RL Assignment 2

Matteo Manias 1822363

November 23, 2023

1 Q-learning and SARSA update

The Q-learning update is given by:

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

the SARSA update is given by:

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

the given values of the tabular Q function are:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

substituting the Q-values, reward = 3, discount factor = 0.9 and learning rate = 0.1 with their respective values, we get:

$$Q(1,2) \leftarrow Q(1,2) + 0.1 \cdot [3 + 0.5 \cdot \max(Q(2,1), Q(2,2)) - Q(1,2)]$$

$$Q(1,2) \leftarrow 2 + 0.1 \cdot [3 + 0.5 \cdot \max(3,4) - 2]$$

$$Q(1,2) \leftarrow 2.3$$

that is the new Q-value for state 1, action 2 is 2.3. The other Q-values remain unchanged.

The SARSA update is given by substituting the same values as above but by choosing the action a' from the policy π.

Given that the action choosen by policy π is the same as the action choosen by the max operator in the q-learning update, the SARSA update will be the same as the q-learning update.

2 N-step error as a sum of TD errors proof

the n-step error is given by:

$$G_{t:t+n} - V_{t+n-1}(S_t)$$

where $G_{t:t+n}$ is the n-step return, defined as:

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

By subtracting $V_{t+n-1}(S_t)$ to the n-step return, we get:

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

This can be written as a sum of TD errors by rearranging the terms as follows:

$$\begin{aligned} &= (R_{t+1} - V_t(S_t)) + \gamma(R_{t+2} - V_{t+1}(S_{t+1})) + \ldots \\ &\quad + \gamma^{n-1}(R_{t+n} - V_{t+n-1}(S_{t+n-1})) + \gamma^n V_{t+n-1}(S_{t+n}) - V_{t+n-1}(S_t) \end{aligned}$$

Given that the value estimates don't change from step to step, we can simplify the expression as follows as some terms cancel out:

$$= \gamma^n V_{t+n-1}(S_{t+n}) - V_{t+n-1}(S_t)$$

by noticing that:

The first term $R_{t+1} - V_t(S_t)$ is the TD error at time step t.

The second term $\gamma(R_{t+2} - V_{t+1}(S_{t+1}))$ involves the TD error at time step $t+1$, where $R_{t+2} - V_{t+1}(S_{t+1})$ is the TD error at time step $t+1$.

And continuing this pattern until the n-th term:

This leaves us with a sum of TD errors:

$$\sum_{k=t}^{t+n-1} \gamma^{k-t} \delta_k$$

where $\delta_k = R_{k+1} - V_k(S_k)$ is the TD error at time step k. This demonstrates that the n-step error can be expressed as a sum of TD errors when the value estimates don't change from step to step.

3 Sarsa-λ

The goal of the agent in the Taxi environment is to pick up the passenger at one location and drop him off at another location. The agent can move in 4 directions (north, south, east, west) so the action space in discret, and can pick up and drop off the passenger at the correct location. The state space also is discrete as it is composed of a finite number of states (500).

The RL algorithm used to solve the Taxi environment is Sarsa-λ, an on-policy TD control algorithm that combines the ideas of Monte Carlo updates and TD updates, and can be express in pseudo-code as follows:

```
Initialize Q(s,a) arbitrarily
Repeat (for each episode):
    E(s,a) = 0, for all s,a
    Initialize S
    Choose A from S using policy derived from Q (e.g., epsilon-greedy)
    Repeat (for each step of episode):
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., epsilon-greedy)
        delta = R + gamma * Q(S',A') - Q(S,A)
        E(S,A) = E(S,A) + 1
        For all s,a:
            Q(s,a) = Q(s,a) + alpha * delta * E(s,a)
        S = S'; A = A';
    until S is terminal
```

The required implementation of Sarsa-λ used to solve the Taxi task, in the student.py file are:

calculate the TD error:

```
delta = reward + self.gamma * self.Q[next_state][next_action] - self.Q[state][action]
```

here the temporal difference error is calculated using the Q function as the difference between the reward and the discounted value of the next state-action pair, choosen according to the current policy

update the eligibility trace:

```
self.E[state][action] += 1
```

as the the state-action pair is visited, the eligibility trace is incremented by 1 keeping track of the number of times the state-action pair is visited and indicating a higher importance of the state-action pair.

4 RBF encoder, TD - λ LVFA Q-learning

The goal of the agent in the mountain car problem is to reach the flag on the right side of the environment. The agent can move in 3 directions (left, right, stay) so the action space in discret, and can move in the x and y direction. The state space in the problem is continuous, this can lead to a large number of states, so it is only solvable using discretization techniques o function approximation. In practice the algorithm builds an approximation of the Q function using a set of weights, in this case with linear value function approximation (LVFA) and radial basis function (RBF) encoder. In this case the RBF encoder is used to encode the state space into a feature space and then the TD λ LVFA Q-learning algorithm is used to solve the problem.

```
Initialize weights w arbitrarily
Repeat (for each episode):
    E = 0, for all s,a
    Initialize S
    Choose A from S using policy derived from Q (e.g., epsilon-greedy)
```

```
 Repeat (for each step of episode):
     Take action A, observe R, S'
     Choose A' from S' using policy derived from Q (e.g., epsilon-greedy)
     delta = R + gamma * Q(S',A') - Q(S,A)
     E = gamma * lambda * E
     E(S,A) = E(S,A) + Gradient(Q(S,A),w)
     w = w + alpha * delta * E
     S = S'; A = A';
 until S is terminal
```

The specific implementation of TD λ LVFA Q-learning used to solve the mountain car task, in the student.py file are Under the `update_transition_backwards` function:

```
#calculate delta td error
      delta = reward + self.gamma*self.Q(s_prime_feats).max() - self.Q(s_feats)[a]
```

Here the temporal difference error is calculated using the Q function as the difference between the reward and the discounted value of the next state-action pair, note that the next action is choosen according to the best action at the next state (max operator) so this implementation is an on policy Q-learning algorithm.
To chagne the algorithm to an off policy Q-learning algorithm, the next action should be choosen according to the current policy as in the SARSA algorithm.

```
#calculate delta td error
      a_prime = self.epsilon_greedy(s_prime_feats)
      delta = reward + self.gamma*self.Q(s_prime_feats)[a_prime] - self.Q(s_feats)[a]
```

The eligibility trace is decayed by a factor of gamma * lambda as follows:

```
#decay traces
      self.traces = self.gamma*self.lambda_*self.traces
```

this is done to keep track of the number of times the state-action pair is visited and proportionally weigh the importance of the state-action pair and the recency of the visit.
The eligibility trace is then updated for the state-action pair as follows.

```
#accumulate traces
      self.traces[a] = self.traces[a] + s_feats
      #substitute traces
      #self.traces[a] = s_feats
      #update weights
      self.weights[a] += self.alpha*delta*self.traces[a]
```

The eligibility trace is accumulated by adding the state features to the trace, note that `s_feats` is equal to the gradient of the Q function with respect to the weights and that the traces can be accumulated or substituted.

The RBF encoder is used in the mountain car problem to encode the state space into a feature space, this is done to increase the dimensionality of the state space and to make the problem solvable by linear value function approximation (LVFA).
The RBF encoder builds its representation by calculating the distance between the state and a set of centroid points, the distance is calculated using the gaussian function.
in the specific implementation the RBF encoder is initialized with 4 different gamma values,

```
# Initialize RBF samplers with different parameters
self.rbf_space = [
  ("rbf1", RBFSampler(gamma=5.0, n_components=100)),
  ("rbf2", RBFSampler(gamma=2.0, n_components=100)),
  ("rbf3", RBFSampler(gamma=1.0, n_components=100)),
  ("rbf4", RBFSampler(gamma=0.5, n_components=100))
]

self.design_matrix = FeatureUnion(self.rbf_space)
```

so the final encoding in the combination of the 4 different RBF encoders, this is done provide the algorithm with a more complex representation of the state space that is capable of capturing different aspects of the state space.

It is good practice to scale the state space before encoding it with the RBF encoder, as position and velocity have different scales and this can lead to a bad representation. In the `__init__` for the `RBFFeatureEncoder` class:

```
observation_examples = np.array([env.observation_space.sample() for x in range(100)])
      self.scaler = sklearn.preprocessing.StandardScaler()
      self.scaler.fit(observation_examples)
```

the state space is sampled 100 times, this returns 100 different states that are used to fit the scaler, and then scaled using the `StandardScaler` class from the `sklearn.preprocessing` module.

The scaled space state is then used to fit the RBF encoder:

```
self.design_matrix.fit(self.scaler.transform(observation_examples))
```

and finally the state is encoded using the RBF encoder in the `encode` function:

```
def encode(self, state):
  scaled = self.scaler.transform(state.reshape(1, -1))
  state_features = self.design_matrix.transform(scaled)
  return state_features
```

this function returns the encoded state features that are then used to calculate the TD error as a matrix of dimension 1x400, where 400 is the sum of the number of components of the 4 RBF encoders.