Homework2 – Code Analysis

Guodong Dong

gvd5289@psu.edu

Pennsylvania State University

College of Information Sciences and Technology

CSE 584: Machine Learning: Tools and Algorithms

Prof. Wenpeng Yin

October 27, 2024

Content

Reinforcement Learning Algorithm

Reinforcement Learning (RL) focuses on teaching agents how to make decisions that maximize a long-term goal. In RL, an agent interacts with an environment by observing its state, taking actions, and receiving feedback in the form of rewards. The key challenge in RL is that these rewards are often delayed, so the agent must learn to map immediate actions to future benefits, aiming to optimize its overall performance over time. One approach in RL is using Monte Carlo (MC) methods, which enable the agent to learn directly from the experiences it gains by interacting with the environment. MC methods rely on episodes of experience, where each episode is a sequence of events represented as (State, Action, Reward, Next State) tuples. These episodes capture the agent's journey through different states and the corresponding rewards it receives (Sutton et al., 2018).

Here is a code that implements MC prediction for estimating the value function of a Blackjack environment in OpenAI's Gym.

```
%matplotlib inline

import gym
import matplotlib
import numpy as np
import sys

from collections import defaultdict

if "../" not in sys.path:
  sys.path.append("../")
from lib.envs.blackjack import BlackjackEnv
from lib import plotting

matplotlib.style.use('ggplot')
```

```
env = BlackjackEnv()
```

```
def mc_prediction(policy, env, num_episodes, discount_factor=1.0):
    """
    Monte Carlo prediction algorithm. Calculates the value function
    for a given policy using sampling.

    Args:
        policy: A function that maps an observation to action probabilities.
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        discount_factor: Gamma discount factor.

    Returns:
        A dictionary that maps from state -> value.
        The state is a tuple and the value is a float.
    """

    # Keeps track of sum and count of returns for each state
    # to calculate an average. We could use an array to save all
    # returns (like in the book) but that's memory inefficient.
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The final value function
    V = defaultdict(float)

    for i_episode in range(1, num_episodes + 1):
        # Print out which episode we're on, useful for debugging.
        if i_episode % 1000 == 0:
            print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
            sys.stdout.flush()

        # Generate an episode.
        # An episode is an array of (state, action, reward) tuples
        episode = []
        state = env.reset()
        for t in range(100):
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        # Find all states the we've visited in this episode
        # We convert each state to a tuple so that we can use it as a dict key
        states_in_episode = set([tuple(x[0]) for x in episode])
        for state in states_in_episode:
            # Find the first occurance of the state in the episode
            first_occurence_idx = next(i for i,x in enumerate(episode) if x[0] == state)
            # Sum up all rewards since the first occurance
            G = sum([x[2]*(discount_factor**i) for i,x in enumerate(episode[first_occurence_idx:])])
            # Calculate average return for this state over all sampled episodes
            returns_sum[state] += G
            returns_count[state] += 1.0
            V[state] = returns_sum[state] / returns_count[state]

    return V
```

```python
def sample_policy(observation):
    """
    A policy that sticks if the player score is >= 20 and hits otherwise.
    """
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

```python
V_10k = mc_prediction(sample_policy, env, num_episodes=10000)
plotting.plot_value_function(V_10k, title="10,000 Steps")

V_500k = mc_prediction(sample_policy, env, num_episodes=500000)
plotting.plot_value_function(V_500k, title="500,000 Steps")
```

Abstract

This code implements an MC prediction algorithm to estimate the value function for a policy-driven agent in a Blackjack environment. The purpose of the algorithm is to approximate the expected return (cumulative future reward) from each game state while following a predefined policy, allowing the agent to improve its understanding of the game dynamics over time. The environment used is OpenAI Gym's Blackjack environment, where an agent interacts with the game by making decisions based on the current state of the game (Britz, 2024).

The core process consists of running multiple episodes of Blackjack, during which the agent follows a policy where it "sticks" if its score is 20 or greater and "hits" otherwise. Each episode produces a sequence of states, actions, and rewards, which the algorithm uses to update the value function V(s) by averaging the returns for each state encountered. The value function estimates how beneficial it is for the agent to be in each state under the given policy, guiding future decision-making in the Blackjack game. After a specified number of episodes, the learned value function is visualized to provide insights into how favorable each state is.

This code demonstrates fundamental concepts of reinforcement learning, such as state-value estimation, MC methods for prediction, and policy-based decision-making. The resulting value function provides the agent with a better understanding of the expected long-term reward for various game states, helping improve the agent's strategy for future games.

Analysis

This code uses MC Prediction to estimate the value function for a Blackjack game under a specific policy. By running many episodes (simulated games), it learns the expected return for each state.

Environment Setup

This snippet sets up the necessary libraries and environment to build and analyze a RL agent that plays Blackjack. It loads the Blackjack environment, ensures proper file paths for custom modules, and prepares for data visualization. This environment provides the state of the game, allows actions (hit or stick), and returns the outcome (reward) after each action.

```
%matplotlib inline

import gym
import matplotlib
import numpy as np
import sys

from collections import defaultdict

if "../" not in sys.path:
  sys.path.append("../")
from lib.envs.blackjack import BlackjackEnv
from lib import plotting

matplotlib.style.use('ggplot')
```

```
env = BlackjackEnv()
```

MC Prediction Algorithm

This function implements the MC prediction algorithm. It estimates the value function for a given policy using sampled episodes from the environment. The value function represents the expected return (reward) from a given state, assuming the agent follows a specified policy. It generates episodes by interacting with the environment, accumulates the rewards for each state, and calculates the average return to estimate the expected value of each state.

```
def mc_prediction(policy, env, num_episodes, discount_factor=1.0):
    """
    Monte Carlo prediction algorithm. Calculates the value function
    for a given policy using sampling.

    Args:
        policy: A function that maps an observation to action probabilities.
        env: OpenAI gym environment.
        num_episodes: Number of episodes to sample.
        discount_factor: Gamma discount factor.

    Returns:
        A dictionary that maps from state -> value.
        The state is a tuple and the value is a float.
    """

    # Keeps track of sum and count of returns for each state
    # to calculate an average. We could use an array to save all
    # returns (like in the book) but that's memory inefficient.
    returns_sum = defaultdict(float)
    returns_count = defaultdict(float)

    # The final value function
    V = defaultdict(float)
```

This function takes four parameters: policy, env, num_episodes and discount_factor. It sets up three variables for storing results.

```
for i_episode in range(1, num_episodes + 1):
    # Print out which episode we're on, useful for debugging.
    if i_episode % 1000 == 0:
        print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
        sys.stdout.flush()
```

This loop runs the simulation for num_episodes iterations. Every 1000 episodes, it prints the current progress.

```
# Generate an episode.
# An episode is an array of (state, action, reward) tuples
episode = []
state = env.reset()
for t in range(100):
    action = policy(state)
    next_state, reward, done, _ = env.step(action)
    episode.append((state, action, reward))
    if done:
        break
    state = next_state
```

episode = []: This list stores the sequence of states, actions, and rewards encountered in one episode.

env.reset(): It resets the environment at the beginning of each new episode, returning the initial state.

The for loop:

The policy(state) function determines the action based on the current state.

env.step(action): This takes the action and returns:

next_state: The new state after the action.

reward: The reward for taking that action.

done: Whether the game is over.

The tuple (state, action, reward) is stored in the episode. If the episode ends (done=True), the loop breaks.

```
    # Find all states the we've visited in this episode
    # We convert each state to a tuple so that we can use it as a dict key
    states_in_episode = set([tuple(x[0]) for x in episode])
    for state in states_in_episode:
        # Find the first occurance of the state in the episode
        first_occurence_idx = next(i for i,x in enumerate(episode) if x[0] == state)
        # Sum up all rewards since the first occurance
        G = sum([x[2]*(discount_factor**i) for i,x in enumerate(episode[first_occurence_idx:])])
        # Calculate average return for this state over all sampled episodes
        returns_sum[state] += G
        returns_count[state] += 1.0
        V[state] = returns_sum[state] / returns_count[state]

return V
```

After the episode finishes, we calculate the return (G) for each state visited.

states_in_episode: It extracts and converts the states in the episode into a set of unique states to avoid counting duplicate visits within the same episode.

For each state:

first_occurence_idx: It finds the first occurrence of each state in the episode.

This is important in MC prediction because we only care about the first time we visit a state within an episode when calculating the return.

G: The return (total future reward) is the cumulative reward from the first occurrence of the state until the end of the episode. The discount factor ensures that rewards in the future are worth less. The sum of returns and the count of visits to the state are updated.

V[state]: The value of the state is updated as the average return across all episodes where that state was encountered.

After running all the episodes, the function returns the value function V, which is a dictionary mapping each state to its estimated value.

Policy

This policy decides to stick (take no further action) if the player's score is 20 or above and hit (draw another card) otherwise. The policy determines the agent's behavior and serves as the input to the mc_prediction function.

```
def sample_policy(observation):
    """
    A policy that sticks if the player score is >= 20 and hits otherwise.
    """
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1
```

Running MC Prediction

The MC prediction runs for 10,000 episodes and 500,000 episodes and the results are saved in two variables, which represent the estimated value functions. The value function is then visualized using a plotting function, which shows how the value function improves with more episodes.

```
V_10k = mc_prediction(sample_policy, env, num_episodes=10000)
plotting.plot_value_function(V_10k, title="10,000 Steps")

V_500k = mc_prediction(sample_policy, env, num_episodes=500000)
plotting.plot_value_function(V_500k, title="500,000 Steps")
```

References

Sutton, R. S., Bach, F., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT Press Ltd.

Britz, D. (2024, October 27). reinforcement-learning. GitHub. https://github.com/dennybritz/reinforcement-learning/