Project 3: Approximation Methods and Policy Gradients

August Jonasson

March 10, 2025

Part 1: gradients of update steps in SGD

In this first part we consider the parameter update steps in approximation methods (Sutton 2018, ch. 9) and policy gradient methods (ch. 13). More specifically, the tasks will consist of deriving the gradients needed in order to perform these update steps, given some model assumptions.

Task 1

In approximation methods of the state-value v_π, we use SGD with the update rule

$$\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta \hat{v}(s, \theta_t), \tag{1}$$

where α is the learning rate, $\delta_t = v_\pi(s) - \hat{v}(s, \theta_t)$ is the estimation error at time t, and ∇_θ denotes the gradient w.r.t. the parameter vector θ.

Suppose we have a linear approximator $\hat{v}(s, \theta) = \theta^T x(s)$, where $x(s)$ denotes some feature of state s, and show that θ updates according to

$$\theta_{t+1} = \theta_t + \alpha \delta_t x(s).$$

Using (1), all we have to show is that $\nabla_\theta \hat{v}(s, \theta) = x(s)$, i.e. that

$$\frac{\partial(\hat{v}(s, \theta))}{\partial \theta} = x(s).$$

Let $\theta, x(s) \in \mathbb{R}^p$, then (using the short-hand $\hat{v} = \hat{v}(s, \theta)$)

$$\frac{\partial \hat{v}}{\partial \theta} = \left[\frac{\partial \hat{v}}{\partial \theta_1}, \ldots, \frac{\partial \hat{v}}{\partial \theta_p}\right]^T,$$

with

$$\frac{\partial \hat{v}}{\partial \theta_i} = \frac{\partial(\theta^T x(s))}{\partial \theta_i} = \frac{\partial}{\partial \theta_i}\left(\sum_{j=1}^{p} \theta_j x(s)_j\right) = x(s)_i,$$

for $1 \le i \le p$. Thus

$$\frac{\partial \hat{v}}{\partial \theta} = [x(s)_1, \ldots, x(s)_p]^T = x(s),$$

and we are done.

Task 2

For policy gradient methods, we update the weights according to

$$\theta_{t+1} = \theta_t + \alpha\delta_t \frac{\nabla_\theta \pi(a|s;\theta_t)}{\pi(a|s;\theta_t)}. \tag{2}$$

(a)

Show that

$$\frac{\nabla_\theta \pi(a|s;\theta)}{\pi(a|s;\theta)} = \nabla_\theta \log \pi(a|s;\theta).$$

Starting from the right-hand side (with short-hand $\pi = \pi(a|s;\theta)$), we have that

$$\nabla_\theta \log \pi = \left[\frac{\partial \log \pi}{\partial \theta_1}, \ldots, \frac{\partial \log \pi}{\partial \theta_p}\right]^T,$$

where

$$\frac{\partial \log \pi}{\partial \theta_i} = \frac{1}{\pi}\frac{\partial \pi}{\partial \theta_i}$$

for $1 \leq i \leq p$. Thus

$$\nabla_\theta \log \pi = \frac{1}{\pi}\left[\frac{\partial \pi}{\partial \theta_1}, \ldots, \frac{\partial \pi}{\partial \theta_p}\right] = \frac{\nabla_\theta \pi(a|s;\theta)}{\pi(a|s;\theta)}, \tag{3}$$

and we are done.

(b)

Suppose the policy is modeled using a softmax function

$$\pi(a|s;\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}, \tag{4}$$

with linear input $h(s,a,\theta) = \theta^T x(s,a)$ for some state-action feature function x. Show that the policy update in (2) is given by

$$\theta_{t+1} = \theta_t + \alpha\delta_t \left(x(s,a) - \sum_b \pi(b|s;\theta_t) x(s,b) \right).$$

Using (2) and (3), all we have to show is that

$$\nabla_\theta \log \pi(a|s;\theta) = x(s,a) - \sum_b \pi(b|s;\theta) x(s,b).$$

Starting from the left-hand side, we first have that

$$\log \pi(a|s;\theta) = \theta^T x(s,a) - \log\left(\sum_b e^{\theta^T x(s,b)}\right),$$

from which using the results from Task 1, the partial derivatives w.r.t. θ of the first term can be directly simplified to the sought after $x(s,a)$. The derivatives of the second term can be manipulated in the following way:

$$\begin{aligned}\frac{\partial}{\partial\theta}\log\left(\sum_b e^{\theta^T x(s,b)}\right) &= \frac{1}{\sum_b e^{\theta^T x(s,b)}}\frac{\partial}{\partial\theta}\left(\sum_b e^{\theta^T x(s,b)}\right)\\ &= \frac{1}{\sum_b e^{\theta^T x(s,b)}}\left(\sum_b \frac{\partial}{\partial\theta}e^{\theta^T x(s,b)}\right),\end{aligned}$$

and again, using the results from Task 1, the partial derivative within the sum reduces to $x(s,b)e^{\theta^T x(s,b)}$, for all b. Putting it all together, we now have

$$\begin{aligned}\nabla_\theta \log\pi(a|s;\theta) &= x(s,a) - \frac{1}{\sum_c e^{\theta^T x(s,c)}}\sum_b x(s,b)e^{\theta^T x(s,b)}\\ &= x(s,a) - \sum_b x(s,b)\frac{e^{\theta^T x(s,b)}}{\sum_c e^{\theta^T x(s,c)}}\\ &= x(s,a) - \sum_b x(s,b)\pi(b|s;\theta),\end{aligned}$$

and we are done.

Part 2: approximation and policy gradient methods for grid-world with a monster

In this second part we are to implement approximation and policy gradient methods in code, for the gridworld with a monster that we worked with in the previous project. Refer back to Project 2 for a full description of system dynamics. We will now state all information that is new.

We set the grid size to $N = 10$, episode length $T = 200$ and discount factor $\gamma = 0.95$. Furthermore, we define the features

$$\begin{aligned} x(s,a) &= [f_1, f_2, f_3, f_4]^T, \quad \text{where}\\ f_1 &= \frac{1}{\text{distance from agent to apple} + 1},\\ f_2 &= \frac{1}{\text{distance from agent to monster} + 1},\\ f_3 &= 1 \text{ if action takes agent closer to apple, else } 0\\ f_4 &= 1 \text{ if action takes agent closer to monster, else } 0,\end{aligned}$$

with the Manhattan distance used as a measure of distance since we are operating on a gridworld. Furthermore, we use the linear approximators for state and action-value

$$\begin{aligned}\hat{q}(s,a,\theta) &= \theta^T x(s,a) \quad \text{and}\\ \hat{v}(s) &= \max_a \hat{q}(s,a,\theta),\end{aligned}$$

and for the policy gradient methods we model the policy as in (4) - using a softmax function with linear input.

Task 1

Implement the following methods:

- Semi-gradient n-step SARSA for $n = 1, 2, 3$, with a suitable ϵ (p. 247 in course book),
- REINFORCE (p. 328 in course book),
- REINFORCE with baseline (p. 330 in course book), and
- One-step Actor-Critic (p. 332 in course book).

See code for full implementation.

Task 2

We are to plot the learning curves for the methods implemented in the previous task.

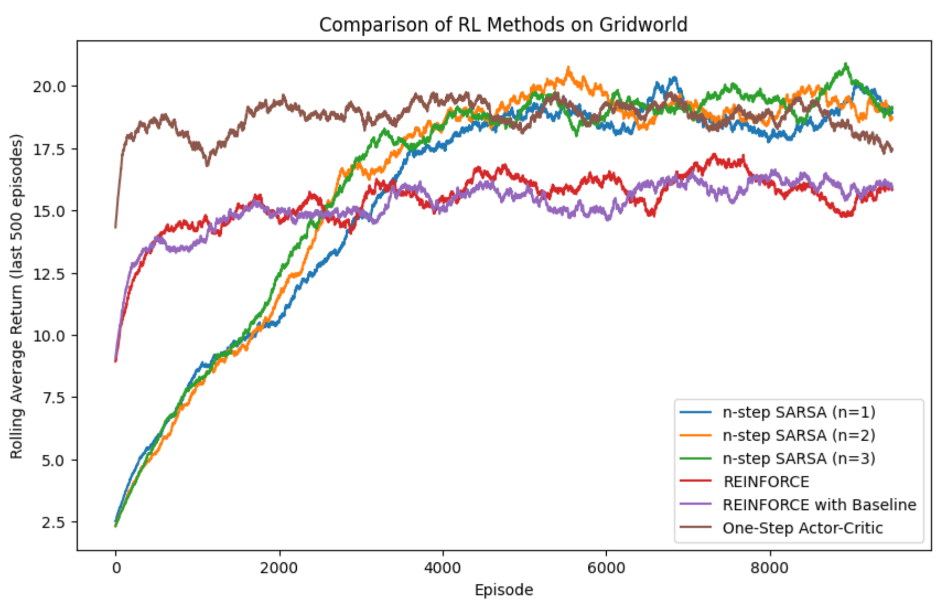

Figure 1: Rolling average (previous 500) of total reward per episode for the different approximation and policy gradient methods.

Task 3

Discuss the following:

1. How does the learning rate α affect the learning speed and convergence of the methods?

2. How do the methods above compare in terms of convergence speed?
3. In this example, does policy gradient methods perform better than the value-based methods? Why or why not?

(1)

Higher learning rate α might lead to faster initial learning but risks continuously over-shooting the "true solution" by a lot, thus leading to oscillations in the learning curve. In general, a smaller α is more safe in the sense that it is more stable and more precise, but comes at the price of slower learning. In most of the methods in Figure 1, we found that $\alpha = 0.01$ could lead to unstable solutions (big oscillations), and thus, we opted for $\alpha = 0.001$ in all methods.

(2)

Methods that use bootstrapping (like one-step actor-critic or n-step SARSA) generally converge faster than pure Monte Carlo methods (e.g. plain REINFORCE). This is because bootstrapping incorporates information from future estimates, which can lead to quicker adjustments of values and policies. So why do the n-step methods in Figure 1 converge the slowest? It is because we are using an exponentially decaying ϵ. At episode 3000, the exploration is still at approximately 23 %, before it at around 6500 episodes drops below 5 %. With this in mind, it is not surprising that the reward has yet to stabilize after so many episodes. We are sacrificing convergence speed for a better policy - and a better policy we do indeed find. With a constant ϵ, the n-step methods performed the worst among all methods. Now, they are arguably the best - as long as we can afford the extra time they take to train.

Among the remaining three episodes, it is difficult to say which one converges the fastest. Based on Figure 1, they all seem rather similar. We can however say that we expect the actor-critic and REINFORCE with baseline methods to converge faster than plain REINFORCE, due to the continual bootstrapping from the critic and baseline respectively. Maybe in this setting, the differences are marginal.

(3)

If we disregard the time it takes to train and focus only on which method achieves the best policy, then the value-based methods (n-step SARSA) seem to perform the best. This is most likely due to the fact that the gridworld system is a very tabular setting. Relatively speaking, there are not that many state-action pairs to consider, and we know that the policy gradient methods really shine the most when the action space is discrete, but the state space continuous.

In summary: gridworld is a low-dimensional well structured and discrete problem. In cases such as these, the simpler value-based methods can capture the necessary dynamics quickly without the additional complexity introduced by additional parameters of e.g. actor-critic or baseline methods. Even though the actor-critic (or REINFORCE with baseline) method may perform as well as the n-step methods, it is simply unnecessary to introduce the extra complexity that comes with policy gradient methods.

Appendix 1: Python code

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

# ---------------------------
# Environment Setup
# ---------------------------
N = 10                  # grid size (N x N)
T = 200                 # maximum time steps per episode
alpha = 0.001            # learning rate
gamma = 0.95            # discount factor
episodes = 10**4        # 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 (used in n-step SARSA)
epsilon_0 = 1.0
epsilon_min = 0.01
decay_rate = 0.0005

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.
```

```
    """
    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 manhattan_distance(pos1, pos2):
    """Return the Manhattan distance between two positions."""
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

# ---------------------------
# Feature Functions
# ---------------------------
def action_state_feature(state, action):
    """
    Computes a 4-dim feature vector x(s,a) given a state and an action.

    Features:
      f1 = 1/(distance from new agent position to apple + 1)
      f2 = 1/(distance from new agent position to monster + 1)
      f3 = 1 if the action moves the agent closer to the apple, else 0
      f4 = 1 if the action moves the agent closer to the monster, else 0
    """
    agent_pos, monster_pos, apple_pos = state

    # Current distances from agent position
    current_dist_apple = manhattan_distance(agent_pos, apple_pos)
    current_dist_monster = manhattan_distance(agent_pos, monster_pos)

    # New agent position if action is taken
```

```python
    new_agent_pos = move(agent_pos, action)

    # New distances from new agent position
    new_dist_apple = manhattan_distance(new_agent_pos, apple_pos)
    new_dist_monster = manhattan_distance(new_agent_pos, monster_pos)

    f1 = 1 / (new_dist_apple + 1)
    f2 = 1 / (new_dist_monster + 1)
    f3 = 1 if new_dist_apple < current_dist_apple else 0
    f4 = 1 if new_dist_monster < current_dist_monster else 0

    return np.array([f1, f2, f3, f4])

def state_feature(state):
    """
    Returns a 4-dim state feature vector.
    Here we use the differences between the agent and the apple/monster.
    """
    agent_pos, monster_pos, apple_pos = state
    return np.array([apple_pos[0] - agent_pos[0],
                     apple_pos[1] - agent_pos[1],
                     monster_pos[0] - agent_pos[0],
                     monster_pos[1] - agent_pos[1]])

# ---------------------------
# n-step SARSA (using action_state_feature)
# ---------------------------
def epsilon_greedy_theta(theta, state, epsilon):
    """
    Choose an action using the epsilon-greedy policy based on the linear approximator.
    Here, theta is a vector and we use action_state_feature.
    """
    if np.random.rand() < epsilon:
        return random.choice(actions)
    q_vals = [np.dot(theta, action_state_feature(state, a)) for a in actions]
    max_q = max(q_vals)
    best_actions = [a for a, q in zip(actions, q_vals) if np.isclose(q, max_q)]
    return random.choice(best_actions)

def n_step_sarsa(n, episodes=1000, alpha=0.001, gamma=0.95):
    """
    Implements semi-gradient n-step SARSA with a linear function approximator using action_state_feature.
    Theta is a 4-dim vector.

    Returns:
      theta: learned parameter vector.
      episode_rewards: list of total returns per episode.
    """
    theta = np.zeros(4)
    episode_rewards = []

    for ep in range(episodes):
        epsilon = get_epsilon(ep)
        state = initial_positions()   # (agent, monster, apple)
        action = epsilon_greedy_theta(theta, state, epsilon)

        states = [state]
```

```
        actions_list = [action]
        rewards = [0]  # dummy for indexing

        T_episode = float('inf')
        t = 0
        total_reward = 0
        max_steps = T

        while True:
            if t < T_episode:
                monster_action = random.choice(actions)
                next_state, reward, done = step(states[t], actions_list[t], monster_
    action)
                rewards.append(reward)
                total_reward += reward

                if done:
                    T_episode = t + 1
                else:
                    if t == max_steps - 1:
                        T_episode = t + 1
                    else:
                        next_action = epsilon_greedy_theta(theta, next_state, epsilon)
                        states.append(next_state)
                        actions_list.append(next_action)

            tau = t - n + 1
            if tau >= 0:
                G = 0.0
                for i in range(tau + 1, min(tau + n, T_episode) + 1):
                    G += (gamma ** (i - tau - 1)) * rewards[i]
                if tau + n < T_episode:
                    feat = action_state_feature(states[tau+n], actions_list[tau+n])
                    G += (gamma ** n) * np.dot(theta, feat)
                feat_tau = action_state_feature(states[tau], actions_list[tau])
                q_val = np.dot(theta, feat_tau)
                theta += alpha * (G - q_val) * feat_tau

            if tau == T_episode - 1:
                break
            t += 1

        episode_rewards.append(total_reward)

    return theta, episode_rewards

# ---------------------------
# REINFORCE and REINFORCE with Baseline (using state_feature for policy, but using
    action_state_feature for baseline)
# ---------------------------
# For the policy, we use the softmax over h(s,a) = theta[:, a]^T state_feature(s).
def softmax_policy_reinforce(theta, state):
    """
    Computes the softmax policy using state features.
    Theta is a (4 x |actions|) matrix.

    Returns:
      policy: dictionary mapping actions to probabilities.
```

```
      phi: the state feature vector for state.
    """
    phi = state_feature(state)
    scores = np.array([np.dot(theta[:, i], phi) for i, a in enumerate(actions)])
    max_score = np.max(scores)
    exp_scores = np.exp(scores - max_score)
    probs = exp_scores / np.sum(exp_scores)
    policy = {a: probs[i] for i, a in enumerate(actions)}
    return policy, phi

def sample_action_reinforce(theta, state):
    """Sample an action from the softmax policy (using state features)."""
    policy, phi = softmax_policy_reinforce(theta, state)
    chosen_action = np.random.choice(actions, p=[policy[a] for a in actions])
    return chosen_action, policy, phi

def compute_returns(rewards, gamma):
    """
    Given a list of rewards (with a dummy at index 0), compute returns for each time step.
    """
    G = 0
    returns = []
    for r in rewards[::-1]:
        G = r + gamma * G
        returns.insert(0, G)
    return returns[1:]  # skip dummy

def reinforce(episodes=1000, alpha=0.001, gamma=0.95):
    """
    Implements the REINFORCE algorithm using state features for the policy.
    Theta is a (4 x |actions|) matrix.

    Returns:
      theta: learned policy parameters.
      episode_rewards: list of total returns per episode.
    """
    theta = np.zeros((4, len(actions)))
    episode_rewards = []

    for ep in range(episodes):
        state = initial_positions()
        trajectory = []  # list of (state, action, reward)
        t = 0
        done = False

        while not done and t < T:
            action, policy, phi = sample_action_reinforce(theta, state)
            monster_action = random.choice(actions)
            next_state, reward, done = step(state, action, monster_action)
            trajectory.append((state, action, reward))
            state = next_state
            t += 1

        rewards_ep = [0] + [r for (_,_,r) in trajectory]
        returns = compute_returns(rewards_ep, gamma)

        for t_step, (state_t, action_t, _) in enumerate(trajectory):
```

```
            policy_t, phi = softmax_policy_reinforce(theta, state_t)
            # Update for each action in policy:
            for i, a in enumerate(actions):
                indicator = 1 if a == action_t else 0
                grad = (indicator - policy_t[a]) * phi
                theta[:, i] += alpha * returns[t_step] * grad
        episode_rewards.append(sum([r for (_,_,r) in trajectory]))

    return theta, episode_rewards

def reinforce_with_baseline(episodes=1000, alpha=0.001, beta=0.001, gamma=0.95):
    """
    Implements the REINFORCE algorithm with a state-value baseline.

    The policy is parameterized using state features as before, but the baseline is now approximated as
      v(s) = max_a [ w^T x(s,a) ],
    where x(s,a) = action_state_feature(s,a) and w is a 4-dim vector.

    Returns:
      theta: learned policy parameter matrix.
      w: learned baseline parameter vector.
      episode_rewards: list of total returns per episode.
    """
    theta = np.zeros((4, len(actions)))  # policy parameters (for softmax using state features)
    w = np.zeros(4)  # baseline parameters (using action_state_feature)
    episode_rewards = []

    for ep in range(episodes):
        state = initial_positions()
        trajectory = []
        t = 0
        done = False

        while not done and t < T:
            action, policy, phi = sample_action_reinforce(theta, state)
            monster_action = random.choice(actions)
            next_state, reward, done = step(state, action, monster_action)
            trajectory.append((state, action, reward))
            state = next_state
            t += 1

        rewards_ep = [0] + [r for (_,_,r) in trajectory]
        returns = compute_returns(rewards_ep, gamma)

        for t_step, (state_t, action_t, _) in enumerate(trajectory):
            # Compute baseline v(s) = max_a [ w^T x(s,a) ] using action_state_feature
            q_values = [np.dot(w, action_state_feature(state_t, a)) for a in actions]
            v_s = max(q_values)
            # Determine the maximizing action a* (using argmax)
            a_star = actions[np.argmax(q_values)]
            phi_baseline = action_state_feature(state_t, a_star)
            A_t = returns[t_step] - v_s  # advantage

            # Policy update (remains as before, using state features)
            policy_t, phi_policy = softmax_policy_reinforce(theta, state_t)
            for i, a in enumerate(actions):
```

```
                indicator = 1 if a == action_t else 0
                grad = (indicator - policy_t[a]) * phi_policy
                theta[:, i] += alpha * A_t * grad

            # Baseline update using the feature vector corresponding to a*
            w += beta * A_t * phi_baseline

        episode_rewards.append(sum([r for (_,_,r) in trajectory]))

    return theta, w, episode_rewards


# ---------------------------
# Softmax Policy for Actor-Critic (using action_state_feature)
# ---------------------------
def softmax_policy_actor_critic(theta, state):
    """
    Computes the softmax policy for a given state using action_state_feature.

    Parameters:
      theta: actor parameter vector (4-dim)
      state: current state

    Returns:
      policy: a dictionary mapping each action to its probability.
    """
    scores = []
    for a in actions:
        # The logit for action a
        score = np.dot(theta, action_state_feature(state, a))
        scores.append(score)
    # Numerical stability: subtract max score
    max_score = np.max(scores)
    exp_scores = [np.exp(s - max_score) for s in scores]
    sum_exp = np.sum(exp_scores)
    probs = [s/sum_exp for s in exp_scores]
    policy = {a: probs[i] for i, a in enumerate(actions)}
    return policy

# ---------------------------
# One-Step Actor-Critic Algorithm
# ---------------------------
def one_step_actor_critic(episodes=15000, alpha=0.01, beta=0.01, gamma=0.95):
    """
    Implements the one-step actor-critic algorithm.

    The policy is defined as:
        (a|s;  ) = exp(  ^T x(s,a)) /    _b exp(  ^T x(s,b))
    with x(s,a)= action_state_feature(s,a).

    The critic estimates the state value as:
      V(s) = max_a [w^T x(s,a)]
    and is updated using the feature vector corresponding to the maximizing action.

    Updates:
         = r +    V(s') - V(s)
      Actor:            +       [ x(s,a) -    _b   (b|s;  ) x(s,b) ]
      Critic: w     w +       x(s,a*),   where a* = argmax_a [w^T x(s,a)]
```

```
    Returns:
      theta: learned actor parameter vector (4-dim)
      w: learned critic parameter vector (4-dim)
      episode_rewards: list of total return per episode.
    """
    # Initialize actor and critic parameters as zero vectors
    theta = np.zeros(4)
    w = np.zeros(4)
    episode_rewards = []

    for ep in range(episodes):
        state = initial_positions()
        done = False
        total_reward = 0
        t = 0
        while not done and t < T:
            # Select action using the softmax policy
            policy = softmax_policy_actor_critic(theta, state)
            a = np.random.choice(actions, p=[policy[a] for a in actions])

            # Execute action; monster acts randomly
            monster_action = random.choice(actions)
            next_state, reward, done = step(state, a, monster_action)
            total_reward += reward

            # Critic: compute V(s) = max_a [w^T x(s,a)]
            q_values_state = [np.dot(w, action_state_feature(state, b)) for b in actions]
            V_s = max(q_values_state)
            # For terminal state, set V(s') = 0
            if done:
                V_next = 0.0
            else:
                q_values_next = [np.dot(w, action_state_feature(next_state, b)) for b in actions]
                V_next = max(q_values_next)

            # TD error
            delta = reward + gamma * V_next - V_s

            # Actor update:
            # Compute gradient:     log    (a|s) = x(s,a) -    _b    (b|s) x(s,b)
            grad_log = action_state_feature(state, a)
            for b in actions:
                grad_log -= policy[b] * action_state_feature(state, b)
            theta = theta + alpha * delta * grad_log

            # Critic update:
            # Find the action that maximizes w^T x(s,a)
            a_star = actions[np.argmax(q_values_state)]
            phi_star = action_state_feature(state, a_star)
            w = w + beta * delta * phi_star

            state = next_state
            t += 1

        episode_rewards.append(total_reward)
```

```
        if (ep+1) % 1000 == 0:
            print("Episode {}: Total Reward = {}".format(ep+1, total_reward))

    return theta, w, episode_rewards

# %%
# ---------------------------
# Utility: Rolling Average Function
# ---------------------------
def rolling_average(data, window_size):
    return np.convolve(data, np.ones(window_size)/window_size, mode='valid')

# ---------------------------
# Run All Methods
# ---------------------------
# n-step SARSA for n=1,2,3 (using action_state_feature)
theta_1, rewards_1 = n_step_sarsa(n=1, episodes=episodes, alpha=alpha, gamma=gamma)
theta_2, rewards_2 = n_step_sarsa(n=2, episodes=episodes, alpha=alpha, gamma=gamma)
theta_3, rewards_3 = n_step_sarsa(n=3, episodes=episodes, alpha=alpha, gamma=gamma)

#%%
# REINFORCE and REINFORCE with Baseline (policy uses state_feature; baseline uses action_state_feature)
theta_reinf, rewards_reinf = reinforce(episodes=episodes, alpha=alpha, gamma=gamma)
theta_rb, w_rb, rewards_rb = reinforce_with_baseline(episodes=episodes, alpha=0.0, beta=0.05, gamma=gamma)

#%%
# actor-critic
theta_ac, w_ac, rewards_ac = one_step_actor_critic(episodes=episodes, alpha=alpha, beta=alpha, gamma=gamma)

# %%
# Compute rolling averages (using a window size appropriate for episodes)
window = 500
roll_rewards_1 = rolling_average(rewards_1, window)
roll_rewards_2 = rolling_average(rewards_2, window)
roll_rewards_3 = rolling_average(rewards_3, window)
roll_reinf = rolling_average(rewards_reinf, window)
roll_rb = rolling_average(rewards_rb, window)
roll_rewards_ac = rolling_average(rewards_ac, window)

# ---------------------------
# Plot All Methods Together
# ---------------------------
plt.figure(figsize=(10,6))
plt.plot(roll_rewards_1, label='n-step SARSA (n=1)')
plt.plot(roll_rewards_2, label='n-step SARSA (n=2)')
plt.plot(roll_rewards_3, label='n-step SARSA (n=3)')
plt.plot(roll_reinf, label='REINFORCE')
plt.plot(roll_rb, label='REINFORCE with Baseline')
plt.plot(roll_rewards_ac, label='One-Step Actor-Critic')
plt.xlabel('Episode')
plt.ylabel('Rolling Average Reward per (last {} episodes)'.format(window))
plt.title('Comparison of RL Methods on Gridworld')
plt.legend()
plt.show()
```

```
# %%
```

References

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