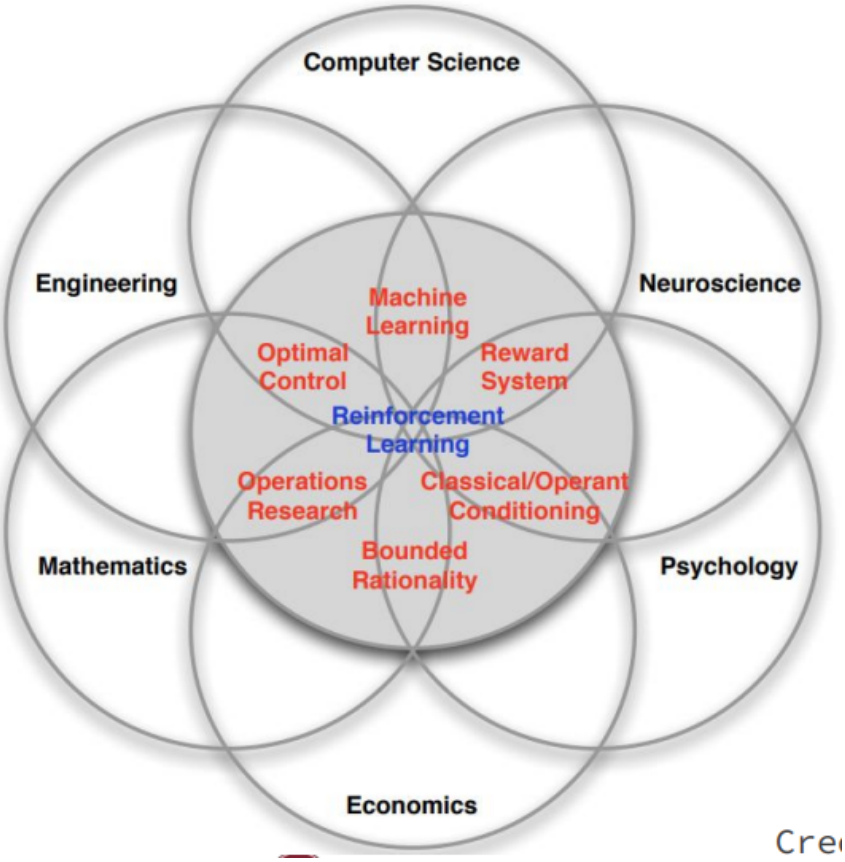


Reinforcement Learning

Assignment 2

Yusupha Juwara
juwara.1936515@studenti.uniroma1.it

November 24, 2023

Contents

Chapter 1

Theory Part

Question 1.1: Given the following Q table:

$$Q(s,a) = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} Q(1,1) & Q(1,2) \\ Q(2,1) & Q(2,2) \end{pmatrix}$$

Assume that $\alpha = 0.1, \gamma = 0.5$, after an experience (s,a,r,s') = (1,2,3,2) compute the update of both Q-learning and SARSA. For the latter consider $a' = \pi_\epsilon(s') = 2$

Answer 1.1: -

1. **Sarsa:**

$$\begin{aligned} Q(s_t,a_t) &= Q(s_t,a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \\ &= Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right] \\ &= Q(s=1,a=2) + \alpha[r + \gamma Q(s'=2,a'=2) - Q(s=1,a=2)] \\ &= 2 + 0.1\left[3 + 0.5 \times 4 - 2\right] = 2 + 0.1 \times 3 = 2.3 \\ \therefore Q(s=1,a=1) &= 2.3 \end{aligned}$$

2. **Q-Learning:**

$$\begin{aligned} Q(s_t,a_t) &= Q(s_t,a_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)] \\ &= Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \\ &= Q(s=1,a=2) + \alpha[r + \gamma \max_{a'} Q(s'=2,a') - Q(s=1,a=2)] \\ &= 2 + 0.1\left[3 + 0.5 \times \max(3, \boxed{4}) - 2\right] = 2 + 0.1 \times (3 + 0.5 \times 4 - 2) = 2.3 \\ \therefore Q(s=1,a=1) &= 2.3 \end{aligned}$$

Question 1.2: Prove that the n-step error can also be written as a sum of TD errors if the value estimates don't change from step to step, i.e.,

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) &= \sum_{k=t}^{t+n-1} \gamma^{k-t}\delta_k \\ \text{where } G_{t:t+n} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} \gamma^n V_{t+n-1}(S_{t+n}) \end{aligned}$$

Answer 1.2: Note that this solution is taken from this github page.

- Recall, $\delta_t = R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)$
- Update of n-step TD is

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)]$$

Derivation is as follows:

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

since we assume V will not change, we have

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

Recall that $\delta_z = R_{z+1} + \gamma V_z(S_{z+1}) - V_z(S_z) \implies R_{z+1} = \delta_z - \gamma V(S_{z+1}) + V(S_z)$

$$\begin{aligned} G_{t:t+n} - V_{t+n-1}(S_t) &= \delta_t - \gamma V(S_{t+1}) + V(S_t) + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) - V(S_t) \\ &= \gamma^0[\delta_t - \gamma V(S_{t+1}) + V(S_t)] \\ &\quad + \gamma^1[\delta_{t+1} - \gamma V(S_{t+2}) + V(S_{t+1})] \\ &\quad + \cdots \\ &\quad + \gamma^{n-1}[\delta_{t+n-1} - \gamma V(S_{t+n}) + V(S_{t+n-1})] \\ &\quad + \gamma^n V(S_{t+n}) - V(S_t) \end{aligned}$$

Group the sums into summation from t to t+n-1

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \left[\gamma^{k-t}\delta_k - \gamma^{k-t+1} V(S_{k+1}) + \gamma^{k-t} V(S_k)\right] + \gamma^n V(S_{t+n}) - V(S_t)$$

Notice that the middle summation has an upper bound of t+n-2
while the last summation has a lower bound of t+1.
Expand them to see why so for a relatively small t

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \left[\gamma^{k-t}\delta_k\right] - \sum_{k=t}^{t+n-2} \left[\gamma^{k-t+1} V(S_{k+1})\right] + \sum_{k=t+1}^{t+n-1} \left[\gamma^{k-t} V(S_k)\right]$$

Make the last two summations equal terms so that they cancel each other

$$G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \left[\gamma^{k-t}\delta_k\right] - \sum_{k=t+1}^{t+n-1} \left[\gamma^{k-t} V(S_k)\right] + \sum_{k=t+1}^{t+n-1} \left[\gamma^{k-t} V(S_k)\right]$$

$$\therefore G_{t:t+n} - V_{t+n-1}(S_t) = \sum_{k=t}^{t+n-1} \left[\gamma^{k-t}\delta_k\right]$$

Chapter 2

Code Report Assignment 2

2.1 Introduction

Reinforcement learning (RL) is a type of machine learning that allows an agent to learn optimal behavior in an environment by interacting with it and receiving rewards. One of the most common RL algorithms is Q-learning, which is an off-policy algorithm that learns a state-action value function, Q(s, a), that maps each state-action pair to a value that represents the expected long-term reward of taking that action in that state.

Another popular RL algorithm is Sarsa, which is an on-policy algorithm that learns a state-action value function by directly updating the Q-value of the state and action that was just taken. Both Sarsa and Q-Learning can also be extended to use eligibility traces, which keep track of the states and actions that have been visited recently and give them more weight when updating the Q-value function.

In this report, I will explore two different versions of RL algorithms: Sarsa-λ and Q-Learning TD(λ). I will first introduce the Taxi environment and the Epsilon Greedy strategy. Then, I will discuss the concepts of eligibility traces and Q-table updates. Finally, I will present the Sarsa-λ backward view algorithm.

In the second part of this report, I will focus on Q-Learning TD(λ) for the MountainCar-v0 environment. I will first introduce the MountainCar environment and the Radial Basis Function (RBF). Then, I will discuss the preprocessing and RBF state featurizer. Finally, I will present the Q-Learning TD(λ) algorithm.

2.2 Sarsa-λ

2.2.1 Taxi Environment Overview

The Taxi Problem involves navigating to passengers in a grid world, picking them up, and dropping them off at one of four locations.

2.2.1.1 Action Space

The action shape is (1,) in the range 0, 5 indicating which direction to move the taxi or to pickup/drop off passengers. There are 6 discrete actions:

1. Move south (down)
2. Move north (up)
3. Move east (right)
4. Move west (left)

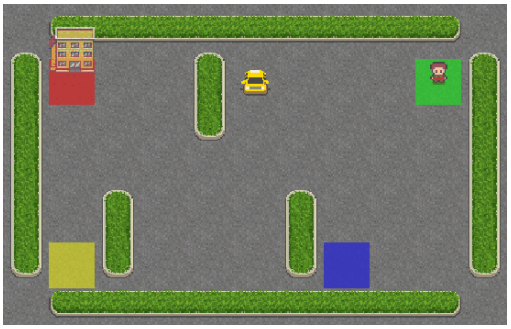


Figure 2.1: Taxi Environment

5. Pickup passenger
6. Drop off passenger

```
Discrete(6)
```

2.2.1.2 Observation Space

```
Discrete(500)
```

Observation Space Details:

- There are 500 discrete states, including 25 taxi positions, 5 possible passenger locations (including in the taxi), and 4 destination locations.
- An observation is returned as an int() that encodes the corresponding state, calculated by ((taxi_row $\times$ 5 + taxi_col) $\times$ 5 + passenger_location) $\times$ 4 + destination

2.2.1.3 Starting State

The episode starts with the player in a random state.

2.2.1.4 Rewards

- -1 per step unless another reward is triggered.
- +20 for delivering the passenger.
- -10 for executing "pickup" and "drop-off" actions illegally.

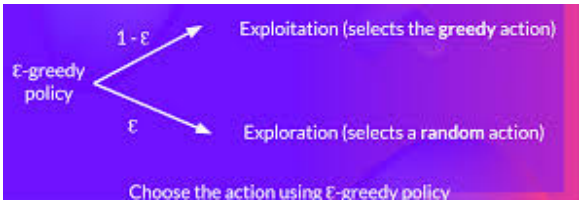
2.2.1.5 Episode End

The episode ends if either:

- Termination: The taxi drops off the passenger.
- Truncation: The length of the episode is 200.

2.2.2 Epsilon Greedy

Epsilon greedy strategy handles **exploration** and **exploitation** tradeoffs. With the probability ϵ, we do exploration, and with a Probability of $1-\epsilon$, we do exploitation.



I implemented this in the assignment as thus:

```python
def greedy_policy(Qtable, state):
  # Exploitation: take the action with the highest state, action value
  action = np.argmax(Qtable[state][:])

  return action

def epsilon_greedy_action(env, Q, state, epsilon):
    # TODO choose the action with epsilon-greedy strategy

    # Randomly generate a number between 0 and 1
    random_num = random.uniform(0,1)
    # if random_num > epsilon --> exploitation

    if random_num > epsilon:
        # Take the action with the highest value given a state
        action = greedy_policy(Q, state)

    # else --> exploration
    else:
        # Randomly sample an action from the action space
        action = env.action_space.sample()

    return action
```

2.2.3 Q Table and Eligibility Traces Updates

Eligibility Traces: instead of waiting for what is going to happen next, we remember what happened in the past just like a credit assignment problem.

- The update for the Eligibility Traces
- For all s, a:

$$e_0(s,a) = 0$$
$$e_t(s,a) = \gamma\lambda e_{t-1}(s,a) + I(s_t = s, a_t = a)$$
where $\gamma =$ recency
$I(s_t = s, a_t = a) =$ frequency
$\lambda \in [0,1]$ trace-decay parameter

- The update for the δ and Q
- Update for all s, a every time:

$$\delta_t = r_t + \gamma Q(S_{t+1}, a \sim \pi(S_{t+1})) - Q(S_t, a_t)$$
$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t e_t(s,a)$$

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S,A) \leftarrow E(S,A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 2.2: Sarsa-λ Backward View algorithm

Recall that at every new episode, the **Eligibility Traces** array has to be zero-ed

```python
...
for ep in tqdm(range(n_episodes)):
    ...
    # Initialize the Eligibility traces to all zero
    E = np.zeros((env.observation_space.n, env.action_space.n))
    ...
...
```

The code snippet for the updates are as thus:

```python
...
 while not done:
    ...
    done = terminated or truncated

    # TODO update q table and eligibility

    # Check the slides about how to update these!
    if done:
        delta = reward - Q[state, action]
    else:
        reward + gamma*Q[next_state, next_action] - Q[state, action]

    E[state, action] += 1
    Q[:,:] = Q[:,:] + alpha*delta*E[:,:]
    E[:,:] = gamma*lambda_ * E[:,:]
    ...
...
```

2.3 Q-Learning TD(λ) for MountainCar-v0

2.3.1 Mountain Car Environment Overview

The Mountain Car MDP is a deterministic MDP with a car placed stochastically at the bottom of a sinusoidal valley. The goal is to strategically accelerate the car to reach the goal state on top of the right hill.

2.3.1.1 Action Space

There are 3 discrete deterministic actions:

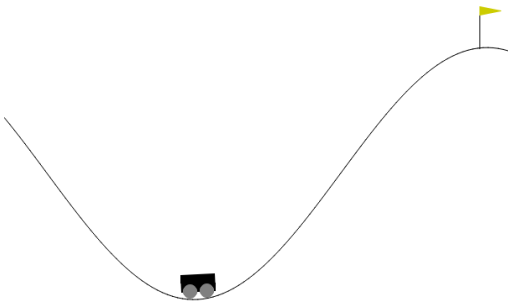


Figure 2.3: Mountain Car Version 0

1. Accelerate to the left
2. Dont accelerate
3. Accelerate to the right

```
Discrete(3)
```

2.3.1.2 Observation Space

```
Box([-1.2 -0.07], [0.6 0.07], (2,), float32)
```

Observation Space Details:

- **Position:** -1.2 to 0.6 (position along the x-axis in meters)
- **Velocity:** -0.07 to 0.07 (velocity in meters per second)

2.3.1.3 Transition Dynamics

Given an action, the mountain car follows the following transition dynamics:

$$\text{vel}_{t+1} = \text{vel}_t + (\text{action} - 1) \times \text{force} - \cos(3 \times \text{pos}_t) \times \text{gravity}$$
$$\text{pos}_{t+1} = \text{pos}_t + \text{vel}_{t+1}$$

where force = 0.001 and gravity = 0.0025. Collisions at either end are inelastic with velocity set to 0 upon collision. Position is clipped to [-1.2, 0.6], and velocity is clipped to [-0.07, 0.07].

2.3.1.4 Reward

The goal is to reach the flag on top of the right hill as quickly as possible, with a penalty of -1 for each timestep.

2.3.1.5 Starting State

Position of the car is assigned a uniform random value in [-0.6, -0.4], and starting velocity is set to 0.

2.3.1.6 Episode End

The episode ends if either:

- Termination: The position of the car is greater than or equal to 0.5 (the goal position on top of the right hill).
- Truncation: Episode length is 200.

2.3.1.7 Source

Mountain Car

2.3.2 Radial Basis Function (RBF)

2.3.2.1 Overview

RBFs are a natural generalization of coarse coding to continuous-valued features. Instead of binary features (0 or 1), RBF features can take values in the interval [0, 1], reflecting different degrees of feature presence. An RBF feature, x_i, exhibits a Gaussian response $x_i(s)$ based on the distance between the state, s, and the feature's center state, c_i, relative to the feature's width:

$$x_i(s) = \exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right)$$

where σ_i represents the width. RBFs produce smooth and differentiable functions.

In this assignment, instead of implementing from scratch the RBF, I utilized the RBFSampler from sk-learn. The RBFSampler is employed to approximate a Radial Basis Function (RBF) kernel feature map using random Fourier features.

2.3.2.2 Preprocessing

In the RBFFeatureEncoder class, the data is preprocessed before applying RBFs. This standardizes the data by subtracting the mean and dividing by the standard deviation (scaled). The transformed states are then used to initialize the RBF state featurizer.

```
# Standardize the data
self.s_mean = samples.mean(axis=0, keepdims=True)
self.s_std = samples.std(axis=0, keepdims=True)
self.transformed_states = (samples - self.s_mean)
    / self.s_sted
```

2.3.2.3 RBF State Featurizer

The RBFFeatureEncoder initializes an RBF state featurizer using different RBF kernels with varying variances to cover different parts of the space. The number of Monte Carlo samples per original feature determines the dimensionality of the computed feature space.

```
num_rbfs = 10
self.rbf_state_featurizer = sklearn.pipeline.
    FeatureUnion([
        ("rbf1", RBFSampler(gamma=5.0,
    n_components=self.n_components//num_rbfs,
    random_state=random_state)),
        # ... (similar lines for other RBFSamplers
    )
        ("rbf10", RBFSampler(gamma=3.0,
    n_components=self.n_components//num_rbfs,
    random_state=random_state))
    ])

    self.rbf_state_featurizer.fit(self.
    transformed_states)
```

However, the formulae used by sk-learn as shown on their github is slightly different from the one above (given by the book and prof's slide). Also, the *gamma* can be a string called "scale" or a float that has a default value of 1.0.

```
gamma : 'scale' or float, default=1.0
        Parameter of RBF kernel: exp(-gamma * x^2)
    .
        If ``gamma='scale'`` is passed then it
    uses
```

```
        1 / (n_features * X.var()) as value of gamma.
```

It is also noteworthy that since the "gamma=scale" version computes the variance, it is much slower (at least for me) than the float values of gamma and is often outperformed by them as well. Therefore, I used the floats throughout all the RBFSamplers.

2.3.2.4 Encoding

The `encode` method of the RBFFeatureEncoder class is responsible for transforming a given state using the RBF transform. It first scales the state and then applies the RBF state featurizer.

```
def encode(self, state):

    # Add new dim to have shape: (1,2) from shape: (2,)
    state = state[np.newaxis, :]

    # Standardize the data beforehand
    scaled = (state - self.s_mean) / self.s_sted

    # Featurize it. shape: (1, n_components)
    state_features = self.rbf_state_featurizer.transform(scaled)

    return state_features[0] # shape: (n_components,)
```

2.3.2.5 Size Property

The `size` property returns the number of features that the state has been projected to as its representation.

```
@property
def size(self):
    return self.n_components
```

2.3.3 Update Transition

In the update_transition function, I first scaled and encode both the current and the next states, which returns their feature representations as discussed above.

```
    # Input s and s_prime encoded using rbf feature encoder
    s_feats = self.feature_encoder.encode(s)
    s_prime_feats = self.feature_encoder.encode(s_prime)
```

For the update part, both the **Eligibility Traces**, **Delta**, and **Weights** are updated according to the formulae for **Q-Learning TD(Lambda)** using the **Backward View** instead of the **Forward View**. Recall that since this is Q-Learning, we take the max for choosing the next action, as shown in the code snippet below.

$$w_{t+1} = w_t + \alpha \delta_t e_t$$
$$\delta_t = R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}, w) - Q(s_t, a_t, w)$$
$$e_t = \gamma \lambda e_{t-1} + \nabla_w Q(s_t, a_t, w)$$

```
    # TODO update the weights

    self.traces = self.gamma * self.lambda_ * self.traces
    self.traces[action] += s_feats

    if done:
        delta = reward - self.Q(s_feats)[action]
    else:
        # Notice the max(Q(s,a)) since this is using the Q-Learning TD(lambda) instead of the SARSA variant
        delta = reward + self.gamma * np.max(self.Q(s_prime_feats)) - self.Q(s_feats)[action]

    # Update the weights w as w_{t+1} = w_t + alpha*delta*e_t
    self.weights[action] = self.weights[action] + self.alpha*delta*self.traces[action]
```

2.4 Conclusion

In this report, we have delved into the realm of two prominent reinforcement learning (RL) algorithms: Sarsa-λ and Q-Learning TD(λ). We began by introducing the Taxi environment and the Epsilon Greedy strategy, a methodology for balancing exploration and exploitation in RL. Subsequently, we delved into the concept of eligibility traces, a mechanism for attributing credit for rewards to past actions, and explored Q-table updates, the fundamental mechanism for updating state-action values in RL. We then presented the Sarsa-λ backward view algorithm, a practical implementation of Sarsa-λ that efficiently updates Q-values.

In the second part of our exploration, we shifted our focus to Q-Learning TD(λ) and its application in the MountainCar-v0 environment. We introduced the MountainCar environment and its challenges, highlighting the need for effective RL algorithms to navigate its complex dynamics. We then introduced Radial Basis Functions (RBFs), a powerful tool for representing continuous-valued features in RL. Finally, we presented the Q-Learning TD(λ) algorithm, demonstrating its ability to effectively learn optimal behavior in the MountainCar-v0 environment.