SIT796 Reinforcement Learning

Distinction Task 7.1D: Function approximation implementation

Semi-Gradient Sarsa(0) is an algorithm that combines the Sarsa(0) algorithm with function approximation using linear function approximation. It is an on-policy control algorithm that learns the action-value function and determines the policy by iteratively updating the weights of the linear function approximator.

Semi-Gradient TD(λ) is an algorithm that combines temporal difference learning with eligibility traces and function approximation using linear function approximation. It is a control algorithm that learns the action-value function and determines the policy by iteratively updating the weights of the linear function approximator.

Similarities:

- Both algorithms use linear function approximation to estimate the action-value function. This enables them to learn and perform well in high-dimensional state spaces.

In the code for both Semi-Gradient Sarsa(0) and Semi-Gradient TD(λ), we can see the definition of the weight vector for linear function approximation:

```
# Define the weight vector for linear function approximation
weights_sarsa = np.random.rand(observation_space + 1)  # Add 1 for the action

# Define the weight vector for linear function approximation
weights_td = np.random.rand(observation_space + 1)  # Add 1 for the action
```

- They both use an epsilon-greedy policy to select actions during training and testing. This ensures a balance between trying out different actions and selecting the currently estimated optimal action.

In both algorithms, the epsilon-greedy policy function epsilon_greedy() is defined and used to select actions:

```
# Define the epsilon-greedy policy function
def epsilon_greedy(weights, state, epsilon):
    if np.random.rand() < epsilon:
        return np.random.randint(action_space)
    else:
        q_values = [np.dot(weights, get_features(state, a)) for a in range(action_space)]
        return np.argmax(q_values)
```

- They both update the weight vector of the linear function approximator using the gradient descent update rule. By doing this, both algorithms aim to minimize the TD error and improve the accuracy of the action-value function approximation. The learning rate (alpha) determines the step size of the updates.

In both algorithms, the weight vector is updated using the gradient descent update rule:

For Semi-Gradient Sarsa(0):

```
# Update the weight vector for TD(λ) using the gradient descent update rule with eligibility t
weights_td += alpha * td_error_td * eligibility_trace_td
```

For Semi-Gradient TD(λ):

```
# Update the weight vector for TD(λ) using the gradient descent update rule with eligibility
weights_td += alpha * td_error_td * eligibility_trace_td
```

Differences:

- Semi-Gradient Sarsa(0) updates the weight vector based on the TD error of the current state-action pair, while Semi-Gradient TD(λ) updates the weight vector based on the TD error of the current state and the eligibility trace. This difference impacts how the algorithms assign credit and update the function approximation.

For Semi-Gradient Sarsa(0):

```
# Compute the TD error for Sarsa(0)
td_error = reward + gamma * np.dot(weights_sarsa, get_features(next_state, next_action_sarsa))
# Update the weight vector for Sarsa(0) using the gradient descent update rule
weights_sarsa += alpha * td_error * get_features(state, action_sarsa)
```

For Semi-Gradient TD(λ):

```
# Compute the TD error for TD(λ)
td_error_td = reward + gamma * np.dot(weights_td, get_features(next_state)) - np.dot(weights_t
# Update the eligibility trace for TD(λ)
eligibility_trace_td = gamma * lambda_ * eligibility_trace_td + get_features(state, action_td)
# Update the weight vector for TD(λ) using the gradient descent update rule with eligibility t
weights_td += alpha * td_error_td * eligibility_trace_td
```

- Semi-Gradient TD(λ) uses an eligibility trace to determine how much credit or blame to assign to current and previous state-action pairs. By adjusting the eligibility trace decay rate (lambda), the algorithm can control the temporal credit assignment and the impact of past experiences. This enhances the learning process by considering a longer-term credit assignment. Semi-Gradient Sarsa(0) does not use an eligibility trace and only considers the TD error of the current state-action pair.

In Semi-Gradient TD(λ):

```
# Update the eligibility trace for TD(λ)
eligibility_trace_td = gamma * lambda_ * eligibility_trace_td + get_features(state, action_td)
```

- Semi-Gradient TD(λ) uses the lambda parameter to control the decay rate of the eligibility trace and determine the time scale over which credit is assigned. Here, lamda_ = 0.5 provides a balanced approach between considering immediate rewards

(lambda close to 0) and future estimates (lambda close to 1) when updating the weight vector. This choice allows for a flexible credit assignment, capturing both short-term and long-term dependencies in the learning process. Semi-Gradient Sarsa(0) does not use lambda and assigns credit only to the current state-action pair.

In Semi-Gradient TD(λ), lambda is defined as a hyperparameter:

```
lambda_ = 0.5  # eligibility trace parameter
```

- Semi-Gradient Sarsa(0) calculates the average reward per episode by taking the mean of the rewards of the current episode and the previous 99 episodes. This smoothing can provide a more stable measure of performance, especially if there is high variability in rewards. Semi-Gradient TD(λ) calculates the average reward per episode by taking the mean of the rewards of the current episode and the previous episodes up to the current episode. This measure gives more weight to recent episodes and may provide a faster response to changes in performance.

For Semi-Gradient Sarsa(0):

```
# Calculate the average reward per episode
if len(rewards_sarsa) >= 100:
    avg_reward = sum(rewards_sarsa[-100:]) / 100
else:
    avg_reward = sum(rewards_sarsa) / len(rewards_sarsa)
```

For Semi-Gradient TD(λ):

```
# Store the total reward for the current episode
rewards_td.append(episode_reward)

# Calculate the average reward per episode
if len(rewards_td) >= 100:
    avg_reward = sum(rewards_td[-100:]) / 100
else:
    avg_reward = sum(rewards_td) / len(rewards_td)
```

Both the Semi-Gradient TD(λ) and Sarsa(0) methods for Mountain Car environment yield a total reward of -200, but TD(λ) shows a higher average reward per episode (as seen in the graph below), it suggests that TD(λ) is more effective in exploring the environment and learning a better policy compared to Sarsa(0). The higher average reward per episode obtained with TD(λ) indicates that, on average, the agent is achieving better performance and receiving higher cumulative rewards over multiple episodes. This suggests that TD(λ) is able to learn a more optimal policy or make more informed decisions compared to Sarsa(0).


Average Reward per Episode vs. Episodes
TD(λ)
SARSA(0)
Average Reward per Episode
−186
−188
−190
−192
−194
−196
−198
−200
0
20
40
60
80
100
Episodes