CS6700 : Reinforcement Learning
Written Assignment #2

Name: Girish Madhavan V **Roll Number:ME20B072**

1. (3 marks) Ego-centric representations are based on an agent's current position in the world. In a sense the agent says, I don't care where I am, but I am only worried about the position of the objects in the world relative to me. You could think of the agent as being at the origin always. Comment on the suitability (advantages and disadvantages) of using an ego-centric representation in RL.

Solution:
Advantages :

- Ego Centric Representations simplify the state space required for agent to train, thereby reducing the complexitiy of the problem
- It could speed up the training in certain scenarios while also requiring a lot less memory, Hence it would be an efficient method

Disadvantages :

- While Ego Centric approach works for simple environments, it tends to perform poorly in complex environments. Especially when the same features used for identifying the states are repeated and some of them could give a huge positive reward while others could end up in a huge negative rewards
- Similarly Ego Centric approach will require the agent to know as much information as possible about the surrounding states to match a feature, so an action could be taken. This could hinder its performance when the all the required states are not completely observable

2. (4 marks) One of the goals of using options is to be able to cache away policies that caused interesting behaviors. These could be to rare state transitions, or access to a new part of the state space, etc. While people have looked at generating options from frequently occurring states in a goal-directed trajectory, such an approach would not work in this case, without a lot of experience. Suggest a method to learn about interesting behaviors in the world while exploring. [*Hint: Think about pseudo rewards.*]

Solution:

To generate options that are capable to exploring or reaching rare transitions we could provide additional rewards to agent apart from the one given by the actual interaction with the environment, called Pseudo Rewards. Subsequently the appearance of any rare state can be marked by an appropriate pseudo rewards, Where upon several episodes of training aimed at maximizing the cumulative actual and pseudo rewards these rare transitions can be identified. One of the challenges of this approach would be assigning pseudo rewards for each rare transition, as this would require careful monitoring during the training process so as to be able to identify these rare states without looping back to the general state space of system

Proposed Algorithm:

- Initialize the environment and agent
- Choose a suitable action selection algorithm [ϵ-greedy, Soft-max, etc]
- Start exploration of the environment while computing the rewards as well as pseudo rewards associated with each transition
- Update the policy and value function based for both rewards till termination
- Adjust the pseudo rewards required interesting behaviours are captured
- Upon completion of exploration phase, the policies that led to rare transitions can be stored

3. (2 marks) Consider a navigation task from a fixed starting point to a fixed goal in an arbitrarily complex(with respect to number of rooms, walls, layout, etc) grid-world in which apart from the standard 4 actions of moving North, South, East and West you also have 2 additional options which take you from fixed points A_1 to A_2 and B_1 to B_2. Now you learn that in any optimal trajectory from the starting point to the goal, you never visit a state that belongs to the set of states that either of the options can visit. Now would you expect your learning to be faster or slower in this domain by including these options in your learning? If it is case dependant, give a potential reason of why in some cases it could slow down learning and also a potential reason for why in some cases it could speed up learning.

Solution:

In this domain the introduction of new options could speed up or slow down the learning process depending on the scenario its used on :

Faster Learning :
The presence of options in certain cases could learn trajectories that avoid dead states present in the environment. By not visiting these dead states repeatedly we could explore the environment more effectively thereby leading to faster learning.

Slower Learning :
The presence of options could hamper the learning in cases where the trajectory incorporating the options can make the agent perform unnecessary exploration that wouldn't contribute to learning the optimal policy.

4. (3 marks) This question requires you to do some additional reading. Dietterich specifies certain conditions for safe-state abstraction for the MaxQ framework. I had mentioned in class that even if we do not use the MaxQ value function decomposition, the hierarchy provided is still useful. So, which of the safe-state abstraction conditions are still necessary when we do not use value function decomposition.

Solution:

The following safe-state abastraction conditions are still necessary when value function decomposition is not used :

- **Leaf Node Irrelevance**, Here the Leaf nodes represent the primitive actions, according to this condition if the transition rewards are same for a set of states while taking a primitive action then they become irrelevant for that primitive action. Such a condition would give flexibility while implementing low level actions without impacting the hierarchy
- **Subtask Irrelevance**, according to which a set of states are irrelevant if the state transition probability distribution for each child action can be written as a product of two distribution. From this we can conclude that there the subtasks are mutually independent, so the calculation of their associated values becomes simpler
- **Termination**, here the termination of a child subtask would imply the termination of the parent subtask as well, this could again reduce the state space of the model
- **Shielding**, Suppose the path connecting the root to a state 's' of the subtask M_i contains a state that has terminated then there would be no necessity to compute the values associated with 's'

5. (1 mark) In any model-based methods, the two main steps are **Planning** and **Model**

Update. Now suppose you plan at a rate of F_P *(F_P times per time-step)* and update the model at a rate of F_M, compare the performance of the algorithm in the following scenarios:

1. $F_P \gg F_M$
2. $F_P \ll F_M$

Solution:
$F_P >> F_M$:

In this case the frequency of planning much higher than the frequency of model update, this could lead to the inaccurate decisions made by the algorithm as the model updates occur slowly. Also since the update frequency of the model is much slower, it would be unable to capture the system dyanmics properly in case of rapid transitions.

$F_P << F_M$:
In this case the frequency of planning is much lower than the frequency of model update, here the policy is unable to adapt to the frequent model updates and as a result it might end up learning sub optimal policies.

6. (3 marks) In the class, we discussed 2 main learning methods, policy gradient methods and value function methods. Suppose that a policy gradient method uses a class of policies that do not contain the optimal policy; and a value function based method uses a function approximator that can represent the values of the policies of this class, but not that of the optimal policy. Which method would you prefer and why?

Solution:

In general, when faced with the decision between policy gradient methods and value function methods and when we are unable to capture the optimal policy, the choice largely depends on the characteristics of the environment.

Suppose we have a stochastic environment with discrete states. In such a scenario, I would prefer to utilize a value function-based method. This preference stems from the fact that value function methods tend to be more sample efficient in such environments. By estimating the value of each state, we can make informed decisions about which actions are likely to lead to higher cumulative rewards. Additionally, the computational complexity of value function methods in stochastic environments with discrete states tends to be lower compared to policy gradient methods. This is because policy gradient methods require estimating gradients through sampling, which can become prohibitively expensive in environments with high stochasticity.

Conversely, in a deterministic environment with continuous states, I would lean towards employing policy gradient methods. In such environments, the computation involved in policy gradient methods is often more manageable compared to value function methods. Policy gradient methods directly optimize the policy parameters to maximize expected rewards, bypassing the need to estimate the value of each state explicitly. This can be advantageous in deterministic environments where the optimal policy may involve subtle, continuous adjustments, making it more natural to directly optimize the policy.

7. (3 marks) The generalized advantage estimation equation ($\hat{A}_t^{GAE}$) is defined as below:

$$\hat{A}_t^{GAE} = (1-\lambda)\left(\hat{A}_t^1 + \lambda \hat{A}_t^2 + \lambda^2 \hat{A}_t^3 + ...\right)$$

where, $\hat{A}_t^n$ is the n-step estimate of the advantage function and $\lambda \in [0,1]$.

Show that

$$\hat{A}_t^{GAE} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

where δ_t is the TD error at time t, i.e.

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Solution: From the definition of Advantage function we know that

$$\hat{A}_t^n = \sum_{i=0}^{n-1} \gamma^i r_i + \gamma^n V(s_{t+n}) - V(s_t)$$

Where the summation term and the state value function of state at (t+n) represent expected rewards obtained after taking actions for n timesteps without following a policy and then following a policy to take action from the n+1 th timestep

Hence $\hat{A}_t^1 = r_t + \gamma V(s_{t+1}) - V(s_t) = \delta_t$
Similarly,

$$\hat{A}_t^2 = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)$$

$$= (r_t + \gamma V(s_{t+1} - V(s_t)) + \gamma(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1})$$

$$= \delta_t + \gamma\delta_{t+1}$$

In general we can say :

$$\hat{A}_t^i = \sum_{j=0}^{i-1} \gamma^j \delta_{t+j}$$

Therefore :

$$\hat{A}_t^{GAE} = (1-\lambda)\left(\delta_t + \lambda(\delta_t + \gamma\delta_{t+1}) + \lambda^2(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}) + ...\right)$$

On rearranging terms we get :

$$\hat{A}_t^{GAE} = (1-\lambda)\left(\delta_t(1+\lambda+\lambda^2+...) + \gamma\lambda\delta_{t+1}(1+\lambda+\lambda^2+...) + \gamma^2\lambda^2\delta_{t+2}(1+\lambda+\lambda^2+...) + ...\right)$$

Using infinite sum computation we get : $\frac{1}{1-\lambda} = 1 + \lambda + \lambda^2 + ...$ for $0 < \lambda < 1$
Substituting this into our Generalized Advantage function computation we obtain :

$$\hat{A}_t^{GAE} = \frac{(1-\lambda)}{(1-\lambda)}\left(\delta_t + \gamma\lambda\delta_{t+1} + \gamma^2\lambda^2\delta_{t+2} + ...\right) = \sum_{i=0}^{\infty} \gamma^i \delta_{t+i}$$

8. (3 marks) In complex environments, the Monte Carlo Tree Search (MCTS) algorithm may not be effective since it relies on having a known model of the environment. How can we address this issue by combining MCTS with Model-Based Reinforcement Learning (MBRL) technique? Please provide a pseudocode for the algorithm, describing the loss function and update equations.

Solution: Inorder to combine the both algorithms we can use MBRL to generate the model for the next environment state after a transition and use this generated model to update the Monte Carlo Tree. This is repeated until convergence is obtained.

```
Algorithm 1: Integration of MCTS with MBRL
  Input: Environment dynamics
  Output: Optimal policy $\pi^*$
  Initialize $Q(s,a)$ and $Model(s,a)$ ∀ s,a;
  Initialize the search tree $T$;
  while true do
      Use a action selection algorithm (e.g., $\epsilon$ - greedy) to select a leaf node;
      If selected node is not terminal then expand it;
      Perform Monte Carlo Simulation from the expanded node using the
        trained model;
      while the current node is not the root do
          Update the visit count and total value of the current node;
          Update the action-value estimate associated with the edge leading to
            this node;
      for each state-action pair $(s,a)$ generated in the simulation do
          Predict the next state and reward using the model function:
            $(s',r) = Model(s,a)$;
          Compute the loss and update the model parameters to minimize
            prediction error: $loss = (r + \gamma \cdot \max(Q(s',a')) - Q(s,a))^2$;
          Back-propagate the loss to update the model parameters;
      for each state-action pair $(s,a)$ generated in the simulation do
          Update the action-value estimate using the simulation reward and
            the predicted next state: $Q(s,a) =$
            $Q(s,a) + \alpha \cdot (simulation\_reward + \gamma \cdot \max(Q(s',a')) - Q(s,a))$;
```