Assignment 2 Problem Statement

SoC: RL for NLP

This is a mandatory assignment. (Submission of this assignment is compulsory for certification). The deadline for this assignment will be **30th June**.

We will now direct our attention towards one of the most important Python libraries available for the use of RL enthusiasts – OpenAIGym. It has amazing APIs which directly provide a visual experience of RL progression.

We've provided 4 games below which you can make an agent on. We'll share the sample code for the Taxi game in a few days so that y'all have something to refer to if you're stuck somewhere. You can choose **ANY ONE** of the other 3. Choose any one of the three games given after Taxi and implement algorithms that lead to consistently good rewards. More instructions follow.

But first, What's OpenAI Gym?

OpenAI has recently become popular due to ChatGPT, a breakthrough that's the talk of the town. OpenAI as a research group has developed some amazing APIs like DALL-E which essentially generates images based on given prompts. So basically, OpenAI is a team of researchers that leads the global AI landscape.

OpenAI gym is a Reinforcement Learning API. The website here documents the toolkit.
It provides a proper baseline to play around with RL algorithms by providing visual tools for games and making environments, observation spaces, and action spaces available for decision-making in a simple fashion.

Important:

Since these games will require repeated execution of certain snippets of code, it would be inefficient to run whole codes for just some small snippets of code. Jupyter Notebooks come to our rescue here. They help us execute certain snippets of code repeatedly without us having to execute other parts of the code always. We therefore urge y'all write these codes on Jupyter Notebooks. Jupyter Notebooks use python only so the language shouldn't be a problem.

You can get the exact steps for the installation of Jupyter Notebooks through a simple Google Search. After installing Jupyter Notebook, you can download the VSC extension for it and run Jupyter Notebook on VSC itself.

Game 1: Taxi

Focus: Monte Carlo vs TD Learning

Documentation: https://gymnasium.farama.org/environments/toy_text/taxi/

The goal is to move the taxi to the passenger's location, pick up the passenger, move to the passenger's desired destination, and drop off the passenger. Once the passenger is dropped off, the episode ends. This game is simple because both the action and observation spaces are discrete.

Model the state vector carefully. There are 500 possible states in the game. Read the documentation thoroughly and ping for doubts. Your implementation should not rely on your agent knowing anything about the environment before it begins playing.

Task: Build a Monte Carlo algorithm and a TD algorithm to attempt to solve this game. Play around with the parameters and try to find the parameters which help you come up with faster-converging algorithms. Make a plot that shows the Monte Carlo cumulative reward and the TD cumulative reward for each episode, for 5000 episodes. Make another plot for each of these algorithms separately in which you play around with parameters in your update formula. What these graphs will depict is up to you, but they should convince me that your choice of parameters is better than the others you tried.

Game 2: CartPole

Focus: Discretization of Continuous Spaces in TD(λ)

Documentation: https://gymnasium.farama.org/environments/classic_control/cart_pole/

The observation space defining the current state has infinitely many values. Discretize this range and then apply either Monte Carlo or TD Learning algorithms.

What does discretization mean? Suppose in a hypothetical game where you are the agent and the current state is defined by a real number in [0,25]. These are infinitely many states, and you can't apply simple MC or TD methods here.

Instead what you do is, reduce your state to simply be defined by the Greatest Integer Less than the current state real number you have. For example, if your current state is (7.56) it automatically is reduced to (7) by your agent. Now you only have to deal with 25 states, much less than Infinite!

Also, you could have instead defined [0,0.5] to be a state and (0.5,1] to be a state and so on. This would give you 50 states. We say that your granularity of discretization is finer here than in the previous case. You may as well have chosen to reduce to only 10 states instead of 25, that means you have coarser granularity. You can easily define granularity as the number of states you reduce your system to. How do you think granularity affects the time taken by your algorithm to converge? How do you think it affects the space occupied by your program in the system for storing state-related values?

Task: Implement a TD(λ) or Sarsa(λ) Learning Method. Try different granularities of discretization. Plot graphs showing the reward vs episode plots for different granularities. Play around with other parameters like learning rate, discount factor, etc. Plot graphs that compare the performance of your algorithm under these different scenarios. (What exactly you plot is up to you but it should convince me that the parameters you claim to be working well are indeed the best).

Game 3: MountainCar

Focus: Discretization of Continuous Spaces in Policy Iteration
Documentation: https://gymnasium.farama.org/environments/classic_control/mountain_car/

The observation space defining the current state is has infinitely many values. Discretize this range and then apply either Monte Carlo or TD Learning algorithms.

What does discretization mean? Suppose in a hypothetical game where you are the agent and current state is defined by a real number in [0,25]. These are infinitely many states, and you can't apply simple MC or TD methods here.

Instead what you do is, you reduce your state to simply be defined by the Greatest Integer Less than the current state real number you have. For example, if your current state is (7.56) it automatically is reduced to (7) by your agent. Now you only have to deal with 25 states, much less than infinite!

Also, you could have instead defined [0,0.5] to be a state and (0.5,1] to be a state and so on. This would give your 50 states. We say that your granularity of discretization is finer here than in the previous case. You may as well have chosen to reduce to only 10 states instead of 25, that means you have coarser granularity. You can easily define granularity as the number of states you reduce your system to. How do you think granularity affects the time taken by your algorithm to converge? How do you think it affects the space occupied by your program in the system for storing state-related values?

Task: Choose the Policy Iteration algorithm. You can deal with deterministic policies for simplicity or with stochastic policies if you are daring. Try different granularities of discretization. Plot graphs showing the reward vs episode plots for different granularities. Play around with other parameters like learning-rate, discount factor, etc. Plot graphs which compare the performance of your algorithm under these different scenarios. (What exactly you plot is upto you but it should convince me that the parameters you claim to be working well are indeed the best). Use a grid-map to visually display your policy in the middle and at the end of the learning process.

Game 4: LunarLander

Focus: Value Approximation Method vs Q-Learning

Documentation: https://gymnasium.farama.org/environments/box2d/lunar_lander/

This is a much more complicated problem than the three preceding ones. It will require significantly more training with our simple algorithms, probably even half a million episodes of practice! Choose the version of the Lunar Lander with continuous observation space. Define a linear value function as described in lecture 6. Use gradient descent RL to determine the optimal weights of the function. Choose the version of the Lunar Lander with discrete observation space. Implement a Q-Learning Algorithm.

Task: Compare the reward vs episode curves of the two learning algorithms. Play around with other parameters like learning-rate, discount factor, etc. Plot graphs which compare the performance of your algorithm under these different scenarios. (What exactly you plot is upto you but it should convince me that the parameters you claim to be working well are indeed the best). Use a heat-map to visually display the weights associated with your features at different stages of learning such as the beginning, the quartile episodes and at the end of the training.

Submission Details:

Submit any files you need including your trained .json models or any other files you used to store your model in. Also add the following file.

- A small report summary.pdf explaining the results obtained and your inferences from it. Add any graphs or plots which you've used to support your inferences. Add any references you've used in the report too.

Please note: This is a compulsory task and must be completed for certification. The hard deadline for the same is **JUNE 30 EOD.**