Reinforcement Learning - Assignment 2

Marco Turchetti
Student ID: 1996734

Theory

Exercise 1: Q-learning and SARSA Updates

Problem Statement

Given the following Q-table:

$$Q(s,a) = \begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix} = \begin{pmatrix} Q(1,1) & Q(1,2) \\ Q(2,1) & Q(2,2) \end{pmatrix}$$

Parameters: $\alpha = 0.3$, $\gamma = 0.5$. Experience tuple: $(s, a, r, s') = (1, 1, 4, 2)$.

The task is to compute the update for both Q-learning and SARSA. For SARSA, the next action is given as $a' = \pi(s') = 1$.

1. Q-Learning Update

The update rule for Q-learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Substituting the values $s = 1, a = 1, r = 4, s' = 2$:

$$\begin{aligned} Q(1,1) &\leftarrow 2 + 0.3[4 + 0.5 \cdot \max\{Q(2,1), Q(2,2)\} - 2] \\ &\leftarrow 2 + 0.3[4 + 0.5 \cdot \max(5,3) - 2] \\ &\leftarrow 2 + 0.3[4 + 0.5 \cdot 5 - 2] \\ &\leftarrow 2 + 0.3[4 + 2.5 - 2] \\ &\leftarrow 2 + 0.3[4.5] \\ &\leftarrow 2 + 1.35 \\ \mathbf{Q(1,1)} &\leftarrow \mathbf{3.35} \end{aligned}$$

2. SARSA Update

The update rule for SARSA is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

Given the next action $a' = 1$, the value $Q(s', a') = Q(2, 1) = 5$ is used:

$$\begin{aligned} Q(1,1) &\leftarrow 2 + 0.3[4 + 0.5 \cdot Q(2,1) - 2] \\ &\leftarrow 2 + 0.3[4 + 0.5 \cdot 5 - 2] \\ &\leftarrow 2 + 0.3[4 + 2.5 - 2] \\ &\leftarrow 2 + 0.3[4.5] \\ &\leftarrow 2 + 1.35 \\ \mathbf{Q(1,1)} &\leftarrow \mathbf{3.35} \end{aligned}$$

Observation

In this specific scenario, both Q-learning and SARSA yield the identical updated value of **3.35** for $Q(1, 1)$. This coincidence arises because the next action selected for SARSA ($a' = 1$) happens to be the greedy action in the next state $s' = 2$ (since $Q(2, 1) = 5 > Q(2, 2) = 3$). Consequently, the SARSA update term, $r + \gamma Q(s', a')$, reduces to the same quantity used by Q-learning, $r + \gamma \max_{a'} Q(s', a')$.

Exercise 2: Proof of n-step Error

The goal is to show that the n-step error is equal to the sum of discounted TD errors, assuming the value estimates are static. The equation to prove is:

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k \tag{1}$$

The n-step return $G_{t:t+n}$ is defined as:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

First, let's write the TD error δ_k formula:

$$\delta_k = R_{k+1} + \gamma V_k(S_{k+1}) - V_k(S_k)$$

The exercise assumes that value estimates don't change from step to step during the calculation. So, we can assume $V_t(s) = V_{t+1}(s) = \cdots = V_{t+n-1}(s)$. To keep the notation consistent with the n-step return formula, V_{t+n-1} is used to indicate the value function.

The TD error becomes:

$$\delta_k = R_{k+1} + \gamma V_{t+n-1}(S_{k+1}) - V_{t+n-1}(S_k) \tag{2}$$

Now, let's expand the sum $\sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k$. To make the cancellation clear, let's multiply the γ factors into the brackets:

$$\begin{aligned} \sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k = {} & [R_{t+1} + \cancel{\gamma V_{t+n-1}(S_{t+1})} - V_{t+n-1}(S_t)] \quad (\text{for } k = t) \\ & + [\gamma R_{t+2} + \cancel{\gamma^2 V_{t+n-1}(S_{t+2})} - \cancel{\gamma V_{t+n-1}(S_{t+1})}] \quad (\text{for } k = t+1) \\ & + [\gamma^2 R_{t+3} + \cancel{\gamma^3 V_{t+n-1}(S_{t+3})} - \cancel{\gamma^2 V_{t+n-1}(S_{t+2})}] \quad (\text{for } k = t+2) \\ & \cdots \\ & + [\gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) - \cancel{\gamma^{n-1} V_{t+n-1}(S_{t+n-1})}] \quad (\text{for } k = t+n-1) \end{aligned}$$

As shown by the crossed-out terms, this is a telescoping sum. The positive value term from one step cancels exactly with the negative value term from the next step.

The only terms that remain are:

1. The initial negative value: $-V_{t+n-1}(S_t)$
2. The final positive value: $+\gamma^n V_{t+n-1}(S_{t+n})$
3. The sum of discounted rewards: $R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n}$

Grouping these together:

$$\sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k = \underbrace{(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n}) + \gamma^n V_{t+n-1}(S_{t+n})}_{G_{t:t+n}} - V_{t+n-1}(S_t)$$

Substituting $G_{t:t+n}$ back into the expression gives the final result:

$$\boxed{\sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k = G_{t:t+n} - V_{t+n-1}(S_t)} \tag{3}$$

Practice

Exercise 1: Implementation of SARSA(λ) on Taxi environment

The objective of this exercise is to implement the SARSA(λ) algorithm. This is an on-policy Temporal Difference (TD) control method that incorporates Eligibility Traces (E) to handle the credit assignment problem more efficiently than one-step TD methods. The algorithm updates the Q-values based on the current action selection policy.

Action Selection Strategy

The implementation utilizes an ϵ-greedy strategy for action selection, balancing exploration and exploitation. A helper function is defined to select an action based on the current Q-values ($Q(s, a)$) and the exploration rate ϵ.

With probability $1-\epsilon$, the agent exploits the current knowledge by choosing the action that maximizes the Q-value:

$$a = \arg\max_a Q(s, a)$$

With probability ϵ, the agent explores by selecting a random action from the environment's action space.

```python
def epsilon_greedy_action(env, Q, state, epsilon):
    if np.random.rand() < 1 - epsilon:
        # Exploitation: choose the best known action
        action = np.argmax(Q[state])
    else:
        # Exploration: choose a random action
        action = env.action_space.sample()
    return action
```

Initialization and Main Loop

The main algorithm initializes the Q-table as a matrix of zeros with dimensions corresponding to the observation and action spaces. The training process iterates through a fixed number of episodes defined by 'n_episodes'.

A critical step in this implementation is the initialization of the Eligibility Traces matrix (E). Unlike the Q-table, E must be reset to zero at the beginning of *every* episode to ensure that traces from previous trajectories do not influence the current episode.

```python
# Initialize Q-table globally
Q = np.zeros((env.observation_space.n, env.action_space.n))

for ep in tqdm(range(n_episodes)):
    # Reset Eligibility Traces at the start of each episode
    E = np.zeros((env.observation_space.n, env.action_space.n))

    state, _ = env.reset()
    action = epsilon_greedy_action(env, Q, state, epsilon)
```

The SARSA Update Rule

Inside the episode loop, the agent interacts with the environment. For every step, the agent observes the reward (R) and the next state (S'), and selects the next action (A') using the same ϵ-greedy policy.

The algorithm then computes the TD error (δ), which represents the difference between the current Q-estimate and the target return:

$$\delta = R + \gamma Q(S', A') - Q(S, A)$$

Simultaneously, the eligibility trace for the currently visited state-action pair is incremented (Accumulating Traces):

$$E(S, A) \leftarrow E(S, A) + 1$$

```
# Simulate the action
next_state, reward, terminated, truncated, _ = env.step(action)
next_action = epsilon_greedy_action(env, Q, next_state, epsilon)

# Calculate TD Error
delta = reward + gamma * Q[next_state, next_action] - Q[state, action]

# Update Eligibility Trace for the current state-action pair
E[state, action] = E[state, action] + 1
```

Vectorized Value Update

Once δ and E are updated, the algorithm updates the Q-values for all state-action pairs. The update rule combines the learning rate α, the TD error, and the eligibility trace:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$$

$$E(s, a) \leftarrow \gamma \lambda E(s, a)$$

The implementation utilizes NumPy vectorization to perform this update across the entire Q-table and E-table simultaneously. This is significantly more efficient than iterating through states and actions with nested loops.

```
# Vectorized update of Q-values and Decay of Eligibility Traces
Q += alpha * delta * E
E *= gamma * lambda_
```

Epsilon Decay

To allow the agent to converge to an optimal policy, the exploration rate ϵ is decayed over time. The decay process begins only after the agent receives its first positive reward, ensuring that sufficient exploration occurs before the agent starts exploiting a potentially sub-optimal path.

```
if not received_first_reward and reward > 0:
    received_first_reward = True

# ... inside the loop ...
if received_first_reward:
    epsilon = 0.99 * epsilon
```

Exercise 2: Mountain Car Simulation using Q-Learning TD(λ) with RBF Approximation

The objective of this exercise was to implement and optimize a Reinforcement Learning agent to solve the *Mountain Car* environment [4]. This classic control problem involves an underpowered car positioned in a valley. The state space is continuous, consisting of two variables: position (x) and velocity (v). The goal is to reach the flag at the top of the right hill. Since the car's engine is not strong enough to climb the slope directly, the agent must learn to oscillate back and forth to build sufficient momentum.

The agent receives a reward of -1 for every time step until the goal is reached. The episode is capped at 200 time steps; therefore, if the car fails to reach the goal within this limit, the episode terminates with a cumulative reward of -200 [7]. Efficient time management is critical, as the agent's primary objective is to solve the task in strictly fewer than 200 steps to achieve a score better than the failure threshold. According to the environment benchmarks, a score around -110.0 is often considered "solved" [6].

Feature Engineering: The RBF Sampler

The Necessity of RBFs in Continuous Spaces

The assignment mandates the use of Radial Basis Functions (RBFs) to handle the continuous state space of the Mountain Car environment. Unlike tabular methods, which require discretization and suffer from the "curse of dimensionality", function approximation allows for generalization across continuous states [8]. Specifically, linear function approximation is employed, where the value function is represented as a weighted sum of features generated by the RBF kernels [9].

State Standardization

Prior to passing state vectors into the RBF sampler, a standardization step was implemented using *sklearn.preprocessing.StandardScaler*. Since the Mountain Car state variables—position and velocity—operate on significantly different scales, unscaled inputs would cause the RBF kernels to be disproportionately sensitive to the feature with the larger magnitude. To mitigate this, the scaler was fit using 5,000 random observations sampled from the environment.

From Single-Kernel to Multi-Resolution Kernels

The RBF kernel parameter γ determines the width (inverse variance) of the Gaussian basis functions. A higher γ corresponds to a narrower kernel, providing high precision but low generalization, while a lower γ results in a broader kernel, offering high generalization but low precision. Initially, a single kernel width was tested, but it yielded suboptimal performance with rewards stagnating around -200.

To address the trade-off between precision and generalization, a **Multi-Resolution RBF Encoder** was implemented using the `RBFSampler` from the *sklearn* library [5]. Instead of selecting a single γ, feature maps generated from a range of values ($\gamma \in [0.2, 0.5, 1.0, 2.0]$) are concatenated. This strategy, inspired by Coarse Coding [1], allows the agent to capture global trends, such as position on the slope, via low γ kernels, while simultaneously retaining the ability to make precise adjustments near the goal via high γ kernels. This is consistent with Multiple Kernel Learning (MKL) principles [3].

The implementation of the multi-kernel initialization and encoding is shown below:

```
class RBFFeatureEncoder:
    def __init__(self, env):
        self.env = env
        # Fit the RBF sampler to sample data
```

```
        observation_examples = np.array([env.observation_space.sample() for i in range(5000)])
        self.scaler = StandardScaler()
        self.scaler.fit(observation_examples)
        scaled_examples = self.scaler.transform(observation_examples)

        self.samplers = []
        gammas = [0.2, 0.5, 1.0]
        n_components_per_gamma = 50

        # Initialize Multiple Kernels
        for g in gammas:
            # Using sklearn.kernel_approximation.RBFSampler
            sampler = RBFSampler(n_components=n_components_per_gamma, gamma=g, random_state=1)
            sampler.fit(scaled_examples)
            self.samplers.append(sampler)

        self._size = len(gammas) * n_components_per_gamma

    def encode(self, state):
        state_2d = np.array(state).reshape(1, -1)
        scaled_state = self.scaler.transform(state_2d)
        # Concatenate features from all samplers
        features_list = [sampler.transform(scaled_state) for sampler in self.samplers]
        return np.concatenate(features_list, axis=1).flatten()
```

Listing 1: Multi-Resolution RBF Encoder Implementation

The Learning Algorithm: TD(λ)

Update Rule Derivation

The learning algorithm employed is Q-Learning TD(λ) with linear function approximation. The goal is to minimize the Mean Squared Error (MSE) between the predicted action-value $Q(s, a; w)$ and the true target value f^*. The loss function is defined as:

$$J(w) = \frac{1}{2}\mathbb{E}\left[(f^* - Q(s, a; w))^2\right] \tag{4}$$

Using Stochastic Gradient Descent (SGD), the weight update is derived by taking the gradient of the loss function with respect to the weights w:

$$w_{t+1} = w_t + \alpha(f^* - Q(s, a; w_t))\nabla_w Q(s, a; w_t) \tag{5}$$

In the context of linear function approximation, $Q(s, a; w) = w^T\phi(s, a)$, so the gradient is simply the feature vector $\nabla_w Q(s, a; w) = \phi(s, a)$. Substituting the TD($\lambda$) target for f^* introduces the concept of eligibility traces e_t, which accumulate gradients over time [10]. The term $(f^* - Q(s, a; w_t))$ is approximated by the TD error δ_t:

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q(s', a'; w_t) - Q(s, a; w_t) \tag{6}$$

Combining these yields the final update rule used in the implementation:

$$w_{t+1} = w_t + \alpha\delta_t e_t \tag{7}$$

The implementation of the update logic is shown below:

```
    def update_transition(self, s, action, s_prime, reward, done):
        s_feats = self.feature_encoder.encode(s)
        s_prime_feats = self.feature_encoder.encode(s_prime)

        next_q_values = self.Q(s_prime_feats)
        current_q = self.Q(s_feats)[action]

        # Calculate TD Error (Delta)
        target = reward + self.gamma * np.max(next_q_values) * (1 - done)
        delta = target - current_q

        # Update Eligibility Traces
        self.traces *= self.gamma * self.lambda_
        self.traces[action] += s_feats

        # Update Weights
        self.weights[action] += self.alpha * delta * self.traces[action]
```

Listing 2: TD Lambda Update with Replacing Traces

Results

With the multi-resolution RBF encoder, the agent's mean reward consistently oscillates between **-105 and -150**. While the strict threshold for "solving" the environment is often cited as -110 over 100 trials, this performance range indicates a robust policy where the car successfully reaches the goal in every episode. The oscillation reflects minor variations in the trajectory efficiency, likely due to the stochastic nature of the function approximation and the remaining exploration noise.

Bibliography

References

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. (Chapters 9.5.4 and 9.5.5).

[2] Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3), 123-158. https://doi.org/10.1007/BF00114726

[3] Gönen, M., & Alpaydın, E. (2011). Multiple kernel learning algorithms. *Journal of Machine Learning Research*, 12, 2211-2268.

[4] Gymnasium Documentation. *Mountain Car - Classic Control*. https://mgoulao.github.io/gym-docs/environments/classic_control/mountain_car/

[5] Scikit-learn Documentation. *sklearn.kernel_approximation.RBFSampler*. https://scikit-learn.org/stable/modules/generated/sklearn.kernel_approximation.RBFSampler.html

[6] OpenAI Gym Wiki. *Leaderboard*. https://github.com/openai/gym/wiki/Leaderboard

[7] Towards Data Science. *Reinforcement Learning Applied to the Mountain Car Problem*. https://medium.com/data-science/reinforcement-learning-applied-to-the-mountain-car-problem-1c4fb16729ba

[8] Towards Data Science. *Reinforcement Learning Part 8: Feature State Construction*. https://towardsdatascience.com/reinforcement-learning-part-8-feature-state-construction-62e7d2fc5152/

[9] Towards Data Science. *Kernel Methods: A Simple Introduction*. https://towardsdatascience.com/kernel-methods-a-simple-introduction-4a26dcbe4ebd/

[10] Sutton, R. S. *Eligibility Traces (Online Book Node 80)*. http://incompleteideas.net/book/ebook/node80.html