Week 2 Problem Statement: Game 1

SoC: Street Fighter II – Reinforcement Learning

This Problem Statement will be mandatory (i.e., required for certification).

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.

This Problem Statement is recommended to be attacked in groups of two.

The games become progressively more difficult from 1 to 4. You can choose ANY ONE.

Choose any one of the following four games 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 have 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.

Game 1: Taxi

Focus: Monte Carlo vs TD Learning

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

Pre-requisites: Lecture 3 and 4 of Prof. David Silver's series.

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. Choose this if you are short on time or are approaching this work solo.

Model the state vector carefully. There 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 which 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.

Video: Record a video in which you explain to viewers, in short, the graphs you found and why you think the graphs are the way you see them. Also argue why MonteCarlo or TD took the upper hand over the other according to your understanding. Also explain your implementation and how to run the code.

Save the Model: Once your agent has learnt well, you usually want to save the learning they did, so that the next time you want to play the game, you don't have to re-train an agent from scratch. And since you are using value-based methods and model free learning, to save a model, all you have to do is save your values. Saving in even a txt file works if your space is small. But this is a great opportunity for you to learn how to use .json files!

Game 2: CartPole

Focus: Discretization of Continuous Spaces in TD(λ)

Documentation: Link

Pre-Requisites: Lecture 3 and 4 of Prof. David Silver's series.

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: 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 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).

Video: Record a video in which you explain to viewers, in short, the graphs you found and why you think the graphs are the way you see them. Also argue what granularity works best according to your understanding. Also explain your implementation and how to run the code. Discuss the space-time and accuracy trade-offs you faced.

Save the Model: Once your agent has learnt well, you usually want to save the learning they did, so that the next time you want to play the game, you don't have to re-train an agent from scratch. And since you are using value-based methods and model free learning, to save a model, all you have to do is save your values. Saving in even a txt file works if your space is small. But this is a great opportunity for you to learn how to use .json files!

Game 3: MountainCar

Focus: Discretization of Continuous Spaces in Policy Iteration

Documentation: Link

Pre-Requisites: Lecture 3 of Prof. David Silver's series

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.

Video: Record a video in which you explain to viewers, in short, the graphs you found and why you think the graphs are the way you see them. Also argue what granularity works best according to your understanding. Also explain your implementation and how to run the code. Discuss the space-time and accuracy trade-offs you faced. Explain the grid-map you observe.

Save the Model: Once your agent has learnt well, you usually want to save the learning they did, so that the next time you want to play the game, you don't have to re-train an agent from scratch. And since you are using value-based methods and model free learning, to save a model, all you have to do is save your values. Saving in even a txt file works if your space is small and you can write code to recover the stored model. But this is a great opportunity for you to learn how to use .json files!

Game 4: LunarLander

Focus: Value Approximation Method vs Q-Learning

Documentation: Link

Pre-Requisites: Lectures 3, 4, 5 and 6 of Prof. David's Course.

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.

Video: Record a video in which you explain to viewers, in short, the graphs you found and why you think the graphs are the way you see them. Also explain your implementation and how to run the code. Discuss the space-time and accuracy trade-offs you faced. Explain the heat-map you observed. Compare the performance of the two algorithms with appropriate graphs and metrics.

Save the Model: Once your agent has learnt well, you usually want to save the learning they did, so that the next time you want to play the game, you don't have to re-train an agent from scratch. And since you are using value-based methods and model free learning, to save a model, all you have to do is save your values and you can write code to recover the stored model. Saving in even a txt file works if your space is small. But this is a great opportunity for you to learn how to use .json files!

Submission Details:

Submit any files you need including your trained .json models or any other files you used to store your model in. Other than the code files which you are free to name as per your wish, you are required to submit a few other files. DEADLINE: JUNE 3 EOD.

- A file `references.txt` which contains links to any papers/YouTube-videos/links you used to come up with your approach. Heads up: A lot of YT videos use an out-dated version of Gym so you can't directly copy their stuff, as it will lead to situations where you would be flabbergasted by errors. Instead, read the documentation, attend the tutorial, and ask doubts.
- A video file in .mp4 or .mkv format named `summary.mkv` or `summary.mp4` which follows the video specifications of your question. Include instructions to run your file in this video.
- A `team.txt` file containing the name of the game you chose and your team composition.

Zip all these files together and create `<your-name>+<teammate-name>.zip` or `<your-name>.zip` to submit.

Tutorial:

One of these days, we will have a meet, where I will demonstrate an OpenAI Gym game where I will implement some simple Learning Algorithm. I will also share the code from the session. I plan to demonstrate Frozen-Lake.

It is possible that a lot of things are unclear in this problem statement. For any discrepancies, please reach out to me.