Project 2: Reinforcement Learning

Aaron Jin aaronjin@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Descriptions

1.1 Small Data Set

The small data set involves a 10x10 grid world problem (with 100 states and 4 actions). For this, I implemented a standard Q-learning algorithm with an adaptive learning rate. My algorithm uses a tabular representation of Q-values with a discount factor of 0.95 (as specified). Furthermore, the adaptive learning rate is computed based on state-action visit counts to guarantee proper convergence. I ran the training process for 100 epochs over the provided transition data, where I saw that the algorithm achieved stable convergence with a runtime of 87.18 seconds. Overall, I noticed that the simple nature of the state space allowed for effective exploration and exploitation balance without requiring any extensive, sophisticated exploration strategies. To highlight, some key features of our implementation include:

- Tabular Q-learning with zero initialization.
- Adaptive learning rate: $\alpha = \frac{\alpha_0}{\sqrt{N(s,a)}}$.
- Discount factor: $\gamma = 0.95$.
- 100 training epochs.

1.2 Medium Data Set

The medium data set involves the MountainCarContinuous-v0 environment. For this, I implemented a more specialized Q-learning approach because I had to address the non-Markovian nature of the discretized states. The state space consisted of 50,000 states (500 position $\times$ 100 velocity values) with 7 discrete actions representing different values of acceleration. I modified our algorithm to use position and velocity information in order to implement physics-aware heuristics; this came particularly useful in overcoming my initial challenges of discretization. With this algorithm, I ran the training process for 300 epochs with a runtime of 398.15 seconds. Here, I noticed that successful episodes were processed more frequently than unsuccessful ones in order to propagate valuable information. Our reward shaping incorporated the following: momentum building rewards in the valley; progress rewards based on position changes; velocity-based rewards for maintaining momentum; and strong positive rewards for reaching the goal. To highlight, some key features of our implementation include:

- Optimistic initialization (200.0) to encourage exploration.
- Physics-based reward shaping.

- Episode-based training with prioritization of successful trajectories.
- Adaptive learning rate with visit count decay.
- State-action memory for successful transitions.

1.3 Large Data Set

The large data set involves an even larger MDP with 302,020 states and 9 actions (with a discount factor of 0.95). For this, I used the same base Q-learning algorithm as the small dataset, but I included some modifications to handle the larger state space efficiently. My approach, thus, focused on being relatively efficient while also dealing with the hidden structure. Additionally, I ran the training process for 300 epochs with a runtime of 560.41 seconds. Most notably, I saw that my algorithm effectively handled the large state space without requiring explicit structure discovery, relying instead on the natural convergence properties of Q-learning with sufficient exploration. To highlight, some key features of our implementation include:

- Standard Q-learning with adaptive learning rates.
- Random shuffling of transitions each epoch.
- 300 training epochs.
- Conservative learning rate (0.1) for stability.

2. Performance Characteristics

Our total execution time for all three datasets was 1045.77 seconds. The breakdown of this includes:

- Small dataset (100 states): 87.18 seconds.
- Medium dataset (50,000 states): 398.15 seconds.
- Large dataset (302,020 states): 560.41 seconds.

3. Code

Note: The implementation has been chunked into logical sections to accommodate LaTeX memory constraints.

3.1 Base Q-Learning Implementation

```
import numpy as np
import pandas as pd
from pathlib import Path
import os
from time import time
from datetime import datetime
from collections import defaultdict
import warnings
warnings.filterwarnings('ignore')

class BatchQLearning:
    def __init__(self, n_states, n_actions, gamma, learning_rate=0.1):
        self.n_states = n_states
        self.n_actions = n_actions
        self.gamma = gamma
        self.learning_rate = learning_rate
        self.Q = np.zeros((n_states, n_actions))
        self.visit_count = np.zeros((n_states, n_actions))

    def update(self, state, action, reward, next_state):
        state = int(state - 1)
        next_state = int(next_state - 1)
        action = int(action - 1)

        if (state < 0 or state >= self.n_states or
            next_state < 0 or next_state >= self.n_states or
            action < 0 or action >= self.n_actions):
            return

        self.visit_count[state, action] += 1
        effective_lr = self.learning_rate / np.sqrt(self.visit_count[state,
    action])

        best_next_value = np.max(self.Q[next_state])
        self.Q[state, action] = (1 - effective_lr) * self.Q[state, action] +
    \
                                effective_lr * (reward + self.gamma *
    best_next_value)

    def get_policy(self):
        return np.argmax(self.Q, axis=1) + 1
```

3.2 Mountain Car Q-Learning Implementation

```
class MountainCarQLearning:
    def __init__(self, n_states, n_actions, learning_rate=0.15):
        self.n_states = n_states
        self.n_actions = n_actions
        self.learning_rate = learning_rate
        self.Q = np.ones((n_states, n_actions)) * 200.0
        self.state_visits = np.zeros(n_states)
        self.action_counts = np.zeros((n_states, n_actions))
        self.success_memory = defaultdict(list)
        self.n_pos = 500
        self.n_vel = 100
    def state_to_pos_vel(self, state):
        state = state - 1
        vel_idx = state // 500
        pos_idx = state % 500
        return pos_idx, vel_idx

    def continuous_values(self, pos_idx, vel_idx):
        pos = -1.2 + (1.8 * pos_idx / 499)
        vel = -0.07 + (0.14 * vel_idx / 99)
        return pos, vel

    def get_heuristic_value(self, pos, vel):
        if pos < -0.5:
            value = 50 * (vel * vel)
        elif pos < 0:
            value = 100 * (pos + 0.5) + 75 * vel
        else:
            value = 150 * pos + 100 * (vel if vel > 0 else 0)

        if pos > 0.4:
            value += 200

        return value

    def get_heuristic_action(self, pos, vel):
        if pos < -0.5:
            return 1 if vel <= 0 else 7
        elif pos < 0:
            if vel > 0:
                return 7
            else:
                return 1
        else:
            return 7 if vel >= 0 else 4
```

3.3 Mountain Car Update and Policy Methods

```
  def update(self, state, action, reward, next_state):
      state_idx = int(state - 1)
      next_state_idx = int(next_state - 1)
      action_idx = int(action - 1)

      self.state_visits[state_idx] += 1
      self.action_counts[state_idx, action_idx] += 1

      pos_idx, vel_idx = self.state_to_pos_vel(state)
      next_pos_idx, next_vel_idx = self.state_to_pos_vel(next_state)
      pos, vel = self.continuous_values(pos_idx, vel_idx)
```

```
        next_pos, next_vel = self.continuous_values(next_pos_idx, next_vel_idx)

        shaped_reward = reward
        if reward <= 0:
            pos_progress = (next_pos - pos)
            vel_progress = abs(next_vel) - abs(vel)

            if pos < -0.5:
                shaped_reward += 50 * vel_progress
            elif pos < 0:
                shaped_reward += 75 * pos_progress + 25 * vel_progress
            else:
                shaped_reward += 100 * pos_progress + (50 * next_vel if next_vel
        > 0 else -25)

            if abs(pos_progress) < 0.01:
                shaped_reward -= 10

        else:
            shaped_reward += 1000
            self.success_memory[state_idx].append((action_idx, reward))

        visits = self.action_counts[state_idx, action_idx]
        effective_lr = self.learning_rate / (1 + 0.1 * np.sqrt(visits))

        if len(self.success_memory[next_state_idx]) > 0:
            next_value = max([r for _, r in self.success_memory[next_state_idx]])
        else:
            next_pos, next_vel = self.continuous_values(next_pos_idx,
        next_vel_idx)
            heuristic_bonus = 0.1 * self.get_heuristic_value(next_pos, next_vel)
            next_value = np.max(self.Q[next_state_idx]) + heuristic_bonus

        self.Q[state_idx, action_idx] = (1 - effective_lr) * self.Q[state_idx,
        action_idx] + \
                                       effective_lr * (shaped_reward + next_value
        )

    def get_policy(self):
        policy = np.zeros(self.n_states, dtype=int)

        for state in range(self.n_states):
            pos_idx, vel_idx = self.state_to_pos_vel(state + 1)
            pos, vel = self.continuous_values(pos_idx, vel_idx)

            if self.state_visits[state] < 5:
                policy[state] = self.get_heuristic_action(pos, vel)
            else:
                q_values = self.Q[state].copy()
            if pos < -0.5:
                if vel >= 0:
                    q_values[6] += 50
                else:
                    q_values[0] += 50
            elif pos < 0:
                if vel > 0:
                    q_values[5:] += 75
                else:
                    q_values[:2] += 75
            else:
                q_values[5:] += 100

            policy[state] = np.argmax(q_values) + 1

    return policy
```

3.4 Processing Functions

```
def process_small(data_path, output_path):
    print(f"\nProcessing small dataset...")
    start_time = time()

    df = pd.read_csv(data_path)
    n_states = 100
    n_actions = 4
    gamma = 0.95
    agent = BatchQLearning(n_states, n_actions, gamma)

    n_epochs = 100
    for epoch in range(n_epochs):
        for _, row in df.iterrows():
            agent.update(row['s'], row['a'], row['r'], row['sp'])

    policy = agent.get_policy()
    np.savetxt(output_path, policy, fmt='%d')
    print(f"Time elapsed: {time() - start_time:.2f}s")

def process_medium(data_path, output_path):
    print(f"\nProcessing medium dataset...")
    start_time = time()

    df = pd.read_csv(data_path)
    n_states = 50000
    n_actions = 7
    agent = MountainCarQLearning(n_states, n_actions, learning_rate=0.15)

    episodes = []
    current_episode = []
```

```
        last_state = None

        df = df.sort_values(['s', 'sp']).reset_index(drop=True)

        for _, row in df.iterrows():
            if last_state is not None and abs(row['s'] - last_state) > 1000:
                if current_episode:
                    episodes.append(current_episode)
                    current_episode = []
            current_episode.append(row)
            last_state = row['s']
            if row['r'] > 0:
                episodes.append(current_episode)
                current_episode = []
                last_state = None

        if current_episode:
            episodes.append(current_episode)

        successful_episodes = [ep for ep in episodes if any(t['r'] > 0 for t in
        ep)]
        other_episodes = [ep for ep in episodes if not any(t['r'] > 0 for t in ep
        )]

        n_epochs = 300
        for epoch in range(n_epochs):
            if successful_episodes:
                for episode in successful_episodes:
                    for transition in episode:
                        agent.update(transition['s'], transition['a'],
                                     transition['r'], transition['sp'])

            if epoch % 2 == 0:
                for episode in other_episodes:
                    for transition in episode:
                        agent.update(transition['s'], transition['a'],
                                     transition['r'], transition['sp'])

        policy = agent.get_policy()
        np.savetxt(output_path, policy, fmt='%d')
        print(f"Time elapsed: {time() - start_time:.2f}s")

    def process_large(data_path, output_path):
        print(f"\nProcessing large dataset...")
        start_time = time()

        df = pd.read_csv(data_path)
        n_states = 302020
        n_actions = 9
        gamma = 0.95
    agent = BatchQLearning(n_states, n_actions, gamma, learning_rate=0.1)

    n_epochs = 300
    for epoch in range(n_epochs):
        df = df.sample(frac=1, random_state=epoch).reset_index(drop=True)
        for _, row in df.iterrows():
            agent.update(row['s'], row['a'], row['r'], row['sp'])

    policy = agent.get_policy()
    np.savetxt(output_path, policy, fmt='%d')
    print(f"Time elapsed: {time() - start_time:.2f}s")

def main():
    base_dir = Path('.')
    data_dir = base_dir / 'data'
    output_dir = base_dir / 'output'
    output_dir.mkdir(exist_ok=True)

    total_start_time = time()
    datasets = ['small', 'medium', 'large']

    for dataset in datasets:
        data_path = data_dir / f"{dataset}.csv"
        output_path = output_dir / f"{dataset}.policy"

        if not data_path.exists():
            print(f"Warning: {data_path} does not exist!")
            continue

        try:
            if dataset == 'small':
                process_small(data_path, output_path)
            elif dataset == 'medium':
                process_medium(data_path, output_path)
            else:
                process_large(data_path, output_path)
        except Exception as e:
            print(f"Error processing {dataset}: {str(e)}")

    print(f"\nTotal execution time: {time() - total_start_time:.2f}s")

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