CS6700 : Reinforcement Learning
Written Assignment #2

Deadline: 9 May 2024, 11:59 pm

Name: Srikar Babu Gadipudi **Roll Number: EE21B138**

- This is an individual assignment. Collaborations and discussions are strictly prohibited.
- Be precise with your explanations. Unnecessary verbosity will be penalized.
- Check the Moodle discussion forums regularly for updates regarding the assignment.
- Type your solutions in the provided LaTeX template file.
- **Please start early.**

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: Ego-centric representations are all about the agent's self-centered view of the world. It is having to build a map with the agent as the center and everything else positioned relative to it. The states observed by the agent will also be transformed into this agent's map only to keep track of changes in the environment relative to the agent. Here are the advantages and disadvantages of ego-centric representation in RL.

- **Advantages:**
  - Sometimes, it can become easier for the agent to learn a policy based on its position and relative object locations. This is because the agent can ignore to build a complex understanding of the environment to the fullest extent.
  - In environments that are dynamic, i.e., the layout keeps changing (perception of a robot in a navigation task) ego-centric representation can be more efficient.
  - In a way this is how humans or robots interact with their surroundings. They might not have the complete map but learn successful actions based on their current perception. This allows for easy transfer of the model in training, let us say, a robot.
  - The agent might also be more adaptable to new or unseen environments by relying only on its observations to navigate or make decisions.
  - In a way this can lead to a lesser number of values/variables to keep track of during training. Essentially performing abstraction in some sense by making everything relative to the agent.
- **Disadvantages:**
  - Ego-centric representation in an RL agent might struggle in large and complex environments, especially in cases where the entire map of the environment is needed. For example, in the case of a search and rescue task, utilizing ego-centric representation might not be a good idea because the robot needs to know how the entire environment to develop an effective search policy and rescue policy.

- Sometimes the agent might not be able to generalize well to new environments. The policy learned might only work from a specific starting point in a particular environment but might fail when it starts somewhere new.
- Planning ahead of time might become difficult. Prediction of long-term sequences becomes a problem because the consequences of the actions taken by the agent might not be effectively represented in the ego-centric representation. A full picture of the environment might make it better.

Ego-centric representation in RL would work well for simple tasks such as navigation, especially when a robot or a human-like agent is present in the environment. It would not be ideal for complex environments and tasks that require long-term planning, such as search and rescue tasks.

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: Before considering pseudo rewards, we can consider defining actions over the state space that would allow us to take such an "interesting behaviour" pathway. This can be performed using notions such as identifying bottlenecks using graph partitioning or betweenness centrality. These methods differ from methods that consider only goal directed states/actions, such as Diverese Density method. These methods allow us to take these "interesting behaviour" pathways by identifying the states/actions that lead to this.

Coming to using pseudo rewards, we can consider modifying the reward formulation such that when any rare transitions occur we increase the magnitude of rewards received by the agent and update the value for that state-action pair accordingly. This change in reward formulation can be inspired by the concept of "Exploration", which was discussed with model-based reinforcement learning techniques. Essentially the concept of exploration bonus can be extended to this particular case. We can say that knowledge-based intrinsic motivation would be the solution to this (as opposed to competence-based intrinsic motivation which is goal-oriented). We can in fact use model-based reinforcement learning to provide the agent with pseudo rewards for encountering rare state transitions. For example, in the Dyna Q+ algorithm apart from the natural reward observed from the environment by executing a state-action pair we also add an exploration bonus term which is given as

$$r^i(s, a, s') = \sqrt{l(s, a)}, \tag{1}$$

where $r^i(s, a, s')$ is the exploration bonus term added to the natural reward (constituting the pseudo reward in total) and $l(s, a)$ is the the number of timesteps since the last trail at (s, a).

Another perspective to this problem would be, given that we know the rare state transitions we can formulate a new reward function for this case. For instance, an extra reward can be given when a new state has been encountered which was never seen before or the agent encounters a state with high uncertainty in the expected outcome (example, entering a dark room). These rewards can be utilized and any model-based or model-free RL algorithm can be employed to solve the task at hand based on the requirements.

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 dependent, 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: The slowdown in the learning process would be case dependent. Considering that the agent can choose from primitive actions and the two options mentioned above that do not include any state that would lie on the optimal path to the end goal, we can expect a slower learning process because the agent spends steps to explore the options only to realise that picking them would not be optimal.

That being said, this becomes case dependent based on the algorithm used for learning. If we employ the SMDP Q learning algorithm, unless the agent finishes executing a particular option we do not receive that option's reward and hence cannot update the value function for that option. Neither are the values for the primitive actions taken along the way of executing the option to be updated. This leads to the agent unnecessarily picking these options only to determine that the options are suboptimal after significant learning steps.

On the other hand, if we employ Intra Option Q learning, being more sample efficient, the learning at least becomes comparable to the case without these options or slightly slower. Since the values for primitive actions and options are updated irrespective of whether an action or option is being executed, the agent learns the general dynamics of the environment faster. Hence this algorithm is more sample efficient. I mentioned it might be slightly slower because we need to account for the case where the options did not exist it would similarly explore the environment, and the extra constraint of finishing the option once chosen would not exist. Although this slowdown is dependent on the nature of the environment and the underlying dynamics, sometimes the performance could even be comparable between the with and without options case.

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 MaxQ framework considers 5 safe-state abstraction conditions. They are:

- Max Node Irrelevance: Examining the subgraphs from a particular node in the MaxQ graph, if the transition probabilities, reward functions, pseudo-reward functions and termination conditions do not vary for a set of state variables, then these state variables can be abstracted out to reduce the number of values to be estimated.
- Leaf Irrelevance: This applies to only leaf nodes in the MaxQ graph. A set of state variables can be abstracted out if the expected reward function does not change for the state action pairs that differ only in the state variables to be abstracted out. This follows from the max node irrelevance condition with a weaker irrelevance condition.
- Result Distribution Irrelevance: A set of state variables become irrelevant from a particular node if all abstract policies that can be executed from that node and its descendant nodes do not differ in the transition probabilities. This translates to having the same competition cost function for the states that only differ in the irrelevant state variables.

- Termination: For a task in a MaxQ graph, if the goal predicate is true for all states then the completion cost function for any policy being executed from that node is zero and does not need to be explicitly mentioned.
- Shielding: For a node in a MaxQ graph, if from the root to this node for all states there exists a subtask whose termination predicate is true, then there is no need to represent the completion cost function at all for this node as it will never execute the task.

We know that without the MaxQ value decomposition techniques, some of these safe-state abstraction conditions can be used in hierarchical reinforcement learning. These conditions are the "Max Node Irrelevance" and the "Leaf Irrelevance". These conditions simply state that a change in the irrelevant state variables will have no impact on the output of any action/subtask that is executed by following any abstract hierarchical policy. From the condition definition mentioned above, it is evident that those state variables that satisfy those conditions can be abstracted out not only in the MaxQ framework, but in any hierarchical reinforcement learning techniques to reduce the number of values to be predicted.

To some extent, the "Shielding" condition can also be considered. In the process of reaching a node if the termination predicate becomes true then the agent will never execute the task mentioned by that node. Therefore, the values for that particular node need not be estimated. However, if we impose the "Max Node Irrelevance" and "Leaf Irrelevance" conditions this happens implicitly when approached systematically.

The other conditions, "Result Distribution Irrelevance" and "Termination", are specific to MaxQ value function decomposition because they focus on the characteristics of the completion cost function and its relevant terms. This need not be the case with other hierarchical learning techniques.

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: We can expect the second case to perform better than the first case, i.e., $F_P \ll F_M$ would perform better than the case where $F_P \gg F_M$.

This is because F_M is the number of modeling steps, if we model the environment a lot of times, it would only lead to a better model of the environment we are trying to solve. Although this is not a very efficient way to solve the environment it would not cause a hindrance in the learning process. On the other hand, if we plan more than the number of times we model, we are essentially trying to solve the task with an incorrect model of the environment. Because the model might deviate from the actual environment dynamics, updating it much less than the number of times we plan would only lead to learning from an incorrect model. This affects the performance of the model in a bad manner.

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 this scenario, a value function based method would be preferable.

- The policy gradient method is restricted by the class of policies it can explore. Since the class does not include the optimal policy, the method is inherently limited in its ability to find the best solution.
- On the other hand, even though value function approximation cannot represent the optimal policy, it can still provide valuable information. It can estimate the value of different policies within the limited class. This information can be used to improve the policy within this restricted set, potentially leading to a better solution compared to the initial policy. Also, the policy we arrive at using the value function methods would be the greedy policy with respect to the values of every state-action pair, even though we might not arrive at the true values of the state-action pairs, the approximation might be able to weigh the values according to the returns received. This might lead to an optimal policy by following these values estimated by the function approximator.

In essence, the value function method can still operate and make improvements within the restricted policy class, whereas the policy gradient method is confined to a limited search space that excludes the optimal solution.

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: The generalized advantage equation ($\hat{A}_t^{GAE}$) given in the question is widely used in all popular implementations to estimate the advantage value. In order to derive the relation mentioned in the question we need to start with the definition of n-step advantage function

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

where γ is the discount factor.

Now, to the above expression add and subtract $\gamma V(s_{t+1}$ term. By simply rearranging the terms we get,

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

We know that

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

. Therefore,

$$\hat{A}_t^n = \delta_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) - \gamma V(s_{t+1}).$$

Now add and subtract $\gamma^2 V(s_{t+2}$ and club terms in a similar as above, we get

$$\hat{A}_t^n = \delta_t + \gamma \delta_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n}) - \gamma^2 V(s_{t+2}).$$

Observing this pattern, if we repeat this process for $n-1$ times. We get,

$$\hat{A}_t^n = \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \dots + \gamma^{n-1} \delta_{t+n-1} + \gamma^n V(s_{t+n}) - \gamma^n V(s_{t+n}).$$

Notice that the last two terms are the same. Therefore,

$$\hat{A}_t^n = \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \dots + \gamma^{n-1} \delta_{t+n-1}.$$

Substituting this in the generalized advantage equation, we get

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

Rearranging the terms in the above expression, we get

$$\hat{A}_t^{GAE} = (1-\lambda)\Big(\delta_t(1 + \lambda + \lambda^2 + \dots) + \gamma\delta_{t+1}(\lambda + \lambda^2 + \dots) + \gamma^2\delta_{t+2}(\lambda^2 + \lambda^3 + \dots)\dots\Big).$$

For $\lambda \in [0,1]$, we can use the infinite summation result to get ($1-\lambda$ term gets cancelled),

$$\hat{A}_t^{GAE} = \delta_t + \gamma\lambda\delta_{t+1} + \gamma^2\lambda^2\delta_{t+2}\dots.$$

Therefore,

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

Hence Proved.

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: We can combine the concepts of Monte Carlo Tree Search (MCTS) and Model-Based RL (MBRL) in several ways. A combination of these concepts are widely used in various fields of research. The famous AlphaGo agent is trained using a version of this technique!

A model can be built using MBRL approaches that allow us to capture the dynamics of the environment by interacting with the real environment. And then MCTS can be learned by simulating trajectories on this model. This allows exploration without directly interacting with the real environment, which can be expensive or risky. The interaction with the real-world environment to plan the model can be done using the actions suggested by the MCTS. In detail, here is the pseudocode for the algorithm mentioned above:

Algorithm:

```
Initialize Model ($\mathcal{M}$) to output next state and reward given state and action as input ($\mathcal{M}(s, a) = (s', r)$).
Initialize MCTS Tree from the start state $s_0$.
while Training incomplete do
    Perform Selection to reach the leaf node.
    From the Selection step execute acitons in the real-world environment.
    Perform Update Model ($\mathcal{M}$).
    Perform Expansion of the tree.
    Perform Simulation, execute this step on $\mathcal{M}$.
    Perform Backup.
end while
```

Here the *Selection* step can be performed using the action selection technique inspired by the UCB1 algorithm

$$\arg\max_a Q(s, a) + c\sqrt{\frac{\ln t_s}{n_{(s,a)}}},$$

where t_s is the number of times the state s is visited and $n_{(s,a)}$ is the number of times the action a has been selected from the state s. At every node in the tree, these actions are performed from that state in the real-world environment, till we reach a leaf node.

The observed transitions (s, a, s', r) can be used to execute the *Update Model* step. Algorithms like Dyna Q can be used for this step, where we keep track of the transition probabilities for every (s, a, s', r), such that given a (s, a) pair we get the transition dynamics. In a deterministic case, this becomes simpler by just mapping (s, a) to observed (s', r). If a neural network or any other parametrization is used for this step, then MSE loss or Cross-Entropy loss function can be used as per the specification.

The *Expansion* step is performed as in the usual case with MCTS algorithms. A new child node is added to the tree to that node which was optimally reached during the selection process.

The *Simulation* step is performed to gather the returns from that child node till termination by performing rollouts on the model $\mathcal{M}$. Any abstract policy can be employed for performing these rollouts on the model.

Backup step can be performed using our regular Monte-Carlo update

$$Q(s, a) = Q(s, a) + \alpha(G - Q(s, a)),$$

where G is the return computed from that node till termination (this is computed in the simulation step) and α is the learning rate. This backup is done for all the nodes in the graph, therefore the value of G changes accordingly for every node. Here if we consider a neural network or any function approximation technique to estimate the values then the loss function that can be used would be the squared error

$$\mathcal{L} = \frac{1}{2}(G - Q(s, a))^2.$$

By leveraging the learned model, MCTS-MBRL can explore complex environments more efficiently than vanilla MCTS. The model provides a "safe" space to try out actions and evaluate their potential outcomes before committing to them in the real world.

Alternative Approach: Another way to do this would be to use the model learned using MBRL techniques mentioned above to estimate values/scores for every node in MCTS. This gives a better estimate of the better-performing sequence of actions. In this case, the selection step is performed using information provided by the model and the maximizing step as mentioned earlier. The expansion step remains the same, while rollouts would have to be performed in the real-world environment while also simultaneously updating the model. Here rollouts can be performed on the model too, given that some planning steps are performed in the real-world environment to update the model. The backup step will follow the same as above. Updation steps and the loss function will remain the same with appropriate changes according to the architecture.