]
def run_experiment_fa(bandit, n_steps=1000, learning_rate=0.1):
    features = np.eye(bandit.k)
    weights = np.zeros(bandit.k)
    rewards = []
    optimal_action_counts = []
    for t in range(1, n_steps + 1):
        predicted_values = features @ weights
        arm = np.argmax(predicted_values)
        reward = bandit.pull(arm)
        rewards.append(reward)
        error = reward - predicted_values[arm]
```

```
        weights += learning_rate * error * features[arm]
        optimal_action_counts.append(1 if arm == bandit.optimal_arm else 0)
    return np.array(rewards), np.array(optimal_action_counts)
if __name__ == "__main__":
    np.random.seed(42)
    n_experiments = 2000
    n_steps = 1000
    cumulative_rewards_fa = np.zeros(n_steps)
    optimal_action_percentages_fa = np.zeros(n_steps)
    for _ in range(n_experiments):
        bandit = Bandit(k_arms=10)
        rewards, optimal_action_counts = run_experiment_fa(bandit, n_steps)
        cumulative_rewards_fa += rewards
        optimal_action_percentages_fa += np.cumsum(optimal_action_counts)
    cumulative_rewards_fa /= n_experiments
    optimal_action_percentages_fa /= n_experiments
    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.plot(cumulative_rewards_fa, label='Function Approximation', color='b')
    plt.title("Average Cumulative Reward (Function Approximation)")
    plt.xlabel("Steps")
    plt.ylabel("Reward")
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(optimal_action_percentages_fa * 100 / np.arange(1, n_steps + 1),
             label="Function Approximation", color='g')
    plt.title("% Optimal Action (Function Approximation)")
```

```
plt.xlabel("Steps")
plt.ylabel("Percentage")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:



Experiment No 8

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import random
class SimplePongEnv:
    def __init__(self):
        self.observation_shape = (84, 84, 1)
        self.action_space = 6
        self.reset()
    def reset(self):
        self.state = np.zeros(self.observation_shape)
        self.steps = 0
        return self.state
    def step(self, action):
        self.steps += 1
        reward = 1 if self.steps % 10 == 0 else 0
        done = self.steps >= 50
        return np.zeros(self.observation_shape), reward, done
class DQNAgent:
    def __init__(self, state_shape, action_size):
        self.action_size = action_size
        self.epsilon = 1.0
```

```
        self.epsilon_min = 0.1
        self.epsilon_decay = 0.995
        self.gamma = 0.99
        self.model = self._build_model(state_shape)
    def _build_model(self, state_shape):
        model = tf.keras.Sequential([
            layers.Flatten(input_shape=state_shape),
            layers.Dense(24, activation='relu'),
            layers.Dense(24, activation='relu'),
            layers.Dense(self.action_size, activation='linear')
        ])
        model.compile(optimizer='adam', loss='mse')
        return model
    def act(self, state):
        if random.random() < self.epsilon:
            return random.randint(0, self.action_size - 1)
        q_values = self.model.predict(state[np.newaxis], verbose=0)
        return np.argmax(q_values[0])
    def train(self, state, action, reward, next_state, done):
        target = reward + (1 - done) * self.gamma *
np.max(self.model.predict(next_state[np.newaxis], verbose=0)[0])
        target_f = self.model.predict(state[np.newaxis], verbose=0)
        target_f[0][action] = target
        self.model.fit(state[np.newaxis], target_f, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
env = SimplePongEnv()
```

```
agent = DQNAgent(env.observation_shape, env.action_space)
episodes = 100
for episode in range(episodes):
    state = env.reset()
    total_reward = 0
    while True:
        action = agent.act(state)
        next_state, reward, done = env.step(action)
        agent.train(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        if done:
            break
    print(f"Episode {episode + 1}: Total Reward: {total_reward}")
```

Output:

Episode 1: Total Reward: 5

Episode 2: Total Reward: 5

Episode 3: Total Reward: 5

Episode 4: Total Reward: 5

Episode 5: Total Reward: 5

Episode 6: Total Reward: 5

Episode 7: Total Reward: 5

Episode 8: Total Reward: 5

Episode 9: Total Reward: 5

.

.

.

.

Episode 96: Total Reward: 5

Episode 97: Total Reward: 5

Episode 98: Total Reward: 5

Episode 99: Total Reward: 5

Episode 100: Total Reward: 5

Experiment No 9

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
class SimpleEnv:
    def __init__(self, variation=0.0):
        self.state_dim = 3
        self.action_dim = 1
        self.action_bound = 1.0
        self.variation = variation
    def reset(self):
        return np.random.uniform(-1, 1, self.state_dim)
    def step(self, action):
        state = np.random.uniform(-1, 1, self.state_dim) + self.variation
        reward = -np.sum(state ** 2) - np.sum(action ** 2)
        done = np.random.rand() < 0.1
        return state, reward, done
class PolicyNetwork:
    def __init__(self, state_dim, action_dim, action_bound):
        self.model = tf.keras.Sequential([
            layers.Dense(32, activation='relu', input_shape=(state_dim,)),
            layers.Dense(32, activation='relu'),
            layers.Dense(action_dim, activation='tanh')
```

```
        ])
        self.optimizer = tf.keras.optimizers.Adam(0.01)
        self.action_bound = action_bound
    def get_action(self, state):
        action = self.model(state[np.newaxis], training=False)[0]
        return action * self.action_bound
    def train(self, states, actions, rewards):
        with tf.GradientTape() as tape:
            loss = -tf.reduce_mean(rewards * tf.reduce_sum(actions * self.model(states),
axis=1))
        grads = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
def train_policy(env, policy, episodes=10, gamma=0.99):
    for ep in range(episodes):
        state = env.reset()
        states, actions, rewards = [], [], []
        while True:
            action = policy.get_action(state)
            next_state, reward, done = env.step(action)
            states.append(state)
            actions.append(action)
            rewards.append(reward)
            state = next_state
            if done:
                break
        discounted_rewards = []
        cumulative_reward = 0
```

```
        for reward in reversed(rewards):
            cumulative_reward = reward + gamma * cumulative_reward
            discounted_rewards.insert(0, cumulative_reward)
        discounted_rewards = np.array(discounted_rewards)
        policy.train(np.array(states), np.array(actions), discounted_rewards)
        print(f"Episode {ep + 1}: Total Reward = {np.sum(rewards):.2f}")
print("Training on Source Environment")
source_env = SimpleEnv(variation=0.0)
policy = PolicyNetwork(source_env.state_dim, source_env.action_dim,
source_env.action_bound)
train_policy(source_env, policy, episodes=20)
print("\nTransferring to Target Environment")
target_env = SimpleEnv(variation=0.5)
train_policy(target_env, policy, episodes=20)
```

Output:

Episode 1: Total Reward = -2.87

Episode 2: Total Reward = -19.39

Episode 3: Total Reward = -4.94

Episode 4: Total Reward = -2.52

Episode 5: Total Reward = -17.37

Episode 6: Total Reward = -7.03

Episode 7: Total Reward = -7.06

Episode 8: Total Reward = -15.48

Episode 9: Total Reward = -13.03

Episode 10: Total Reward = -5.48

Episode 11: Total Reward = -6.82

Episode 12: Total Reward = -11.87

Episode 13: Total Reward = -3.84

Episode 14: Total Reward = -6.60

Episode 15: Total Reward = -13.10

Episode 16: Total Reward = -0.98

Episode 17: Total Reward = -6.76

Episode 18: Total Reward = -5.19

Episode 19: Total Reward = -5.17

Episode 20: Total Reward = -10.01

Transferring to Target Environment

Episode 1: Total Reward = -10.08

Episode 2: Total Reward = -4.19

Episode 3: Total Reward = -10.41

Episode 4: Total Reward = -29.40

Episode 5: Total Reward = -9.24

Episode 6: Total Reward = -1.80

Episode 7: Total Reward = -5.02

Episode 8: Total Reward = -6.90

Episode 9: Total Reward = -15.76

Episode 10: Total Reward = -20.82

Episode 11: Total Reward = -3.05

Episode 12: Total Reward = -2.41

Episode 13: Total Reward = -3.57

Episode 14: Total Reward = -46.61

Episode 15: Total Reward = -27.68

Episode 16: Total Reward = -0.38

Episode 17: Total Reward = -12.31

Episode 18: Total Reward = -10.50

Episode 19: Total Reward = -16.29

Episode 20: Total Reward = -2.96

Experiment No 10

Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
class SimpleEnv:
    def __init__(self):
        self.state_dim = 3
        self.action_dim = 1
        self.action_bound = 1.0
    def reset(self):
        return np.random.uniform(-1, 1, self.state_dim)
    def step(self, action):
        state = np.random.uniform(-1, 1, self.state_dim)
        reward = -np.sum(state ** 2) - np.sum(action ** 2)
        done = np.random.rand() < 0.1
        return state, reward, done
class PolicyNetwork:
    def __init__(self, state_dim, action_dim, action_bound):
        self.model = tf.keras.Sequential([
            layers.Dense(32, activation='relu', input_shape=(state_dim,)),
            layers.Dense(32, activation='relu'),
            layers.Dense(action_dim, activation='tanh')
        ])
```

```
        self.optimizer = tf.keras.optimizers.Adam(0.01)
        self.action_bound = action_bound
    def get_action(self, state):
        action = self.model(state[np.newaxis], training=False)[0]
        return action * self.action_bound
    def train(self, states, actions, rewards):
        with tf.GradientTape() as tape:
            loss = -tf.reduce_mean(rewards * tf.reduce_sum(actions * self.model(states),
axis=1))
        grads = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
def train_policy_gradient(episodes=10, gamma=0.99):
    env = SimpleEnv()
    policy = PolicyNetwork(env.state_dim, env.action_dim, env.action_bound)
    for ep in range(episodes):
        state = env.reset()
        states, actions, rewards = [], [], []
        while True:
            action = policy.get_action(state)
            next_state, reward, done = env.step(action)
            states.append(state)
            actions.append(action)
            rewards.append(reward)
            state = next_state
            if done:
                break
        discounted_rewards = []
```

```
        cumulative_reward = 0
        for reward in reversed(rewards):
            cumulative_reward = reward + gamma * cumulative_reward
            discounted_rewards.insert(0, cumulative_reward)
        discounted_rewards = np.array(discounted_rewards)
        policy.train(np.array(states), np.array(actions), discounted_rewards)
        print(f"Episode {ep + 1}: Total Reward = {np.sum(rewards):.2f}")
train_policy_gradient(episodes=20)
```

Output:

Episode 1: Total Reward = -5.97

Episode 2: Total Reward = -11.36

Episode 3: Total Reward = -24.41

Episode 4: Total Reward = -1.10

Episode 5: Total Reward = -3.29

Episode 6: Total Reward = -28.92

Episode 7: Total Reward = -31.18

Episode 8: Total Reward = -5.33

.

.

.

Episode 17: Total Reward = -4.14

Episode 18: Total Reward = -8.45

Episode 19: Total Reward = -31.44

Episode 20: Total Reward = -36.34