CIS 419/519 Applied Machine Learning
Assignment 6

Due: Fri Apr 17, 2020 11:59pm

Instructions

Read all instructions in this section thoroughly.

Collaboration: Make certain that you understand the collaboration policy described on the course website.

You should feel free to talk to other members of the class in doing the homework. I am more concerned that you learn how to solve the problem than that you demonstrate that you solved it entirely on your own. You may *discuss* the homework to understand the problems and the mathematics behind the various learning algorithms, but you must **write down your solutions yourself**. In particular, **you are not allowed to share problem solutions or your code with any other students.**

We take plagiarism and cheating very seriously, and will be using automatic checking software to detect academic dishonesty, so please don't do it.

You are also prohibited from posting any part of your solution to the internet, even after the course is complete. Similarly, please don't post this PDF file or the homework skeleton code to the internet.

Formatting: This assignment consists of two parts: a problem set and program exercises.

For the problem set, you must write up your solutions electronically and submit it as a single PDF document. We will not accept handwritten or paper copies of the homework. Your problem set solutions must use proper mathematical formatting.

All solutions must be typeset using LaTeX; handwritten solutions of any part of the assignment (including figures or drawings) are not allowed.

Your solutions to the programming exercises must be implemented in python, following the precise instructions included in Part 2. Portions of the programming exercise will be graded automatically, so it is imperative that your code and output follows the specified API. A few parts of the programming exercise asks you to create plots or describe results; these should be included in the same PDF document that you create for the problem set.

Homework Template and Files to Get You Started: Skeleton code is available at:
`https://www.seas.upenn.edu/~cis519/spring2020/homework/hw6_skeleton.ipynb`.

Citing Your Sources: Any sources of help that you consult while completing this assignment (other students, textbooks, websites, etc.) ***MUST*** be noted in your PDF file. This includes anyone you briefly discussed the homework with. If you received help from the following sources, you do not need to cite it: course instructor, course teaching assistants, course lecture notes, course textbooks or other readings.

Submitting Your Solution: You will submit this assignment through Gradescope.

Files to submit:

- **HW6_report.pdf**
- **HW6.ipynb**
- `pi_0.th`
- `pi_1.th`
- `pi_2.th`

Assignment Version 20200409

PART I: PROBLEM SET

1 Reinforcement Learning I (10 pts)

(Adapted from Sutton and Barto, Ex. 3.5) Imagine that you are designing a robot to run a maze. You decide to give it a reward of +1 for escaping the maze and a reward of zero at all other times. The task seems to break down naturally into episodes – the successive runs through the maze – so you decide to treat it as an episodic task, where the goal is to maximize expected total reward $R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$, where T is the final time step of an episode. After running the learning agent for a while, you find that it is showing no improvement in escaping from the maze. Something is going wrong.

Does the reward function effectively communicate the goal to the agent? If not, can you suggest another reward function that will work? If the reward function is fine, what else is going wrong?

2 Reinforcement Learning II (CIS 519 ONLY – 15 pts)

(Adapted from Sutton and Barto, Ex. 3.10.) Consider reinforcement learning in a gridworld where rewards are positive for goals, negative for running into the edge of the world, and zero otherwise.

(a) Are the signs of these rewards important, or only the intervals between them? (e.g., {+1 for a goal, -1 for a collision} versus {-1 for a goal, -3 for a collision})

(b) Provide a formal proof that adding a constant C to all the rewards simply adds a constant, K, to the values of all states, and thus does not affect the relative value of any policies. In your proof, you will find it useful to use the following equations we studied in class for the expected discounted return and value of a state:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad\qquad V^\pi(s) = \mathbb{E}_\pi\Big[R_t \mid s_t = s\Big]$$

(c) What is K in terms of C and gamma?

PART II: PROGRAMMING EXERCISES

In this programming exercise, you will solve a classic problem in control, MountainCar, using two techniques we learned in class: reinforcement learning and imitation learning.

In MountainCar, the agent's actions control a car that is initially located at the bottom of a valley between two mountains, as seen in Fig 1. Success is defined as reaching the flag at the top of the hill on the right. However, the car has limited engine power, so the task is not quite as easy as driving to the right. Instead, a good policy would swing back and forth between the two hills for a while, slowly building up enough momentum to eventually be able to get to the flag. This task was initially developed to stress-test exploration (recall the "exploration-exploitation tradeoff") in reinforcement learning algorithms: success requires that the agent explore states far away from the target.

We will use the OpenAI Gym version of the MountainCar environment, where the agent gets a negative reward of -1 for every step spent in the environment without reaching the target. An episode ends when you have spent 200 steps in the environment without reaching the flag (minimum reward = -200), or when you reach the flag. Please read the python notebook introduction for more details on this environment.

This programming section is aimed at getting you acquainted with OpenAI gym, RL, and imitation learning.

As you complete the programming assignment, the sections of this assignment correspond directly to different parts of the python notebook (hint: search for the word "TODO" in the notebook). Understanding the past homework on neural networks and the Expert Q-learning code that we provide in this section should help you complete this assignment quickly. *Please start early and please ask questions!*

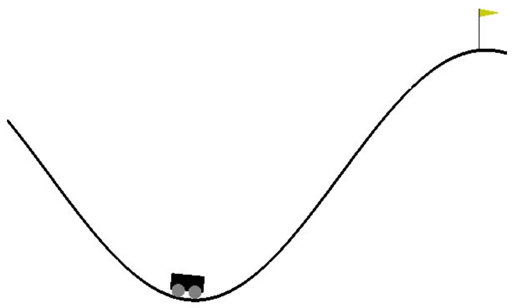

Figure 1: A snapshot of the MountainCar environment

3 Random policy for the MountainCar gym environment [5 pts]

Create a dummy policy by sampling actions uniformly at random from the space of all possible actions permitted by the environment.

To do this, go to the section *Getting to know OpenAI Gym* in the notebook, and fill out the function `dummy_policy(env, num_episodes)`. As inputs it takes in the environment object, and the number of episodes to run the simulation. Run this for 10 episodes and save the rendering of the environment on the last episode. Return the mean reward over 10 episodes, and the frames that you recorded.

In the starter code for this section, we have already created the MountainCar environment and passed it through a wrapper that resizes the rendering to a $100 \times 100 \times 3$ image. Maintain this resizing for this section. **We are resizing the rendering because the original rendering is too large to maintain. Always use the `ResizeObservation` class to resize the rendering, and pass the size of the resizing as a parameter to the constructor**. In the later sections, you are free to resize to any shape you like, but make sure it's not too large.

We have also provided code to generate a video of the episode that you recorded. This corresponds to the frames that you recorded while executing the code. Use this code to generate the video and see what the environment is like.

In your PDF report, explain: (i) what are the action and observation spaces? (how many actions, and how they affect the agent, the dimensionality of the agent's state, and what it represents), (ii) what is the mean reward obtained by the random policy over 10 episodes?

Relevant new operations: `env.action_space`, `env.observation_space`, `env.step(action)`, `env.render()`, `env.reset()`

4 Train a Q-learner and generate expert trajectories [10 pts]

The skeleton code includes a fully implemented Q-learning function `Qlearning()`. We also provide visualization code that you can use to see how the agent is performing. When you execute this function, a $Q(s, a)$ table is estimated and saved as `expert_Q.npy`. We will treat this Q-learner as our "expert" while implementing imitation learning in the rest of this homework. **Please go through this code and understand it completely, it'll help you quickly implement the rest of the sections!**

After training, how does a Q-learning agent select which actions to execute for a given state? This is done in the section *Generate Expert Trajectories*. Implement this action selection policy inside `get_expert_action`. Once implemented, fill in the code for `generate_expert_trajectories` to create a dataset with 100 expert demonstration episodes. You will be implementing this in such a way that you'll be able to store expert trajectories for states and actions, and also images and actions, for visualization. The images here are the renderings of the environment. This is given as a flag in the function argument `save_images`, which is set to False by default. For a given episode of observations and states, you will store a compressed numpy dictionary. This storing code is given to you. Fill in the rest of the function, and use that function to generate expert trajectories and store them.

In your report, please provide the answer to the following questions:

- What is the function `discretize()` attempting to do? Why do you need this function?
- What do you expect will be the effect of varying the `discretization` argument to `discretize()` on Q-learning model performance and model size?

You will need to use `get_expert_action` again in the imitation learning sections, when you will need to get expert actions for a specific observation.

5 Train an imitation policy [30 pts]

You will now implement a state-based imitation learning agent. There are multiple sections to this. Before you get into this, note the section *Launch Imitation Learning* in the skeleton. Here, you will be defining `args = Args()`, an object with which you can carry around all your settings for the code, which could be the intermediary trained model, or hyper-parameters that you use to train, or anything else you desire. Use this object to move quantities around the code. We have given an initial list of args attributes **that you should be using** in your code. You can add new attributes as needed.

5.1 Working with data

The first part of this section is implementing the function `load_initial_data`. It takes in `args` as input. This function is supposed to read the initial expert data that you collected from the directory and return numpy arrays of the observations and actions. You will specify the number of episodes in `args` (i.e if you have a total of 50 episodes recorded, you should be able to select any $x < 50$ episodes).

The second part is implementing the `load_dataset` function. This should return a `torch.utils.data.DataLoader` object that you will use during training.

The next part of this section is implementing `process_individual_observations`. This function does not return a batched dataset, but rather processes individual observations (i.e., convert a float array to a tensor, coupled with any transformations that you might make). You will use this section when you are evaluating your model, where you don't have a batch but are processing states as you step through them one-by-one.

5.2 Defining the network architecture

Implement a small neural network policy as a state-based imitator, using the demonstration dataset collected in the previous section. The skeleton code for this is in the class `StatesNetwork`. This takes in `env` as an argument (so you can extract information about the state space and the action space). The architecture choices are left to you, but remember the policy takes in the state information as input, and must select some action as output.

5.3 Train the model

Implement the training loop in `train_model`. Store any hyper-parameters you want in `args`. **Store your model, i.e., an instance of** `StatesNetwork| inside args.model`. Use practices you learned from the previous homework to aid you in this section.

5.4 Test model performance

The function `test_model` will be used to evaluate how well your model did on one specific episode. It takes in `record_frames` as a flag that you may optionally set to record frames for visual inspection. **This function has been provided to you in the skeleton**, and we will use the same function on Gradescope to grade your submission.

It returns the final position of the car at the end of the episode, whether the episode was a success or not, the reward obtained in this episode, and the frames, if you've recorded them. Use this to evaluate your model performance throughout.

5.5 Main imitation learning method

You will now integrate all the functions from the preceding sections in this method. Take in the initial data, create a dataloader, train the model and test the model performance. Use the `imitate` method, which takes in `args` to first implement a normal imitation learner (i.e only supervised learning). Use the flag `args.do_dagger= False` to denote this.

When you're doing normal imitation learning, the `args.max_dagger_iterations` flag needs to be set to 1 (because you're not doing any Dagger iterations).

Be sure to save your models as necessary.

5.6 Your mission (should you choose to accept it)

Train two separate imitation learning policies:

- `pi_0`, trained on the all 20 expert episodes
- `pi_1`, trained on the first 2 expert episodes.

You should not require more than 2 epochs of training for `pi_0`, or more than 20 epochs for `pi_1`.

For each policy, record its training loss after each iteration, and mean reward and success rate over 5 episodes at the end of each epoch. Include these plots in your report. You might find tensorboard to be a convenient tool for preparing these plots.

To get a full score on this, your agent should have a success rate $> 75\%$ (i.e when your policy is running on the environment, it should succeed at least 75% of the time).

5.7 Evaluation

The code to evaluate the average performance of the model is given in the notebook as `get_average_performance` under the section *Average Performance Metrics*. Use this to evaluate your models. We will use the same function on Gradescope to grade your submissions.

6 Implement DAgger [20 pts]

In this section, you will be implementing DAgger (Dataset Aggregation) on top of normal imitation learning. The necessary sections are given below:

6.1 DAgger execution procedure

First, you will need to fill in the function `execute_dagger`. It takes in `args` as argument. This should return numpy arrays of one episode. This should be the observations seen by your imitation learning agent, but the actions should be that of the expert's. Use the `get_expert_action` function that you used earlier while generating expert trajectories. You should be storing the expert-Q and the discretization inside `args` to use it to get the expert actions.

6.2 Dataset aggregation

In this function `aggregate_dataset`, you will be taking the existing full training observations and actions dataset, and appending on the new one that you just received from `execute_dagger`, before returning the appended dataset.

6.3 Main DAgger imitation method

In the previous section, you implemented the `imitate` method to support vanilla imitation learning. Now, add in support for DAgger using the above two functions you implemented. This method should support both vanilla and DAgger functionalities simultaneously. Use the flag `args.do_dagger=True` to switch between the two functionalities.

6.4 Task

Starting by setting `pi_2 = pi_1`, the imitation policy you have already trained on 2 expert episodes for 2 epochs. Then iteratively improve `pi_2` using DAgger. In particular, collect a single episode with `pi_2`, collect expert annotations on it, add it to `pi_2`'s imitation dataset, and retrain `pi_2`. Repeat this until the size of the imitation dataset reaches 20. In other words, the last version of `pi_2` that you trained should have been trained on a total of 20 episodes (2 expert episodes and 18 DAgger episodes), which is the same as the dataset we used for training `pi_0`. Save the final `pi_2` as `pi_2.th`.

Once again, record its training loss after each iteration, and mean reward and success rate over 5 episodes at the end of each epoch. Include these plots in your PDF report. Include a note on how these compare against the learning curves of `pi_0`.

To get a full score, `pi_2` should have a success rate greater than 90%.

6.5 Evaluation

Once again, use the same evaluation function described in Section 5.7 to evaluate your model. We will use the same function on Gradescope to grade your submissions.