Project report

Gridworld Navigation with Q-Learning and Policy Gradient Methods

By - Navneet Pratap Singh (21BCE7832)

Introduction:

Imagine you are on a treasure hunt in a maze. Each step you take brings you closer to finding the treasure, but there are traps and dead ends you need to avoid. How would you figure out the best way to navigate the maze and find the treasure quickly? This is similar to what reinforcement learning (RL) does.

In this project, we explore how to teach a computer to navigate a simple maze, called Gridworld, using two powerful learning techniques: Q-Learning and Policy Gradient. The Gridworld is like a small 5x5 chessboard where our computer agent needs to move from the starting square to the treasure square, avoiding obstacles along the way.

To make the agent smart, we use special methods that help it learn the best moves. Q-Learning uses a magical formula called the Bellman Equation to update a table (the Q-table) that tells the agent the best action to take in each situation. On the other hand, the Policy Gradient method uses a neural network (a type of computer brain) to directly learn the best strategy for reaching the treasure.

We measure how well our agent learns by looking at how fast it learns, how much reward it gets for each move, and how often it successfully finds the treasure. By comparing the two methods, we discover their strengths and weaknesses, helping us understand when to use each one.

Our findings provide a fun and insightful way to understand basic RL concepts, setting the stage for exploring more advanced techniques like deep Q-learning in the future. Through this project, we not only teach our agent to become a treasure-hunting expert but also gain valuable knowledge about the fascinating world of reinforcement learning.

Literature review:

Reinforcement Learning (RL) is a subfield of machine learning where agents learn to make decisions by interacting with an environment to maximize cumulative rewards. Over the years, significant research has been conducted to develop and refine RL algorithms, leading to the emergence of various methodologies. In this literature review, we explore foundational concepts, significant advancements, and practical applications relevant to the project of training an RL agent in a Gridworld environment using Q-Learning and Policy Gradient methods.

Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) form the mathematical framework underpinning RL. Bellman (1957) introduced MDPs, where decisions are made to maximize rewards over time. An MDP is defined by a set of states, a set of actions, transition probabilities, and a reward function. MDPs provide the theoretical basis for RL algorithms by modelling the environment as a series of state transitions governed by probabilistic rules.

Q-Learning

Q-Learning, introduced by Watkins and Dayan (1992), is one of the most widely used value-based RL algorithms. It focuses on learning the optimal action-value function, known as the Q-function, which estimates the expected utility of taking an action in a given state and following the optimal policy thereafter. The Q-function is updated iteratively using the Bellman Equation, allowing the agent to improve its estimates based on the rewards received from the environment. Q-Learning is model-free, meaning it does not require knowledge of the environment's dynamics, making it highly versatile.

Bellman Equation

The Bellman Equation, named after Richard Bellman, is central to dynamic programming and RL. It provides a recursive decomposition of the value function, enabling efficient computation of optimal policies. In the context of Q-Learning, the Bellman Equation is used to update the Q-values based on observed rewards and estimated future rewards. The equation ensures that the Q-values converge to the optimal values as the agent explores the environment.

Policy Gradient Methods

Policy Gradient methods represent a different approach to RL, focusing on optimizing the policy directly rather than the value function. Williams (1992) introduced the REINFORCE algorithm, a foundational policy gradient method. These methods use gradient ascent to optimize the policy parameters by maximizing the expected cumulative reward. Policy Gradient methods are particularly useful in environments with large or continuous action spaces, where value-based methods may struggle.

Exploration Strategies

Effective exploration is critical for RL agents to discover optimal policies. The ε-greedy strategy, where the agent occasionally selects random actions, is a simple yet effective exploration method. Another approach, Upper Confidence Bound (UCB), balances exploration and exploitation by considering the uncertainty in the estimated action values. These strategies ensure that the agent explores sufficiently to avoid suboptimal policies while exploiting known high-reward actions to maximize cumulative rewards.

Applications in Gridworld

Gridworld environments serve as a common testbed for evaluating RL algorithms due to their simplicity and ease of visualization. They provide a controlled setting for studying the convergence properties and efficiency of different RL methods. Prior research has used Gridworlds to demonstrate the effectiveness of Q-Learning and Policy Gradient methods, offering insights into their relative strengths and weaknesses.

Comparative Studies

Several studies have compared Q-Learning and Policy Gradient methods in various environments. For instance, Mnih et al. (2015) demonstrated the superiority of deep Q-networks (a variant of Q-Learning) in certain Atari games, while Schulman et al. (2017) showed the effectiveness of Proximal Policy Optimization (PPO), a policy gradient method, in complex continuous control tasks. These comparisons highlight the trade-offs between value-based and policy-based approaches, guiding the choice of algorithm based on the task at hand.

Overview

In this project, we aim to create a Reinforcement Learning (RL) agent that can learn to navigate a simple Gridworld environment efficiently. The agent will primarily use Q-Learning to discover optimal actions, and we will also implement a Policy Gradient approach to compare the performance of these two methods. This project covers several key topics fundamental to RL.

Key Topics Covered:

1. **MDP Model:** The Gridworld is defined as a Markov Decision Process (MDP). This involves specifying states (grid positions), actions (moves in the grid), rewards (gains for reaching the goal or penalties for hitting obstacles), and transition probabilities (how the agent moves from one state to another). By modelling Gridworld as an MDP, we provide a structured framework for the agent to learn and make decisions.

2. **Value Function-Based Methods:** We implement Q-Learning, a popular value function-based method, where the agent learns a Q-value for each state-action pair. This value represents the expected reward for taking a specific action from a given state. The agent uses these Q-values to make decisions, guiding it towards the goal more efficiently over time.

3. **Policy Gradient:** In addition to Q-Learning, we explore a simple Policy Gradient method. Unlike Q-Learning, which updates value functions, Policy Gradient directly optimizes the policy (the strategy of choosing actions) by adjusting the parameters of a neural network. This approach provides an alternative way to determine the best actions for the agent.

4. **Bellman Equation:** A critical component in the Q-Learning algorithm, the Bellman Equation, is used to update the Q-values. It expresses the relationship between the Q-value of a state-action pair and the expected rewards from subsequent states. By iteratively applying the Bellman Equation, the agent refines its Q-values and improves its decision-making process.

5. **Exploration Strategies:** To balance exploration (trying new actions) and exploitation (using known actions with high rewards), we use strategies like ε-greedy and Upper Confidence Bound (UCB) during the Q-Learning phase. These strategies ensure the agent explores enough to discover optimal actions while exploiting the best-known actions to maximize rewards.

Steps to Implement the code base:

Define the Environment:

```python
import numpy as np

class Gridworld:
    def __init__(self, size=5):
        self.size = size
        self.grid = np.zeros((size, size))
        self.start_state = (0, 0)
        self.goal_state = (size-1, size-1)
        self.state = self.start_state
        self.obstacles = [(1, 1), (2, 2), (3, 3)] # Add obstacles here
        for obstacle in self.obstacles:
            self.grid[obstacle] = -1

    def reset(self):
        self.state = self.start_state
        return self.state

    def step(self, action):
        x, y = self.state
        if action == 0 and x > 0:  # Up
            x -= 1
        elif action == 1 and x < self.size - 1:  # Down
            x += 1
        elif action == 2 and y > 0:  # Left
            y -= 1
        elif action == 3 and y < self.size - 1:  # Right
            y += 1

        self.state = (x, y)
        reward = 1 if self.state == self.goal_state else -0.01
        done = self.state == self.goal_state
        return self.state, reward, done
```

Implement the Q-Learning Algorithm:

```python
import random

class QLearningAgent:
    def __init__(self, env, lr=0.1, gamma=0.9, epsilon=1.0, epsilon_decay=0.995):
        self.env = env
        self.q_table = np.zeros((env.size, env.size, 4))  # Four actions: up, down, left, right
        self.lr = lr
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice([0, 1, 2, 3])
        else:
            x, y = state
            return np.argmax(self.q_table[x, y])

    def learn(self, state, action, reward, next_state):
        x, y = state
        next_x, next_y = next_state
        predict = self.q_table[x, y, action]
        target = reward + self.gamma * np.max(self.q_table[next_x, next_y])
        self.q_table[x, y, action] += self.lr * (target - predict)

    def train(self, episodes):
        for _ in range(episodes):
            state = self.env.reset()
            done = False
            while not done:
                action = self.choose_action(state)
                next_state, reward, done = self.env.step(action)
                self.learn(state, action, reward, next_state)
                state = next_state
            self.epsilon *= self.epsilon_decay
env = Gridworld()
agent = QLearningAgent(env)
agent.train(1000)

print("Training completed!")
```

Implement the Policy Gradient Algorithm:

```
import torch
import torch.nn as nn
import torch.optim as optim

class PolicyNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.softmax(self.fc3(x), dim=-1)
        return x
class PolicyGradientAgent:
    def __init__(self, env, lr=0.01):
        self.env = env
        self.policy_net = PolicyNetwork(2, 4)
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
        self.gamma = 0.99

    def choose_action(self, state):
        state = np.array(state)
```

```python
      state = torch.FloatTensor(state).unsqueeze(0)
      probs = self.policy_net(state)
        action = np.random.choice(4, p=probs.detach().numpy()[0])
        return action

    def train(self, episodes):
        for episode in range(episodes):
            state = self.env.reset()
            rewards = []
            actions = []
            states = []

            done = False
            while not done:
                states.append(state)
                action = self.choose_action(state)
                next_state, reward, done = self.env.step(action)
                rewards.append(reward)
                actions.append(action)
                state = next_state

            # Compute discounted rewards
            discounted_rewards = []
            cumulative_reward = 0
            for reward in reversed(rewards):
                cumulative_reward = reward + self.gamma * cumulative_reward
                discounted_rewards.insert(0, cumulative_reward)
            discounted_rewards = torch.FloatTensor(discounted_rewards)

            # Normalize rewards
            discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewards.std() + 1e-9)

            # Update policy network
            self.optimizer.zero_grad()
            for state, action, reward in zip(states, actions, discounted_rewards):
                state = np.array(state)
                state = torch.FloatTensor(state).unsqueeze(0)
                probs = self.policy_net(state)
                loss = -torch.log(probs[0, action]) * reward
                loss.backward()
            self.optimizer.step()
            print(f'Episode {episode + 1}/{episodes}, Loss: {loss.item()}')
env = Gridworld()
```

```
    pg_agent = PolicyGradientAgent(env)
    pg_agent.train(1000)

    print("Policy Gradient Training completed!")
```

Q-Learning agent function:

```
class QLearningAgent:
    def __init__(self, env, lr=0.1, gamma=0.9, epsilon=1.0,
epsilon_decay=0.995):
        self.env = env
        self.q_table = np.zeros((env.size, env.size, 4))  # Four actions: up,
down, left, right
        self.lr = lr
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice([0, 1, 2, 3])
        else:
            x, y = state
            return np.argmax(self.q_table[x, y])

    def learn(self, state, action, reward, next_state):
        x, y = state
        next_x, next_y = next_state
        predict = self.q_table[x, y, action]
        target = reward + self.gamma * np.max(self.q_table[next_x, next_y])
        self.q_table[x, y, action] += self.lr * (target - predict)

    def train(self, episodes):
        episode_rewards = []
        success_count = 0

        for episode in range(episodes):
            state = self.env.reset()
            total_reward = 0
            done = False

            while not done:
                action = self.choose_action(state)
                next_state, reward, done = self.env.step(action)
                self.learn(state, action, reward, next_state)
                total_reward += reward
                state = next_state
            self.epsilon *= self.epsilon_decay
            episode_rewards.append(total_reward)
            if done and reward > 0:
                success_count += 1

        avg_reward = np.mean(episode_rewards)
        success_rate = success_count / episodes
        print(f"Q-Learning - Episodes: {episodes}, Average Reward:
{avg_reward}, Success Rate: {success_rate * 100}%")
```

Policy gradient agent function

```
class PolicyGradientAgent:
    def __init__(self, env, lr=0.01):
        self.env = env
        self.policy_net = PolicyNetwork(2, 4)  # Input size should be 2 for
(x, y) state
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
        self.gamma = 0.99

    def choose_action(self, state):
        state = np.array(state)
        state = torch.FloatTensor(state).unsqueeze(0)
        probs = self.policy_net(state)
        action = np.random.choice(4, p=probs.detach().numpy()[0])
        return action

    def train(self, episodes):
        episode_rewards = []
        success_count = 0

        for episode in range(episodes):
            state = self.env.reset()
            rewards = []
            actions = []
            states = []

            done = False
            while not done:
                states.append(state)
                action = self.choose_action(state)
                next_state, reward, done = self.env.step(action)
                rewards.append(reward)
                actions.append(action)
                state = next_state

            # Compute discounted rewards
            discounted_rewards = []
            cumulative_reward = 0
            for reward in reversed(rewards):
                cumulative_reward = reward + self.gamma * cumulative_reward
                discounted_rewards.insert(0, cumulative_reward)
            discounted_rewards = torch.FloatTensor(discounted_rewards)

            # Normalize rewards
            discounted_rewards = (discounted_rewards -
discounted_rewards.mean()) / (discounted_rewards.std() + 1e-9)

            # Update policy network
            self.optimizer.zero_grad()
            for state, action, reward in zip(states, actions,
discounted_rewards):
                state = np.array(state)
                state = torch.FloatTensor(state).unsqueeze(0)
                probs = self.policy_net(state)
                loss = -torch.log(probs[0, action]) * reward
                loss.backward()
            self.optimizer.step()

            total_reward = sum(rewards)
            episode_rewards.append(total_reward)
            if done and reward > 0:
                success_count += 1

        avg_reward = np.mean(episode_rewards)
        success_rate = success_count / episodes
        print(f"Policy Gradient - Episodes: {episodes}, Average Reward:
{avg_reward}, Success Rate: {success_rate * 100}%")
```

Evaluate Performance:

```
# Q-Learning Evaluation
env = Gridworld()
ql_agent = QLearningAgent(env)
ql_agent.train(1000)

# Policy Gradient Evaluation
pg_agent = PolicyGradientAgent(env)
pg_agent.train(1000)
```

Visualize Results:

```
import matplotlib.pyplot as plt

def plot_q_table(q_table):
    fig, ax = plt.subplots(1, figsize=(5, 5))
    for x in range(q_table.shape[0]):
        for y in range(q_table.shape[1]):
            if (x, y) in env.obstacles:
                ax.text(y, x, 'X', ha='center', va='center', color='red',
fontsize=20)
            elif (x, y) == env.goal_state:
                ax.text(y, x, 'G', ha='center', va='center', color='green',
fontsize=20)
            else:
                actions = ['↑', '↓', '←', '→']
                action = np.argmax(q_table[x, y])
                ax.text(y, x, actions[action], ha='center', va='center',
color='black', fontsize=20)
    ax.set_xticks(np.arange(-0.5, q_table.shape[1], 1))
    ax.set_yticks(np.arange(-0.5, q_table.shape[0], 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(color='gray')
    plt.title("Learned Policy (Q-Learning)")
    plt.show()

# Visualize the Q-table
plot_q_table(ql_agent.q_table)
def plot_policy(policy_net, env):
    fig, ax = plt.subplots(1, figsize=(5, 5))
    for x in range(env.size):
        for y in range(env.size):
            if (x, y) in env.obstacles:
                ax.text(y, x, 'X', ha='center', va='center', color='red',
fontsize=20)
            elif (x, y) == env.goal_state:
                ax.text(y, x, 'G', ha='center', va='center', color='green',
fontsize=20)
            else:
                state = torch.FloatTensor([x, y]).unsqueeze(0)
                probs = policy_net(state).detach().numpy()[0]
                actions = ['↑', '↓', '←', '→']
                action = np.argmax(probs)
                ax.text(y, x, actions[action], ha='center', va='center',
color='black', fontsize=20)
    ax.set_xticks(np.arange(-0.5, env.size, 1))
    ax.set_yticks(np.arange(-0.5, env.size, 1))
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    ax.grid(color='gray')
    plt.title("Learned Policy (Policy Gradient)")
    plt.show()
```

```
# Visualize the policy
plot_policy(pg_agent.policy_net, env)
```

expected output:

result:

first trial

```
Q-Learning - Episodes: 1000, Average Reward: 0.88769, Success Rate: 100.0%
Policy Gradient - Episodes: 1000, Average Reward: 0.8014999999999857, Success Rate: 100.0%
```

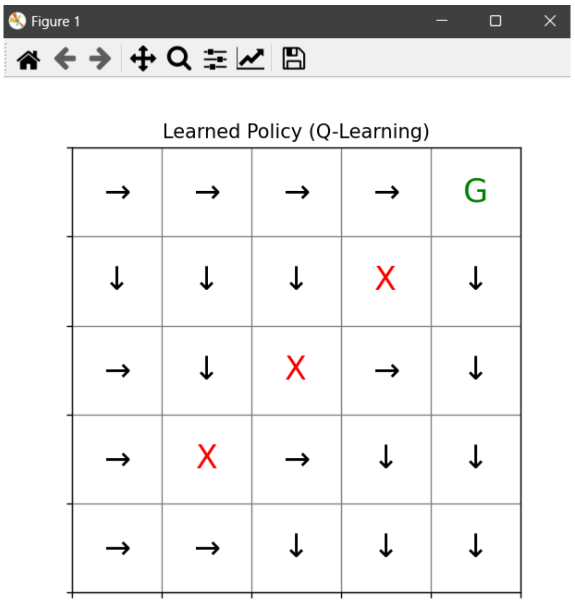
Second trial

```
Q-Learning - Episodes: 1000, Average Reward: 0.8878199999999999, Success Rate: 100.0%
Policy Gradient - Episodes: 1000, Average Reward: 0.9234099999999998, Success Rate: 100.0%
```

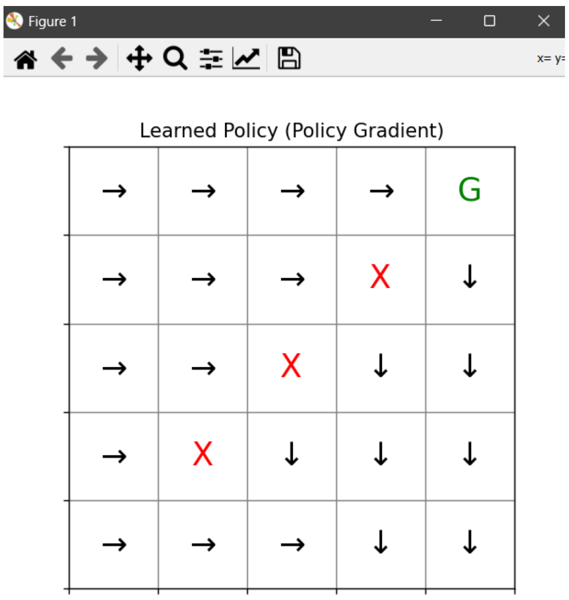
Third trial

```
Q-Learning - Episodes: 1000, Average Reward: 0.8865799999999999, Success Rate: 100.0%
Policy Gradient - Episodes: 1000, Average Reward: 0.9252599999999999, Success Rate: 100.0%
```

visualisation:



→	→	→	→	G
↓	↓	↓	X	↓
→	↓	X	→	↓
→	X	→	↓	↓
→	→	↓	↓	↓



Advantages

1. **Hands-On Learning:**
   - This project provides a practical, hands-on approach to understanding reinforcement learning (RL) concepts. By implementing and comparing different RL methods, you deepen your comprehension of how these algorithms work.
2. **Clear Visualization:**
   - The Gridworld environment is simple yet effective for visualizing the learning process of RL agents. The visualization of Q-values and policies makes it easier to grasp the agent's decision-making process and the convergence of learning algorithms.
3. **Comparison of Methods:**
   - Comparing Q-Learning and Policy Gradient methods offers insights into the strengths and weaknesses of each approach. This comparison helps in understanding when to use each method based on the problem characteristics.
4. **Structured Framework:**
   - Defining the Gridworld as a Markov Decision Process (MDP) provides a structured framework for the agent's learning. This helps in systematically analyzing and improving the agent's performance.
5. **Exploration Strategies:**
   - Implementing exploration strategies like ε-greedy and Upper Confidence Bound (UCB) introduces important RL concepts, highlighting the balance between exploration and exploitation in learning.
6. **Foundation for Advanced Techniques:**

- This project lays a solid foundation for more advanced RL techniques such as deep Q-learning and Actor-Critic methods. It provides the necessary groundwork to delve into more complex environments and algorithms.

Disadvantages

1. **Simplified Environment:**
   - The Gridworld environment is quite simple and may not fully capture the complexities of real-world scenarios. The performance and insights gained here might not directly translate to more complex environments.
2. **Limited Generalization:**
   - The specific implementations of Q-Learning and Policy Gradient methods may be limited to discrete state and action spaces. Extending these methods to continuous spaces or more complex tasks may require significant modifications.
3. **Computational Resources:**
   - Training RL agents, especially with neural networks, can be computationally intensive. Depending on the complexity of the environment and the algorithms, this can become a bottleneck, requiring substantial computational resources.
4. **Convergence Issues:**
   - RL algorithms can sometimes face convergence issues, especially in environments with sparse rewards or high-dimensional state spaces. This can make it challenging to achieve consistent results.
5. **Hyperparameter Sensitivity:**
   - The performance of RL algorithms can be highly sensitive to the choice of hyperparameters (e.g., learning rate, discount factor). Finding the optimal hyperparameters often involves trial and error, which can be time-consuming.
6. **Implementation Complexity:**
   - Implementing RL algorithms, especially Policy Gradient methods with neural networks, can be complex and prone to errors. Debugging and fine-tuning the implementation may require substantial effort and expertise.

Conclusion:

While there are some limitations and challenges, this project offers a comprehensive introduction to reinforcement learning, providing valuable insights and practical experience. The advantages far outweigh the disadvantages, making it a worthwhile endeavour in domain of reinforcement learning. We can extend it later by incorporating more advanced topics or complexities based on your learning pace and interest!