Project 2: TD methods and n-step bootstrapping

August Jonasson

February 25, 2025

Part 1: Expected SARSA and importance sampling ratio

Suppose environment consists of four states $\{s_0, s_1, s_2, s_3\}$, with an action space $\{+, -\}$ and deterministic dynamics $p(s_{i+1}|s_i, +) = p(s_{i-1}|s_i, -) = 1$, with exceptions $p(s_0|s_3, +) = p(s_3|s_0, -) = 1$.

Task 1

Given policy $\pi(+) = \pi(-) = 0.5,\ \forall s$, and having observed the transition

$$(s_0, +) \to (s_1, r = 2),$$

we want to compute the one-step *expected SARSA* update of the action-value estimate $Q(s_0, +)$. In order to do this, we assume initial estimates $Q(s, a) = 1,\ \forall s, a$, learning rate $\alpha = 0.2$, and discount factor $\gamma = 0.9$. Now, using equation (6.9) from the course book (Sutton 2018, p. 133)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

with the given policy, parameters, values and observations, we get the one-step update

$$Q(s_0, +) \leftarrow 1 + 0.2\,[2 + 0.9(0.5 + 0.5) - 1] = 1.38,$$

and we are done.

Task 2

Suppose an agent is following a behavior policy $b(+) = 0.7$ and $b(-) = 0.3$, with the target policy equal to that of the previous task. Having observed the trajectory

$$(s_0, +) \to (s_1, +) \to s_3,$$

we want to compute the *importance sampling ratio* $\rho_{0:1}$. In order to do this, we simply use equation (5.3) from the course book (p. 104), which defines

$$\rho_{t:T-1} := \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}.$$

Plugging in our policies for the two time steps, we get

$$\rho_{0:1} = \left(\frac{0.5}{0.7}\right)^2 \approx 0.51,$$

which can be interpreted as: "the observed trajectory is roughly half as likely to occur under the target policy than under the behavior policy", and we are done.

Part 2: TD methods applied to gridworld with a monster

In this task, an agent acts in an N^2 gridworld with a randomly moving monster and a randomly spawned, stationary apple. The goal of the agent is to collect the apple as many times as possible before the episode is over, without getting caught by the monster.

Each state is represented by three tuples (agent position, monster position and apple position) and the entire space is given by

$$\mathcal{S} = \big\{(x_p, y_p), (x_m, y_m), (x_a, y_a) \,\big|\, x_p, y_p, x_m, y_m, x_a, y_a \in \{0, \dots N-1\}\big\},$$

i.e. $|\mathcal{S}| = N^6$. The action space is

$$\mathcal{A} = \{\text{left}, \text{up}, \text{right}, \text{down}\},$$

which yields that the number of state-action pairs is $4 \cdot N^6$.

Some dynamics of the system are:

- the agent moves deterministically according to chosen action; if a wall is hit, the agent instead remains in place,
- the monster moves uniformly random in all four directions; a wall hit is the same as for the agent,
- the apple, if collected by agent, randomly respawns in a new empty cell,
- an episode ends if the monster catches the agent, or after T time steps.

Given a state-action pair, the rewards are

$$R(s,a) = \begin{cases} +\quad 1 & \text{if the agent collects an apple,} \\ -\quad 1 & \text{if the agent is caught by the monster,} \\ \quad\;\; 0 & \text{otherwise.} \end{cases}$$

For the following tasks, we use hyperparameters:

- side length $N = 5$,
- eipsode length $T = 30$,
- discont factor $\gamma = 0.9$,
- learning rate $\alpha = 0.1$ and
- all state-action value estimates $Q(s,a)$ initialized at zero.

Task 1: write code

In this first task we implement the gridworld system as defined above, as well as the following TD methods, using an ϵ-greedy policy, for action-value estimation: SARSA, off-policy Q-learning, Double Q-learning, and n-step SARSA.

See Appendix 1 for the Python code.

Task 2: plot the learning curves

In this task we are to answer which method converges the fastest and we also want to decide which n is optimal, in the n-step SARSA method.

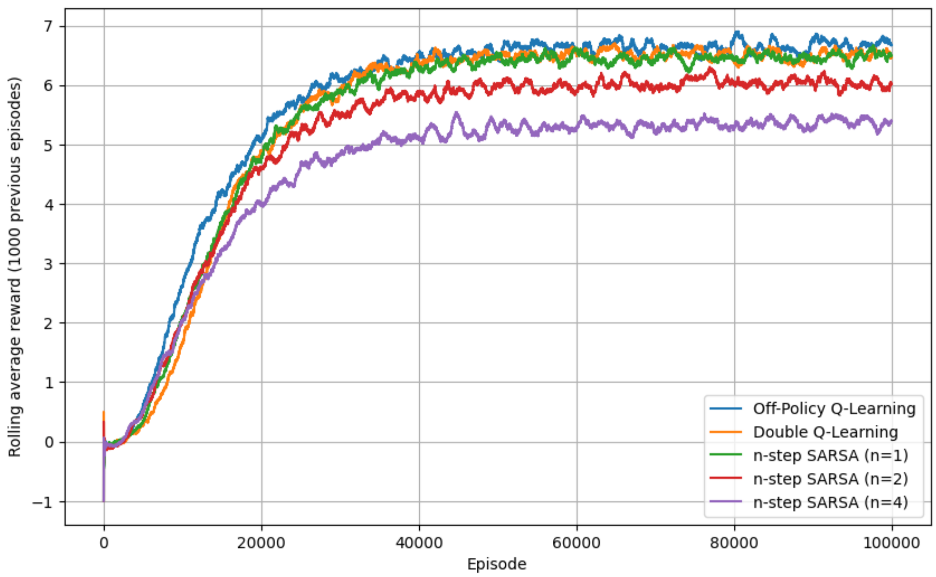

Figure 1: Moving average reward per episode (each average computed with previous 1000 episodes) for the different TD methods using ϵ-greedy policy with exponentially decaying ϵ as a function of the current number of episodes (such that $\epsilon_1 = 1.0$ and $\epsilon_{10^5} \approx 0.01$).

From Figure 1 we see that all methods have similar convergence rates. For all five of them, convergence seems to happen after somewhere between 40 and 60 thousand episodes.

Considering the n-step SARSA methods, as we already mentioned, they converge at similar rates. Thus, if we are to answer which one is optimal we look to the one that produces the highest average reward, which is $n = 1$, and we are done.

Task 3: discussion

In this last task, we discuss the following:

1. why all three n-step SARSA methods have similar rates of convergence,
2. why the apparent differences between regular Q-learning and Double Q-learning,
3. how convergence would be affected if the monster instead moves deterministically towards the agent.

1)

There is a case to be made for the bias-variance trade-off when it comes to these n-steps methods. A lower n would correspond to a higher bias but low variance wheras a higher n would have the reversed effects. Why we see so little variation in terms of rate of convergence among the three different n could be that the trade-offs are pretty linear. Maybe it has something to do with the randomness of how the monster moves.

2)

The regular Q-learning method potentially suffers from the maximization bias (p. 134 in the course book), which means that we expect it to overestimate its action values early on in the training. The Double Q-learning method is the remedy to this bias and as such, it produces more reliable action values throughout the whole training. As we, in Figure 1, are looking at the actual rewards that result from the potentially biased policy and not the action-values themselves, the bias isn't apparent. We do, however, see a more rapid increase in reward early on from the Q-learning method, before the methods seem to converge at very similar values. This indicates that in this case, the maximization bias might actually be beneficial to early rewards.

3)

Convergence should be quicker as there are less paths to consider. Instead of ≈ 4 monster moves per agent action, with a deterministically moving monster the number of paths should at least be halved. Thus leading to a smaller state-action space, thus leading to faster exploring of the entire space, thus leading to faster convergence.

Appendix 1: Python code

```
# %%
import numpy as np
import random
from matplotlib import pyplot as plt
import pandas as pd

# Gridworld parameters
N = 5                    # grid size (N x N)
T = 30                   # maximum time steps per episode
alpha = 0.1              # learning rate
gamma = 0.9              # discount factor
episodes = 10**5         # number of episodes per method

# Create grid coordinates
grid = [(r, c) for r in range(N) for c in range(N)]

# Define actions and their effects (using (row, col) convention)
actions = ["up", "right", "down", "left"]
actions_dict = {
    "up": (-1, 0),
    "right": (0, 1),
    "down": (1, 0),
    "left": (0, -1)
}

# Define rewards
rewards_dict = {
    "collect_apple": 1,
    "caught_by_monster": -1,
    "empty": 0
}

# Parameters for epsilon decay
epsilon_0 = 1.0   # Initial exploration rate
epsilon_min = 0.01 # Minimum exploration
decay_rate = 0.0001 # Controls speed of decay

def get_epsilon(episode):
    """ Exponentially decaying epsilon """
    return epsilon_min + (epsilon_0 - epsilon_min) * np.exp(-decay_rate * episode)

def initial_positions():
    """Return distinct starting positions for agent, monster, and apple."""
    return tuple(random.sample(grid, 3))

def respawn_apple(agent_pos, monster_pos):
    """Return a new apple position that is not occupied by agent or monster."""
    available_positions = [pos for pos in grid if pos != agent_pos and pos != monster_pos]
    return random.choice(available_positions)

def move(pos, action):
    """
    Given a position and an action, return the new position.
    If the move is out-of-bounds, return the original position.
    """
```

```python
    delta = actions_dict[action]
    new_r = pos[0] + delta[0]
    new_c = pos[1] + delta[1]
    if 0 <= new_r < N and 0 <= new_c < N:
        return (new_r, new_c)
    else:
        return pos

def step(state, agent_action, monster_action):
    """
    Execute one simultaneous step for agent and monster.
    Returns next_state, reward, and a done flag.
    """
    agent_pos, monster_pos, apple_pos = state
    new_agent_pos = move(agent_pos, agent_action)
    new_monster_pos = move(monster_pos, monster_action)

    # Check if agent and monster collide.
    if new_agent_pos == new_monster_pos:
        return (new_agent_pos, new_monster_pos, apple_pos), rewards_dict["caught_by_monster"], True

    # Check for apple collection.
    reward = rewards_dict["empty"]
    if new_agent_pos == apple_pos:
        reward = rewards_dict["collect_apple"]
        new_apple_pos = respawn_apple(new_agent_pos, new_monster_pos)
    else:
        new_apple_pos = apple_pos

    next_state = (new_agent_pos, new_monster_pos, new_apple_pos)
    return next_state, reward, False

def epsilon_greedy(Q, state, epsilon):
    """
    Return an action chosen by the epsilon-greedy policy based on Q.
    """
    if state not in Q or random.random() < epsilon:
        return random.choice(actions)
    max_val = max(Q[state].values())
    best_actions = [a for a, v in Q[state].items() if v == max_val]
    return random.choice(best_actions)

def epsilon_greedy_double(Q1, Q2, state, epsilon):
    """
    For double Q-learning: choose an action using the sum of Q1 and Q2 values.
    """
    if state not in Q1 or state not in Q2 or random.random() < epsilon:
        return random.choice(actions)
    combined = {a: Q1[state][a] + Q2[state][a] for a in actions}
    max_val = max(combined.values())
    best_actions = [a for a, v in combined.items() if v == max_val]
    return random.choice(best_actions)

# %%
# ---------------------------
# Off-Policy Q-Learning
# ---------------------------
```

```python
Q_learning = {}
Q_learning_rewards = []

print("Starting Off-Policy Q-Learning...")
for episode in range(episodes):
    state = initial_positions()  # (agent_pos, monster_pos, apple_pos)
    if state not in Q_learning:
        Q_learning[state] = {a: 0.0 for a in actions}

    epsilon = get_epsilon(episode)
    total_reward = 0
    t = 0
    done = False
    while t < T and not done:
        # Select action using Q_learning's own epsilon-greedy.
        agent_action = epsilon_greedy(Q_learning, state, epsilon)
        monster_action = random.choice(actions)

        next_state, reward, done = step(state, agent_action, monster_action)
        total_reward += reward

        if not done:
            if next_state not in Q_learning:
                Q_learning[next_state] = {a: 0.0 for a in actions}
            best_next = max(Q_learning[next_state].values())
        else:
            best_next = 0

        Q_learning[state][agent_action] += alpha * (reward + gamma * best_next - Q_learning[state][agent_action])
        state = next_state
        t += 1

    # save accumulated episode reward
    Q_learning_rewards.append(total_reward)

    if (episode + 1) % 100 == 0:
        print(f"Off-Policy Q-Learning Episode {episode+1}: Total Reward = {total_reward}")

# %%
# ---------------------------
# 1-Step SARSA
# ---------------------------
Q_sarsa = {}
Q_sarsa_rewards = []

print("\nStarting 1-Step SARSA...")
for episode in range(episodes):
    state = initial_positions()
    if state not in Q_sarsa:
        Q_sarsa[state] = {a: 0.0 for a in actions}
    # Choose initial action using Q_sarsa.
    agent_action = epsilon_greedy(Q_sarsa, state, epsilon)

    epsilon = get_epsilon(episode)
    total_reward = 0
    t = 0
```

```
    done = False
    while t < T and not done:
        if state not in Q_sarsa:
            Q_sarsa[state] = {a: 0.0 for a in actions}
        monster_action = random.choice(actions)
        next_state, reward, done = step(state, agent_action, monster_action)
        total_reward += reward

        if not done:
            if next_state not in Q_sarsa:
                Q_sarsa[next_state] = {a: 0.0 for a in actions}
            next_action = epsilon_greedy(Q_sarsa, next_state, epsilon)
            Q_sarsa[state][agent_action] += alpha * (reward + gamma * Q_sarsa[next_state][next_action] - Q_sarsa[state][agent_action])
        else:
            Q_sarsa[state][agent_action] += alpha * (reward - Q_sarsa[state][agent_action])

        state = next_state
        if not done:
            agent_action = next_action
        t += 1

    Q_sarsa_rewards.append(total_reward)
    if (episode + 1) % 100 == 0:
        print(f"SARSA Episode {episode+1}: Total Reward = {total_reward}")

# --------------------------
# Double Q-Learning
# --------------------------
# In double Q-learning, the behavior policy is derived from the sum of Q_double1 and Q_double2.
Q_double1 = {}
Q_double2 = {}
Q_double_rewards = []

print("\nStarting Double Q-Learning...")
for episode in range(episodes):
    state = initial_positions()
    for Q in (Q_double1, Q_double2):
        if state not in Q:
            Q[state] = {a: 0.0 for a in actions}

    epsilon = get_epsilon(episode)
    total_reward = 0
    t = 0
    done = False
    # Choose initial action using the double Q behavior policy.
    agent_action = epsilon_greedy_double(Q_double1, Q_double2, state, epsilon)

    while t < T and not done:
        for Q in (Q_double1, Q_double2):
            if state not in Q:
                Q[state] = {a: 0.0 for a in actions}
        monster_action = random.choice(actions)
        next_state, reward, done = step(state, agent_action, monster_action)
        total_reward += reward

```

```
        for Q in (Q_double1, Q_double2):
            if next_state not in Q:
                Q[next_state] = {a: 0.0 for a in actions}

        if not done:
            next_action = epsilon_greedy_double(Q_double1, Q_double2, next_state, epsilon)
            # Randomly update one of the two Q-tables.
            if random.random() < 0.5:
                best_action = max(Q_double1[next_state], key=Q_double1[next_state].get)
                target = reward + gamma * Q_double2[next_state][best_action]
                Q_double1[state][agent_action] += alpha * (target - Q_double1[state][agent_action])
            else:
                best_action = max(Q_double2[next_state], key=Q_double2[next_state].get)
                target = reward + gamma * Q_double1[next_state][best_action]
                Q_double2[state][agent_action] += alpha * (target - Q_double2[state][agent_action])
        else:
            # Terminal state update.
            if random.random() < 0.5:
                Q_double1[state][agent_action] += alpha * (reward - Q_double1[state][agent_action])
            else:
                Q_double2[state][agent_action] += alpha * (reward - Q_double2[state][agent_action])

        state = next_state
        if not done:
            agent_action = next_action
        t += 1

    Q_double_rewards.append(total_reward)
    if (episode + 1) % 100 == 0:
        print(f"Double Q-Learning Episode {episode+1}: Total Reward = {total_reward}")

# At the end, Q_learning, Q_sarsa, and (Q_double1, Q_double2) hold the learned action values for each method.

# %%
# ---------------------------
# n-step SARSA Function
# ---------------------------
def n_step_sarsa(n, episodes):
    """
    Runs n-step SARSA on the gridworld for a given number of episodes.
    No pre-initialization of all states is needed; states are added as encountered.

    Parameters:
      n        : number of steps for bootstrapping
      episodes : number of episodes to run

    Returns:
      Q_n      : learned Q-value table (dictionary)
      rewards_n: list of total rewards per episode
    """
```

```
    Q_n = {}
    rewards_n = []
    for episode in range(episodes):
        epsilon_val = get_epsilon(episode)
        # Initialize starting state and Q-values for that state if needed.
        state = initial_positions()
        if state not in Q_n:
            Q_n[state] = {a: 0.0 for a in actions}
        # Choose initial action using the epsilon-greedy policy.
        action = epsilon_greedy(Q_n, state, epsilon_val)

        # Lists to store the trajectory:
        states = [state]            # states[0] is the initial state
        actions_list = [action]     # actions_list[0] is the initial action
        rewards_list = [0]          # rewards_list[0] is a dummy reward

        T_episode = 30
        t = 0
        total_reward = 0

        while True:
            if t < T_episode:
                monster_action = random.choice(actions)
                next_state, reward, done = step(state, action, monster_action)
                total_reward += reward
                rewards_list.append(reward)
                states.append(next_state)
                if done:
                    T_episode = t + 1
                else:
                    if next_state not in Q_n:
                        Q_n[next_state] = {a: 0.0 for a in actions}
                    next_action = epsilon_greedy(Q_n, next_state, epsilon_val)
                    actions_list.append(next_action)
            tau = t - n + 1
            if tau >= 0:
                # Determine the upper index for the reward sum.
                limit = min(tau + n, T_episode)
                G = 0.0
                for i in range(tau + 1, limit + 1):
                    G += (gamma ** (i - tau - 1)) * rewards_list[i]
                if tau + n < T_episode:
                    G += (gamma ** n) * Q_n[states[tau + n]][actions_list[tau + n]]
                Q_n[states[tau]][actions_list[tau]] += alpha * (G - Q_n[states[tau]][actions_list[tau]])
            t += 1
            if tau == T_episode - 1:
                break
            # Update state and action only if available.
            if t < len(states):
                state = states[t]
            if t < len(actions_list):
                action = actions_list[t]
        rewards_n.append(total_reward)
        if (episode + 1) % 1000 == 0:
            print(f"n-step SARSA (n={n}) Episode {episode+1}: Total Reward = {total_reward}")
    return Q_n, rewards_n
```

```

# %% [code]
# ---------------------------
# Run n-step SARSA for different n values
# ---------------------------
# print("\nStarting n-step SARSA (n=2)...")
_, rewards_n2 = n_step_sarsa(2, episodes)

# print("\nStarting n-step SARSA (n=4)...")
_, rewards_n4 = n_step_sarsa(4, episodes)

# For comparison, n=1-step SARSA is equivalent to standard 1-step SARSA:
print("\nStarting n-step SARSA (n=1) [equivalent to 1-step SARSA]...")
_, rewards_n1 = n_step_sarsa(1, episodes)


# %%
# ---------------------------
# Plotting the Learning Curves
# ---------------------------
avg_len = 1000

# creating lists of average over 1000 last episodes
Q_sarsa_avg = pd.Series(Q_sarsa_rewards).rolling(window=avg_len, min_periods=1).mean().to_numpy()
Q_learning_avg = pd.Series(Q_learning_rewards).rolling(window=avg_len, min_periods=1).mean().to_numpy()
Q_dbl_avg = pd.Series(Q_double_rewards).rolling(window=avg_len, min_periods=1).mean().to_numpy()

# Compute rolling averages over the last 1000 episodes
n1_avg = pd.Series(rewards_n1).rolling(window=avg_len, min_periods=1).mean().to_numpy()
n2_avg = pd.Series(rewards_n2).rolling(window=avg_len, min_periods=1).mean().to_numpy()
n4_avg = pd.Series(rewards_n4).rolling(window=avg_len, min_periods=1).mean().to_numpy()

episodes_range = list(range(1, episodes + 1))
plt.figure(figsize=(10, 6))
plt.plot(episodes_range, Q_learning_avg, label="Off-Policy Q-Learning")
# plt.plot(episodes_range, Q_sarsa_avg, label="1-Step SARSA")
plt.plot(episodes_range, Q_dbl_avg, label="Double Q-Learning")
plt.plot(episodes_range, n1_avg, label="n-step SARSA (n=1)")
plt.plot(episodes_range, n2_avg, label="n-step SARSA (n=2)")
plt.plot(episodes_range, n4_avg, label="n-step SARSA (n=4)")
plt.xlabel("Episode")
plt.ylabel("Rolling average reward (1000 previous episodes)")
plt.legend()
plt.grid(True)
plt.show()
# %%
```

References

Sutton, Richard S (2018). “Reinforcement learning: An introduction”. In: *A Bradford Book*.