CSCI1410 Fall 2020
Assignment 4: Reinforcement Learning

Code Due Monday, November 16 at 11:59pm ET
Writeup Due Tuesday, November 17 at 11:59pm ET

1 Goals

In this assignment, you will implement several variants of an on-policy reinforcement learning (RL) algorithm called SARSA. SARSA stands for state-action-reward-state-action, as the algorithm updates its Q-function at the current state and action based on its current reward, its next state, and its next action. This Q-function can be a table, indexed by states and actions, or a function of features and actions. In the latter case, the usual representation is as a "linear" function of (potentially non-linear) features and actions, which is approximated by learning weights by which to linearly combine those features.[1]

The success of a reinforcement learning algorithm depends on the settings of its multiple parameters: $\alpha, \gamma, \epsilon, \lambda$, etc. The process of optimizing these many parameters is called *hyper-parameter optimization*. Because the search space over parameter settings is vast (they generally cannot be optimized independently), this process is usually automated. Nonetheless, in this assignment, you will experiment with just a few settings of these parameters, which you will hand tune by inspecting learning curves.

2 Silly Premise

After weeks of a broken phone screen, George has finally decided to drive over to the Apple store to get his screen fixed. But by now, his car is an autonomous reinforcement-learning agent, as are all the taxis on the road! George and his iPhone are thus at the mercy of how well autonomous cars—which are piloted by RL—can drive. As George's car is stuck deep in a valley, he first attempts to get to the Apple store via taxi. Even if successful, he still has to get his own car out of the valley. Your RL algorithms will guide both his taxi and his car.

3 Overview

This assignment has four parts:

1. In the first part, you will implement SARSA and SARSA-λ assuming a tabular representation of the Q-function to help navigate a taxi to its destination.
2. In the second part, you will implement SARSA-λ assuming the Q-function is represented using a linear function approximator with Fourier basis functions to solve the mountain car problem, in which a car is stuck in a deep valley.
3. In the third part, you will answer technical questions about RL. You will also submit learning curves that vary with your hyperparameter settings, and summarize the results of your hyperparameter tuning.

[1] This second use of linear—linear combination—is what gives linear function approximation its name.

4. In the fourth part, you will answer ethics questions about RL.

The coding portions of this assignment use OpenAI's Gym, an open source library with a collection of RL tasks, for use in developing and benchmarking RL algorithms. Gym is written in Python, and includes multiple domains, ranging from the classics (Acrobot, CartPole, Pendulum, Mountain Car, etc.) to Atari games. More details about Gym are provided in Section 8.

4 Taxi: SARSA and SARSA-λ

In this problem, your goal is to help George get from a depot to the Apple Store via taxi. A sample taxi problem is depicted in Figure 1. It consists of a 5×5 grid, with four designated pick-up and drop-off locations, R, G, B, and Y.

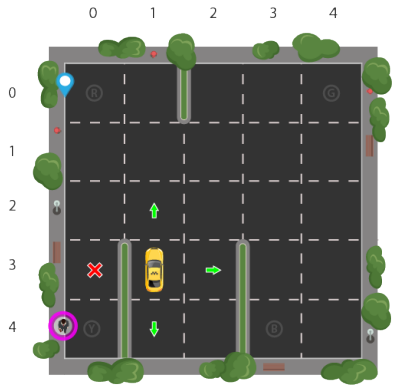

Figure 1: A visualization of the taxi problem, in which a passenger is waiting to be picked up at Y ($(4, 0)$) and dropped off at R ($(0, 0)$).

As the taxi environment is available in Gym, you have been relieved of the task of representing this problem as an MDP: i.e., of identifying the relevant states and actions. Still, before diving into the code, you should spend a few minutes thinking about how you might have tackled this representation problem.

Gym has also defined rewards for this problem somehow. Intuitively, the taxi should earn a high reward when George is picked up at his location, and a very high reward when George is dropped off at the Apple Store. Additionally, it might make sense to penalize the taxi for every step it takes, so that it doesn't drive around in circles wasting gas. Finally, it should also be heavily penalized if it drops George off anywhere other than the Apple Store. All of this information is necessary to formulate the taxi problem as an MDP.

Just as the taxi can learn about its rewards by traversing its environment, it can likewise learn about the transition probabilities. Indeed, both transition probabilities and rewards are unknown. If they were known, the taxi could solve for an optimal policy using dynamic programming. But as this information is unknown, the taxi faces a reinforcement learning problem. Your job is to implement SARSA and SARSA-λ, so the taxi can learn something close to an optimal policy for picking up and dropping off passengers.

4.1 Stencil Code

Here, we briefly describe the primary functions in each file in the taxi stencil code. Please refer to comments in the code itself for more details. We also describe your specific coding tasks.

- `tabular_sarsa.py`: This file contains the `Tabular_SARSA` class, which is the parent class of the tabular SARSA classes that you will be implementing. It contains three data structures: `qtable` for storing state-action values, `etable` for storing eligibility values, and `policy` for storing a policy. It also contains

a function called `learning_policy`, which returns an optimal action, namely one that optimizes the Q-function.[2] **Do not edit this file.**

- `taxi_sarsa.py`: This file contains the `SARSA` class, within which you can find the `learn_policy` function. This function takes as input all the relevant hyperparameters (e.g., $\alpha, \gamma, \epsilon, \lambda$, etc.), and returns a policy, a Q-function, and a sequence of rewards. Here is where you should implement the SARSA update rule. Doing so involves updating the `qtable`.

- `taxi_sarsa_lambda.py`: Like `taxi_sarsa.py`, this file contains another `SARSA` class,[3] within which you can again find a `learn_policy` function. Again, this function takes as input all the relevant hyperparameters (e.g., $\alpha, \gamma, \epsilon, \lambda$, etc.), and returns a policy, a Q-function, and a sequence of rewards. Here is where you should implement the SARSA-λ update rule. Doing so involves updating the `qtable` and the `etable`.

  N.B. Do not edit the `LearningPolicy` functions in `taxi_sarsa.py` or `taxi_sarsa_lambda.py`. [4]

4.2 Evaluation

To test your code, you can run `python taxi_sarsa.py test`, which executes your learned policy on multiple tasks, meaning multiple different locations in which to pick George up and drop him off. This testing code outputs the total rewards earned on each task, and the number of steps taken. On average, your learned policy should complete a task in fewer than 30 steps.

Running `python taxi_sarsa.py test` also generates learning curves, again averaged over multiple runs. The x-axis of these learning curves is the number of learning episodes (i.e., the number of tasks on which the agent was trained), and the y-axis is the total reward earned during that episode, averaged across the various runs. For SARSA, you should plot a learning curve over 1,000 episodes, and for SARSA-λ, 10,000.

You should also spend some, but not too much, time (e.g., 30 minutes) hand-tuning the hyperparameters ($\alpha, \epsilon, \gamma, \lambda$, etc.), before submitting two of your best learning curves with your written handin.

The `test` argument to the `taxi_sarsa.py` file calls the `test_policy` function, which loads your policy from `policy_taxi_sarsa_grading.npy`, because that is the file name the autograder expects. You thus have two choices when doing your own local testing: either change the name of your output files to `policy_taxi_sarsa_grading.npy`, or change `test_policy` to refer to `policy_taxi_sarsa.npy` instead.

N.B. Although the test code does not load the `qtable` (it relies only on the `policy`), the autograder will check that the `policy` you provide can be derived from the `qtable` you provide, so be sure to submit two tables from the same run.

A Note about Grading: The SARSA classes save the policy and state-action values to two files, `policy_taxi_sarsa.npy` and `qvalues_taxi_sarsa.npy`, respectively, for SARSA; likewise, for SARSA-λ. You must submit versions of these files from a reasonable run for grading. To do so, rename the saved files `policy_taxi_sarsa_grading.npy` and `qvalues_taxi_sarsa_grading.npy`, respectively; likewise, for SARSA-λ. The autograder will only grade these renamed files.

5 Mountain-car

George's car is stuck at the bottom of a valley. The Apple store, however, is at the top of the mountain, marked with a green flag. Figure 2 depicts this domain.

[2] In future versions of this assignment, `learning_policy` will be renamed something more accurate like `choose_action`. Apologies for not making the change this year. We did not want to risk breaking the autograder.

[3] In future versions of this assignment, we will not use the same name for two different classes. Apologies for not making the change this year. We did not want to risk breaking the autograder.

[4] In future versions of this assignment, we will not mix-and-match naming conventions. We will follow a style guide, which will mean we either name all functions using a style like `LearningPolicy`, or we will name all functions using a style like `learn_policy`. Apologies for not making this fix this year. We did not want to risk breaking the autograder.

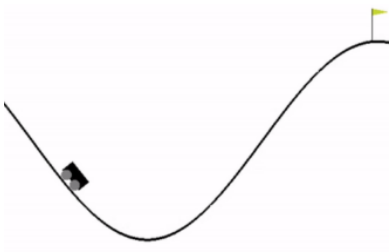

Figure 2: A visualization of the mountain car problem.

The car does not have enough power to just drive up the hill. Instead, it needs to rock back and forth until it gains enough momentum. So, it's actions are move left, move right, and skip (i.e., do nothing).

The car's state space S is continuous. It consists of two real-values, namely the car's position along the x-axis and its velocity. Since $S = \mathbb{R}^2$, we cannot represent the Q-function for mountain car as a table. Instead, we will use a linear function approximator.

We start by representing the state-value function V using a linear function approximator. To do so, we choose a set of n basis functions, $\phi_i : S \to \mathbb{R}^m$, for $i \in \{1, \ldots, n\}$, each of which maps a state $s \in S$ onto an m-dimensional basis. We then define the V-function to be a weighted combination of these basis functions: i.e., for weights $\boldsymbol{w} \in \mathbb{R}^m$,

$$V(s) = \sum_{i=1}^{n} w_i \phi_i(s) \tag{1}$$

Therefore,

$$Q(s,a) = R(s,a) + \gamma \mathbb{E}_{s' \in S}\left[V(s')\right] \tag{2}$$

$$= R(s,a) + \gamma \mathbb{E}_{s' \in S}\left[\sum_{i=1}^{n} w_i \phi_i(s')\right] \tag{3}$$

For this assignment, we recommend the Fourier basis. The univariate nth order Fourier basis consists of $n+1$ basis functions defined as follows: $\phi_i(x) = \cos(\pi i x)$, for $i \in \{0, \ldots, n\}$ and $x \in S$. For a multi-dimensional state space, such as that of the mountain car problem, where the number of dimensions $d = 2$, the nth order Fourier basis consists of $(n+1)^d$ basis functions defined as follows: $\phi_i(\boldsymbol{x}) = \cos(\pi \boldsymbol{c}^i \cdot \boldsymbol{x})$, for all $i \in \{0, \ldots, (n+1)^d\}$ and $\boldsymbol{x} \in S$, where the $\boldsymbol{c}$ vectors range over all d-dimensional vectors in $\{0, \ldots, n+1\}^d$. For more details and motivation for the Fourier basis, we refer you to a paper by George himself.

Like the taxi, the mountain car would like to execute an optimal policy. Again, the rewards and transitions are unknown; only a simulator is provided (within Gym). So the mountain car faces a reinforcement learning problem. Your job is to implement SARSA-λ with function approximation, so the mountain car can learn something close to an optimal policy. Pseudocode for SARSA-λ can be found in this online RL textbook.

5.1 Stencil Code

Here, we briefly describe the primary functions in `mountain_car_sarsa_fourier.py`. Please refer to comments in the code itself for more details. We also describe your specific coding tasks.

Please do not modify any functions except those you are explicitly instructed to modify.

1. `normalize_state`: This function normalizes each dimension of the state (position and velocity) to lie within the range $[0, 1]$. Normalization is not strictly necessary, as the weights may be adjusted to account for any differences in magnitude among the various state dimensions, but it does simplify and often speed up the learning process.

2. `phi`: This function takes in a normalized state and computes its value, given a basis. **You need to fill in this function.**

3. `create_multipliers`: This function creates the Fourier basis coefficients (i.e., the $\boldsymbol{c}$ vectors). **You need to fill in this function.**

4. `action_value`: This function returns an action value, given a state-action pair.

5. `learning_policy` and `test_policy`: These functions return an optimal action, given a state, with and without exploration, respectively. If an agent is no longer learning, it need not need explore any further either; hence, a separate function.[5]

6. `SARSA_Learning`: This function should update the weights associated with the basis functions using the SARSA-λ algorithm. Some of the function is filled in for you, to interface to the rest of the code, but **you need to fill in the rest.**

7. `SARSA_Test`: This function uses the weights learned or saved to act over a specified number of episodes. You can run this function to test your policy before you submit it.

5.2 Evaluation

To test your code, you can run `python mountain_car_sarsa_fourier.py test`, which executes your learned policy multiple times. This testing code outputs the total rewards earned on each execution, and the number of steps taken. On average, your learned policy should complete the task in fewer than 400 steps.

Running `python mountain_car_sarsa_fourier.py test` also generates learning curves,[6] again averaged over multiple runs. The x-axis of these learning curves is the number of learning episodes (i.e., the number of tasks on which the agent was trained), and the y-axis is the total reward earned during that episode, averaged across the various runs. You should plot a learning curve over 10,000 episodes.

You should also spend some, but not too much, time (e.g., 30 minutes) hand-tuning the hyperparameters ($\alpha, \epsilon, \gamma, \lambda$, etc.), before submitting one of your best learning curves with your written handin.

Note that order of the Fourier basis is an important parameter. We initialized this value to 3. It should remain odd (e.g., 1, 5, 7, etc.), but you should otherwise feel free to experiment with alternatives.

The `test` argument to the `mountain_car_sarsa_fourier.py`
file loads your policy from `policy_mountain_car_saved_weights.npy`.
This time, that is not the name the autograder expects! (Apologies for the inconsistencies in this assignment.)

A Note about Grading: The `SARSA_Learning` function saves the learned weights to a file called `mountain_car_saved_weights.npy`. You must submit a version of this file from a reasonable run for grading. To do so, rename the saved file `mountain_car_saved_weights_grading.npy`. The autograder will only grade a file by this latter name.

6 Written Questions

Answer the following questions in clearly labeled sections.

1. Describe any possible state space representation for the taxi environment (even the one used in Gym).

[5] This is poor programming practice. Can you see why? Because there should not be two functions where one (`test_policy`) is a special case of the other (`learn_policy`). In future versions of this assignment, `test_policy` will be defined as `learn_policy` with epsilon fixed at 0, to prevent exploration.

[6] Actually, it does not—unfortunately—but it should. For extra credit, please go ahead and adjust the test code so that it does generate plots, so that you can more easily tune the hyperparameters of your implementation.

2. Submit your learning curves, and explain what you gleaned from them. What settings of the hyper-parameters did you find worked best in the taxi problem? And what about in mountain car?[7] What happened when you changed the order of the Fourier basis?

3. Recall that Arthur Samuel used machine learning back in 1959 to train his checker-playing program. More relevant this assignment, in 1992 Gerry Tesauro, used *reinforcement learning* with (non-linear) function approximation in his award-winning backgammon-playing program, TD-Gammon.[8] In both cases, the programs improved their play (i.e., learned) by playing against themselves.

   How can you formulate learning to play a two-player game, like backgammon or Go, as an MDP? What are the states, the actions, the rewards, and the transitions? And why does it make more sense to formulate such a problem as a reinforcement learning problem, in which rewards and transitions are unknown to the learning agent, rather than attempt to solve such an MDP via dynamic programming?

7 Ethics Questions

One of the most important and quickly-developing uses of reinforcement learning is for self-driving vehicles. These autonomous vehicles stand to revolutionize industries like ride-sharing, trucking, and package delivery, potentially automating over four million currently existing jobs. Read this short article about automated decision making in self-driving cars, and then answer the following questions.

1. It will someday happen that an AI agent, like a driverless car, will need to make a decision that involves an ethical tradeoff. Who should determine how these ethical decisions are made? Consider the impacts of handing such a responsibility to manufacturers, to governments, to industry standards bodies, etc. Should machine ethics studies like the Moral Machine study be conducted so that AI agents mirror as best as possible the ethical preferences of a culture or a region?

2. Someday AI agents will be used in critical applications, like the military, healthcare, and transportation. How safe should driverless cars be before they are approved by governments? Should they drive as well as the average human driver? Better than the best human driver? What standard should be applied?

3. Who should be held responsible when an AI agent makes a mistake and causes a human bodily harm? In the case of self-driving cars, consider the impact of designating liability to the passenger, the individual software engineer or the software team, the company as a whole, the government, or some other entity. Similarly, how should such designations extend to AI agents used in other critical application areas, such as healthcare.

8 Gym Installation

There are several packages that are needed for this project. Some of the necessary packages are: `gym`, `matplotlib`, and `numpy`. If you use the course virtual environment you shouldn't need to install anything, but if you are working locally you may need to install these by running `pip install 'gym[all]'`. Additionally, here are two tutorials that explain how to use AI gym to solve the taxi and mountain car problems.[9]

9 Grading

As usual, you can refer to `rubric.txt` to see how heavily each part of this assignment is weighted.

[7] For extra credit, you may submit learning curves for mountain car as well as the taxi problem. You are also permitted to post your plotting code on Piazza, as that would be a help to your fellow students. Please let Amy know if you do this, so that she can award you good Samaritan points (and can recommend you as a TA for next year!).

[8] Now you know what "TD" means! Hurray!

[9] We cannot vouch for their accuracy. There are many tutorials out there. If you find better ones, please share on Piazza.

We will autograde the output of your SARSA and SARSA-λ implementations for the taxi problem using the files you submit, containing your learned policy and Q-values. We will simulate your policy on 10 random tasks, and award you full points if the average number of steps to completion is under 30.

We will also autograde the output of your SARSA-λ implementation for mountain car using the file you submit, containing your learned weights. Again, we will simulate your policy 10 times, but as mountain car is a harder problem, we will award you full points if the average number of steps to completion is under 400.

In all cases, the time limit for each test run is 30 seconds. In other words, if ever your policy does not complete a task within this time limit, you will score 0 on that test run.

In addition to the aforementioned autograding, we will also verify that your submitted code corresponds to your submitted data files and learning curves. **If we find that the data files and/or the learning curves that you submit could not have been generated by your code, you will earn a zero for the entire assignment.**

Note: After generating your `.npy` files, you need to rename them by appending `_grading.npy` to the end of each file name. Incorrectly named files will not be graded.

10 Install and Handin Instructions

To install, run `cs1410_install RL` in `~/course/cs1410/`.

To handin, run `cs1410_handin RL` in `~/course/cs1410/RL/`, which should contain your code and your five saved data files.

In addition, please submit the written portion of the assignment via Gradescope.