Decision Making Under Uncertainty

Project 2: Reinforcement Learning

Wei-Lin Pai, wpai@stanford.edu

1. Algorithm Description

In this project, I implement the simplest form of Q-learning to generate the optimal policy. Q-learning generates reasonably good policies for small, medium, and large datasets.

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma\, max_{\{a'\}}Q(s',a') - Q(s,a)\right)$$

The initial problem I countered while performing Q-Learning was that the policy generated for medium and large datasets performs way worse than the random policy in the grade scope. Hence, I started tuning the two hyperparameters in Q-Learning, iteration time and α. After some tries, I found out that the main reason for the bad result was because α was too large, which prevented the Q value from converging. Hence, I changee α from 2.0 to range between 0.1 and 0.25 and the optimal policy then performs way better.

I also tried altering the iteration time. Generally, when iterations increase, the policy performs better. Until 1000 iterations did the medium data policy generate policy way better than the previous ones. However, the policy of small and large datasets doesn't seem to improve much due to the increased iteration.

I then increased the iteration time to 1000; the medium policy performs way better.

1 iteration

Raw Policy Score (Training time)	alpha = 0.1	alpha = 0.2	alpha = 0.25
small.csv	34.1 (0.48s)	34.8	33.49
medium.csv	-0.47 (0.98 s)	-0.47	-0.47
Large.csv	252.22 (0.98 s)	260.54	267.70

10 iterations

Raw Policy Score (Training time)	alpha = 0.1	alpha = 0.15	alpha = 0.20
small.csv	35.11 (4.8 s)	34.9	33.65
medium.csv	-0.47 (9.6 s)	-0.48	-0.47
Large.csv	512 (9.45 s)	522	461

100 Iterations

Raw Policy Score (Training time)	alpha = 0.1
small.csv	35.11 (48.31 s)
medium.csv	-0.48 (99.86 s)
Large.csv	517.84 (98.90 s)

1000 iterations

Raw Policy Score (Training time)	alpha = 0.1
small.csv	35.11 (491.50 s)
medium.csv	78.06 (996.68 s)
Large.csv	517.84 (1005.63)

2. Reward Shaping

Since the reward for the mountain car is relatively sparse (reward only relies on whether the car reaches the goal), I try to add reward shaping to simulate the car with velocity reward and position reward:

$$r_{shaped} = r + r_{vel} + r_{pos}$$

Velocity Reward (r_vel):

(1) When the car is on the left side of the track, the reward is proportional to the absolute value of velocity. Since building up velocity in either way should be beneficial to reaching the goal.

(2) When the car is on the right side of the track, give the reward for moving right.

Position Reward (r_pos): Give a positive reward for moving rightward.

After applying reward shaping on medium dataset policy:

Iteration	After Reward Shaping (alpha = 0.1)	Before Reward Shaping (alpha = 0.1)
10	-0.029	-0.47
100	-0.027	-0.48
1000	78.15	78.06

We can find out that in some iteration, the general performance of the policy is way better. However, in 1000 iteration the performance is only slightly better. I think the reason is either the reward shaping is not intrinsically correct or the iteration number should be further increased.

Q_Learning.py

```
import numpy as np
import pandas as pd
import sys
import time

# Q-Learning class
class QLearning():
    def __init__(self, S: list[int], A: list[int], gamma: float, Q: np.ndarray, alpha:
float):
        self.A = A          # action space
        self.gamma = gamma  # discount factor
        self.Q = Q           # The action value function Q[s, a] is a numpy array
        self.alpha = alpha  # learning rate
        self.S = S  # state space
    # Return the action to be taken in state s
    def lookahead(self, s: int, a: int):
        return self.Q[s, a]
```

```
    # Update the action value function Q[s, a] based on the reward r and the next state
s_prime
    def update(self, s: int, a: int, r: float, s_prime: int):
        self.Q[s, a] += self.alpha * (r + (self.gamma * np.max(self.Q[s_prime])) -
self.Q[s, a])

    # Transform index to state
    def index_to_state(self, index: int) -> tuple:
        """Convert state index back to position and velocity"""
        vel = index // 500
        pos = index % 500
        return pos, vel

    def reward_shaped_update(self, s: int, a: int, r: float, s_prime: int):
        # Modified Q-learning update with reward shaping
        shaped_reward = self.shape_reward(s, r, s_prime)

        self.Q[s, a] += self.alpha * (shaped_reward + (self.gamma *
np.max(self.Q[s_prime])) - self.Q[s, a])

    def shape_reward(self, s: int, r: float, s_prime: int) -> float:
        """Shape the reward to provide better learning signals"""

        # Extract position and velocity from states
        pos, vel = self.index_to_state(s)
        next_pos, next_vel = self.index_to_state(s_prime)
        # Normalize position and velocity
        pos -= 250
        next_pos -= 250
        vel -= 50
        next_vel -= 50
        # Reward for moving right
        position_reward = (next_pos - pos) * 0.1

        # Reward for gaining velocity in useful directions
        velocity_reward = 0
        if pos < 0:  # Left side of track
            velocity_reward = abs(next_vel) * 0.1  # Reward building up velocity
        else:  # Right side of track
            velocity_reward = next_vel * 0.1 if next_vel > 0 else 0  # Reward moving right

        return r + position_reward + velocity_reward

def main():
    # Read the input file
    if sys.argv[1] == "small":
        A = [1,2,3,4]
```

```
        gamma = 0.95
        Q = np.zeros((100, 4))
        alpha = 0.2
        S = list(range(100))

    elif sys.argv[1] == "medium":
        A = [1,2,3,4,5,6,7]
        gamma = 1.0
        Q = np.zeros((50000, 7))
        alpha = 0.1
        S = list(range(50000))

    elif sys.argv[1] == "large":
        A = [1,2,3,4,5,6,7,8,9]
        gamma = 0.95
        Q = np.zeros((302020, 9))
        alpha = 0.15
        S = list(range(302020))

    else:
        print("Invalid input")
        return

    # Read the data file
    data = pd.read_csv("data/" + sys.argv[1] + ".csv", skiprows=1)
    q_learning = QLearning(S, A, gamma, Q, alpha)
    start_time = time.time()

    # Train the Q-Learning agent
    max_iter = 10
    for episdoe in range(max_iter):
        for index, row in data.iterrows():
            s, a, r, s_p = row
            # Update the action value function Q[s, a]
            # Convert the 1-based state and action to 0-based state and action
            if sys.argv[1] == "medium":
                q_learning.reward_shaped_update(s - 1, a - 1, r, s_p - 1)
            else:
                q_learning.update(s - 1, a - 1, r, s_p - 1)
    end_time = time.time()
    print("Training time with iter = ",max_iter ,", alpha = ",alpha , ":", end_time -
start_time)

    # Generate the optimal policy
    policy = np.argmax(q_learning.Q, axis=1)
     # Write the policy to a file
    with open(sys.argv[1]+ ".policy", "w") as f:
```

```
        for action in policy:
            # Convert 0-based action to 1-based action
            f.write(f"{action + 1}\n")

if __name__ == "__main__":
    main()
```