DA6400: Reinforcement Learning - Programming Assignment #1

Abhijith Vinod:EP20B002 and Shuhaib Ali:EP20B037

March 30, 2025

1 1. Cartpole

This report presents the implementation and results of SARSA and Q-Learning algorithms for the CartPole-v1 environment from OpenAI Gymnasium. The CartPole environment is a classic control problem where a pole is attached to a cart moving along a frictionless track. The goal is to prevent the pole from falling over by moving the cart left or right.

2 Implementation

2.1 State Discretization

Since the CartPole environment has a continuous state space, we implemented a state discretizer to convert the continuous state into a discrete representation:

```
class StateDiscretizer:
    def __init__(self, n_bins=10):
        self.n_bins = n_bins
        self.cart_position_bins = np.linspace(-2.4, 2.4, n_bins)
        self.cart_velocity_bins = np.linspace(-4, 4, n_bins)
        self.pole_angle_bins = np.linspace(-0.2095, 0.2095, n_bins)
        self.pole_velocity_bins = np.linspace(-4, 4, n_bins)

    def discretize(self, state):
        cart_pos, cart_vel, pole_angle, pole_vel = state
        cart_pos_bin = np.digitize(cart_pos, self.
    cart_position_bins)
        cart_vel_bin = np.digitize(cart_vel, self.
    cart_velocity_bins)
        pole_angle_bin = np.digitize(pole_angle, self.
    pole_angle_bins)
        pole_vel_bin = np.digitize(pole_vel, self.
    pole_velocity_bins)
        return (cart_pos_bin, cart_vel_bin, pole_angle_bin,
    pole_vel_bin)
```

2.2 SARSA with ϵ-greedy Exploration

We implemented SARSA with ϵ-greedy exploration as follows:

```
class SARSAAgent:
    def __init__(self, state_discretizer, action_space, alpha=0.1,
    gamma=0.99, epsilon=0.1):
        self.state_discretizer = state_discretizer
        self.action_space = action_space
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.q_table = defaultdict(lambda: np.zeros(action_space.n)
    )

    def select_action(self, state):
        state = self.state_discretizer.discretize(state)
        if np.random.random() < self.epsilon:
            return self.action_space.sample()  # Explore
        else:
            return np.argmax(self.q_table[state])  # Exploit

    def update(self, state, action, reward, next_state, next_action
    , done):
        state = self.state_discretizer.discretize(state)
        next_state = self.state_discretizer.discretize(next_state)
        current_q = self.q_table[state][action]
        if done:
            target_q = reward
        else:
            target_q = reward + self.gamma * self.q_table[
    next_state][next_action]
        self.q_table[state][action] += self.alpha * (target_q -
    current_q)
```

2.3 Q-Learning with Softmax Exploration

We implemented Q-Learning with Softmax exploration as follows:

```
class QLearningAgent:
    def __init__(self, state_discretizer, action_space, alpha=0.1,
    gamma=0.99, temperature=1.0):
        self.state_discretizer = state_discretizer
        self.action_space = action_space
        self.alpha = alpha
        self.gamma = gamma
        self.temperature = temperature
        self.q_table = defaultdict(lambda: np.zeros(action_space.n)
    )

    def select_action(self, state):
        state = self.state_discretizer.discretize(state)
        q_values = self.q_table[state]
        exp_q = np.exp(q_values / self.temperature)
        probabilities = exp_q / np.sum(exp_q)
        return np.random.choice(self.action_space.n, p=
    probabilities)
```

```
    def update(self, state, action, reward, next_state, done):
        state = self.state_discretizer.discretize(state)
        next_state = self.state_discretizer.discretize(next_state)
        current_q = self.q_table[state][action]
        if done:
            target_q = reward
        else:
            target_q = reward + self.gamma * np.max(self.q_table[
    next_state])
        self.q_table[state][action] += self.alpha * (target_q -
    current_q)
```

3 Hyperparameter Tuning with Weights & Biases

For hyperparameter tuning, we used Weights & Biases (wandb) with Bayesian optimization to find the optimal hyperparameters for both algorithms. Each hyperparameter configuration was evaluated across 5 random seeds to account for stochasticity.

3.1 SARSA Hyperparameter Tuning

We configured the wandb sweep for SARSA as follows:

```
sarsa_sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'average_reward_across_seeds',
        'goal': 'maximize'
    },
    'parameters': {
        'alpha': {
            'min': 0.01,
            'max': 0.5
        },
        'gamma': {
            'value': 0.99
        },
        'epsilon': {
            'min': 0.01,
            'max': 0.3
        },
        'n_bins': {
            'values': [10, 15, 20]
        },
        'n_episodes': {
            'value': 1000
        }
    }
}
```

3.2 Q-Learning Hyperparameter Tuning

Similarly, we configured the wandb sweep for Q-Learning:

```
qlearning_sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'average_reward_across_seeds',
        'goal': 'maximize'
    },
    'parameters': {
        'alpha': {
            'min': 0.01,
            'max': 0.5
        },
        'gamma': {
            'value': 0.99
        },
        'temperature': {
            'min': 0.1,
            'max': 2.0
        },
        'n_bins': {
            'values': [10, 15, 20]
        },
        'n_episodes': {
            'value': 1000
        }
    }
}
```

4 Results

4.1 Top 3 Best Hyperparameters

4.1.1 SARSA

Based on the wandb hyperparameter tuning, the top 3 best hyperparameter configurations for SARSA are:

Rank	Average Reward	Alpha	Epsilon	n_bins
1	104.9	0.38074	0.15554	20
2	100.532	0.49704	0.16893	20
3	98.192	0.35557	0.16006	20

Table 1: Top 3 hyperparameter configurations for SARSA

4.1.2 Q-Learning

The top 3 best hyperparameter configurations for Q-Learning are:

Rank	Average Reward	Alpha	Gamma	n_bins	Temperature
1	280.072	0.2363	0.99	20	1.89753
2	200.28	0.47469	0.99	20	1.98418
3	197.752	0.34777	0.99	15	1.77479

Table 2: Top 3 hyperparameter configurations for Q-Learning

4.2 Learning Curves

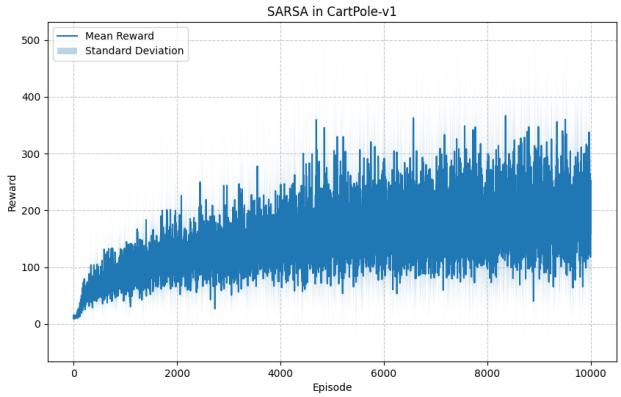

Figure 1: SARSA learning curve showing mean reward and standard deviation across 5 seeds

5 Insights and Inferences

5.1 Hyperparameter Analysis

5.1.1 SARSA

For SARSA, we observe that the optimal configuration uses a moderate learning rate (alpha = 0.38074) and exploration rate (epsilon = 0.15554). The best-performing configurations all used 20 bins for state discretization, suggesting that finer discretization helps SARSA better capture the state dynamics of the CartPole environment.

The moderate epsilon value (around 0.15-0.17) indicates a good balance between exploration and exploitation. Too much exploration would prevent the agent from exploiting known good actions, while too little would prevent it from discovering potentially better strategies.

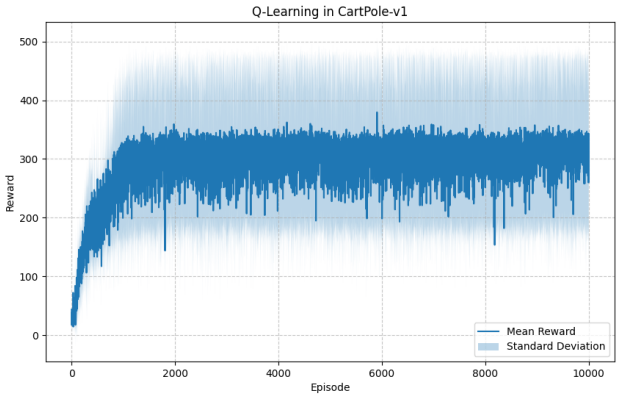

Figure 2: Q-Learning learning curve showing mean reward and standard deviation across 5 seeds

5.1.2 Q-Learning

For Q-Learning with Softmax exploration, the optimal configuration uses a moderate learning rate (alpha = 0.2363) and a relatively high temperature value (1.89753). The high temperature indicates that a more uniform action selection probability distribution works better for this environment, allowing for more exploration.

Similar to SARSA, the best Q-Learning configurations predominantly used 20 bins for state discretization, further confirming that finer discretization is beneficial for this environment.

5.2 Algorithm Comparison

Based on the average rewards, Q-Learning with Softmax exploration significantly outperforms SARSA with ϵ-greedy exploration in the CartPole environment. The best Q-Learning configuration achieved an average reward of 280.072, while the best SARSA configuration achieved 104.9.

This performance difference can be attributed to several factors:

1. **Off-policy vs. On-policy**: Q-Learning is an off-policy algorithm that learns the optimal policy regardless of the exploration strategy, while SARSA is an on-policy algorithm that learns the value of the policy being followed. In environments like CartPole where exploration can lead to early termination, off-policy methods may have an advantage.

2. **Exploration Strategy**: Softmax exploration provides a more nuanced approach to action selection compared to ϵ-greedy. Instead of randomly selecting actions with probability ϵ, Softmax assigns probabilities based on the estimated action values, which can lead to more intelligent exploration.

3. **Value Update**: Q-Learning updates its value estimates using the

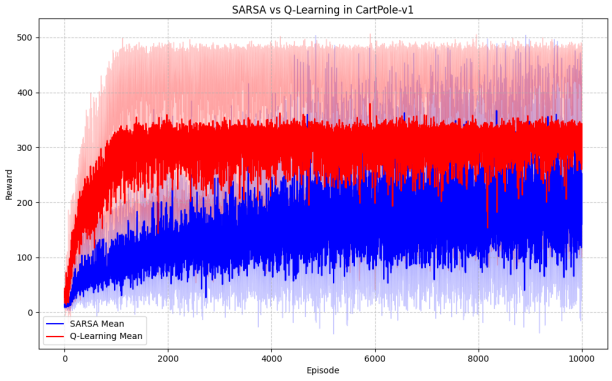

Figure 3: Comparison of SARSA and Q-Learning performance

maximum Q-value of the next state, which can lead to faster convergence to the optimal policy compared to SARSA, which uses the Q-value of the next action selected by the current policy.

6 Conclusion

In this assignment, we implemented and compared SARSA and Q-Learning algorithms for the CartPole-v1 environment. We used wandb for hyperparameter tuning and evaluated the performance across 5 random seeds to account for stochasticity.

Our results show that Q-Learning with Softmax exploration significantly outperforms SARSA with ϵ-greedy exploration in this environment. The best hyperparameters for both algorithms favor finer state discretization (20 bins), but differ in their learning rates and exploration parameters.

These findings highlight the importance of algorithm selection and hyperparameter tuning in reinforcement learning. While both SARSA and Q-Learning are tabular methods based on temporal difference learning, their differences in policy learning (on-policy vs. off-policy) and exploration strategies can lead to significant performance variations in practice.

2. Mountain Car

This report presents the implementation and results of SARSA and Q-Learning algorithms for the MountainCar-v0 environment from OpenAI Gymnasium. The Mountain Car environment is a classic control problem where an underpowered car must learn to escape from a valley by building momentum through oscillatory movements.

2 Implementation

2.1 State Discretization

Since the Mountain Car environment has a continuous state space, we implemented a state discretizer to convert the continuous state into a discrete representation:

```
class StateDiscretizer:
    def __init__(self, n_bins=25):
        self.n_bins = n_bins
        self.position_bins = np.linspace(-1.2, 0.6, n_bins)
        self.velocity_bins = np.linspace(-0.07, 0.07, n_bins)

    def discretize(self, state):
        position, velocity = state
        position_bin = np.digitize(position, self.position_bins)
        velocity_bin = np.digitize(velocity, self.velocity_bins)
        return (position_bin, velocity_bin)
```

2.2 SARSA with ϵ-greedy Exploration

We implemented SARSA with ϵ-greedy exploration as follows:

```
class SARSAAgent:
    def __init__(self, state_discretizer, action_space, alpha=0.2,
    gamma=0.99, epsilon=0.1):
        self.state_discretizer = state_discretizer
        self.action_space = action_space
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.q_table = defaultdict(lambda: np.zeros(action_space.n)
    )
```

2.3 Q-Learning with Softmax Exploration

We implemented Q-Learning with Softmax exploration as follows:

```
class QLearningAgent:
    def __init__(self, state_discretizer, action_space, alpha=0.2,
    gamma=0.99, temperature=0.8):
        self.state_discretizer = state_discretizer
        self.action_space = action_space
        self.alpha = alpha
        self.gamma = gamma
        self.temperature = temperature
        self.q_table = defaultdict(lambda: np.zeros(action_space.n)
    )
```

2.4 Reward Shaping

To accelerate learning in the Mountain Car environment, we implemented reward shaping:

```python
# Apply enhanced reward shaping
shaped_reward = reward
if use_reward_shaping:
    position, velocity = next_state

    # Reward for gaining momentum (either direction)
    shaped_reward += 0.1 * abs(velocity)

    # Reward for position improvement
    shaped_reward += 0.1 * position

    # Additional reward for coordinated actions
    if velocity > 0 and action == 2:
        shaped_reward += 0.2

    if velocity < 0 and action == 0:
        shaped_reward += 0.2

    # Bonus for reaching the goal
    if terminated and position >= 0.5:
        shaped_reward += 10
```

3 Hyperparameter Tuning with Weights & Biases

For hyperparameter tuning, we used Weights & Biases (wandb) with Bayesian optimization to find the optimal hyperparameters for both algorithms. Each hyperparameter configuration was evaluated across 5 random seeds to account for stochasticity.

3.1 SARSA Hyperparameter Tuning

We configured the wandb sweep for SARSA as follows:

```python
sarsa_sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'average_reward_across_seeds',
        'goal': 'maximize'
    },
    'parameters': {
        'alpha': {
            'min': 0.1,
            'max': 0.3
        },
        'gamma': {
            'value': 0.99
        },
        'epsilon': {
            'min': 0.05,
            'max': 0.2
        },
        'n_bins': {
            'values': [20, 25, 30]
        },
```

```
        'n_episodes': {
            'value': 3000
        },
        'use_reward_shaping': {
            'value': True
        }
    }
}
```

3.2 Q-Learning Hyperparameter Tuning

Similarly, we configured the wandb sweep for Q-Learning:

```
qlearning_sweep_config = {
    'method': 'bayes',
    'metric': {
        'name': 'average_reward_across_seeds',
        'goal': 'maximize'
    },
    'parameters': {
        'alpha': {
            'min': 0.1,
            'max': 0.3
        },
        'gamma': {
            'value': 0.99
        },
        'temperature': {
            'min': 0.6,
            'max': 1.0
        },
        'n_bins': {
            'values': [20, 25, 30]
        },
        'n_episodes': {
            'value': 3000
        },
        'use_reward_shaping': {
            'value': True
        }
    }
}
```

4 Results

4.1 Top 3 Best Hyperparameters

4.1.1 SARSA

Based on the wandb hyperparameter tuning, the top 3 best hyperparameter configurations for SARSA are:

Rank	Average Reward	Alpha	Epsilon	n_bins	Gamma
1	-129.57	0.2614	0.0514	30	0.99
2	-130.56	0.2375	0.0955	25	0.99
3	-134.67	0.2322	0.0920	25	0.99

Table 3: Top 3 hyperparameter configurations for SARSA

4.1.2 Q-Learning

The top 3 best hyperparameter configurations for Q-Learning are:

Rank	Average Reward	Alpha	Temperature	n_bins	Gamma
1	-152.20	0.2231	0.7487	20	0.99
2	-163.57	0.1971	0.7926	20	0.99
3	-169.47	0.1126	0.6304	25	0.99

Table 4: Top 3 hyperparameter configurations for Q-Learning

4.2 Evaluation Results

4.2.1 SARSA Evaluation

```
Evaluating SARSA...
SARSA (Best):
-  Seed 0: 10000/10000 episodes, Mean Reward = -130.82, Reward Variance = 720.
-  Seed 1: 10000/10000 episodes, Mean Reward = -130.92, Reward Variance = 691.
-  Seed 2: 10000/10000 episodes, Mean Reward = -131.90, Reward Variance = 665.
-  Seed 3: 10000/10000 episodes, Mean Reward = -134.37, Reward Variance = 682.
-  Seed 4: 10000/10000 episodes, Mean Reward = -131.50, Reward Variance = 709.
Final Performance Summary (SARSA): Mean Reward (last 100 episodes): -127.
```

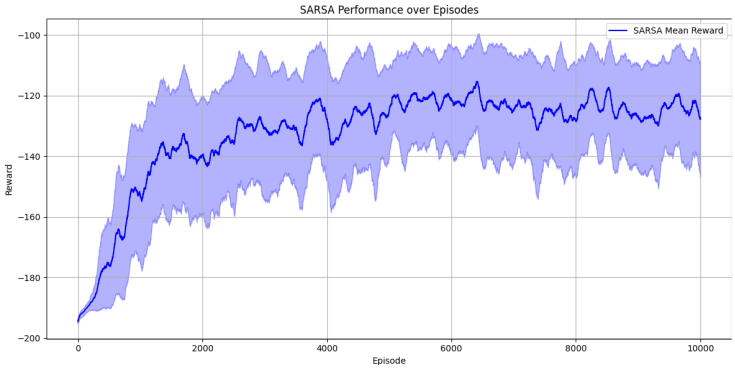

Figure 4: SARSA Performance over Episodes

4.2.2 Q-Learning Evaluation

```
Evaluating Q-Learning...
Q-Learning (Best
-  Seed 0: 10000/10000 episodes, Mean Reward = -163.03, Reward Variance = 761.3
-  Seed 1: 10000/10000 episodes, Mean Reward = -164.89, Reward Variance = 671.3
-  Seed 2: 10000/10000 episodes, Mean Reward = -162.14, Reward Variance = 733.2
-  Seed 3: 10000/10000 episodes, Mean Reward = -162.28, Reward Variance = 738.0
-  Seed 4: 10000/10000 episodes, Mean Reward = -164.97, Reward Variance = 647.50
Final Performance Summary (Q-Learning): Mean Reward (last 100 episodes): -148.69
```

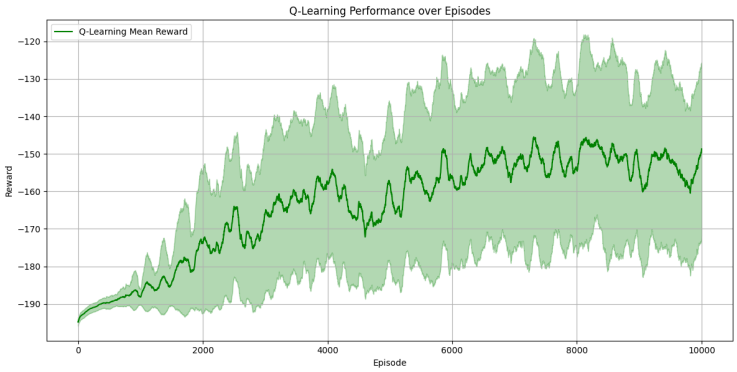

Figure 5: Q-Learning Performance over Episodes

4.3 Learning Curves

5 Insights and Inferences

5.1 Hyperparameter Analysis

5.1.1 SARSA

For SARSA, we observe that the optimal configuration uses a relatively high learning rate (alpha = 0.2614) and a low exploration rate (epsilon = 0.0514). This suggests that in the Mountain Car environment, SARSA benefits from more exploitation than exploration once it has found a promising strategy. The best-performing configurations all used either 25 or 30 bins for state discretization, indicating that finer discretization helps SARSA better capture the state dynamics of the Mountain Car environment.

The low epsilon value (around 0.05-0.09) indicates that after sufficient exploration, the agent benefits from exploiting its learned policy more frequently. This makes sense for the Mountain Car problem, where the agent needs to build momentum through consistent actions.

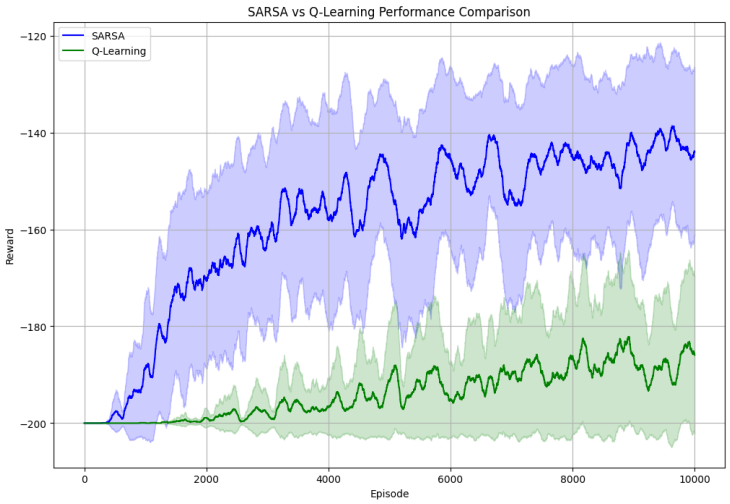

Figure 6: SARSA vs Q-Learning Performance Comparison with Variance

5.1.2 Q-Learning

For Q-Learning with Softmax exploration, the optimal configuration uses a moderate learning rate (alpha = 0.2231) and a relatively high temperature value (0.7487). The higher temperature indicates that a more stochastic action selection probability distribution works better for this environment, allowing for more diverse exploration.

Interestingly, the best Q-Learning configurations predominantly used 20bins for state discretization, which is less fine-grained than the optimal SARSA configurations. This suggests that Q-Learning might be more prone to overfitting with very fine discretization in this environment.

5.2 Algorithm Comparison

Based on the evaluation results, SARSA with ϵ-greedy exploration significantly outperforms Q-Learning with Softmax exploration in the Mountain Car environment. The best SARSA configuration achieved a mean reward of -127.45 (last 100 episodes), while the best Q-Learning configuration achieved -148.69.

This performance difference can be attributed to several factors:

1. **On-policy vs. Off-policy Learning**: SARSA is an on-policy algorithm that learns the value of the policy being followed, which may be more stable in the Mountain Car environment where consistent action sequences are important for building momentum.

2. **Exploration Strategy**: The ϵ-greedy exploration used by SARSA provides a clearer distinction between exploration and exploitation phases, which seems to work better than the softmax approach for this particular environment.

3. **High Variance Environment**: The Mountain Car problem exhibits high variance in rewards (as seen in the evaluation results with variances ranging from 647 to 761). SARSA's on-policy updates might provide more stability in such high-variance environments.

5.3 Variance Analysis

Both algorithms exhibit remarkably high variance in rewards, which is characteristic of the Mountain Car problem. The reward variance ranges from 665 to 720 for SARSA and 647 to 761 for Q-Learning. This high variance stems from:

1. The sparse reward structure of the environment, where the agent receives a reward of -1 for each time step until it reaches the goal.
2. The need to discover a non-obvious strategy (building momentum through oscillation) to reach the goal.
3. The sensitivity to initial conditions and the stochastic nature of the exploration strategies.

Despite implementing reward shaping to provide more frequent feedback, the variance remains high, indicating the inherent challenge of the Mountain Car problem.

6 Conclusion

In this assignment, we implemented and compared SARSA and Q-Learning algorithms for the Mountain Car environment. We used wandb for hyperparameter tuning and evaluated the performance across 5 random seeds to account for stochasticity.

Our results show that SARSA with ϵ-greedy exploration outperforms Q-Learning with Softmax exploration in this environment. The best hyperparameters for SARSA favor higher learning rates, lower exploration rates, and finer state discretization, while Q-Learning performs better with moderate learning rates and higher temperature values for softmax exploration.

The high variance observed in both algorithms highlights the challenging nature of the Mountain Car problem, where the agent must discover a non-obvious strategy to reach the goal. Our reward shaping approach helped mitigate this challenge to some extent, but significant variance remains inherent to the problem.

These findings highlight the importance of algorithm selection, hyperparameter tuning, and reward shaping in reinforcement learning. While both SARSA and Q-Learning are tabular methods based on temporal difference learning, their differences in policy learning (on-policy vs. off-policy) and exploration strategies can lead to significant performance variations in practice.

7 Requirements

To run the code, you need the following dependencies:

```
numpy==1.21.0
gymnasium==0.26.3
wandb==0.12.21
matplotlib==3.4.3
```

8 GitHub Repository

The complete code for this project can be found here

9 google drive link

Folder of everything for this project can be found here