XCS221 Assignment 3 — Controlling Mountain Car

Due Sunday, December 14 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Also note that your answers should be in order and clearly and correctly labeled to receive credit. Be sure to submit your final answers as a PDF and **tag all pages correctly when submitting to Gradescope**.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

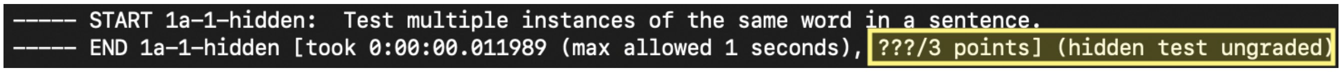
Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

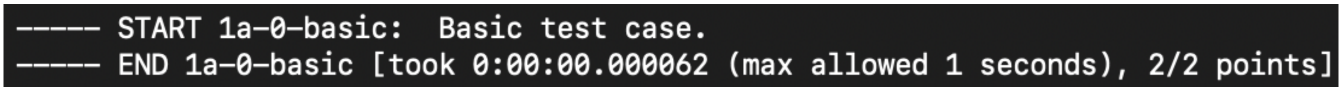
All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:



```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

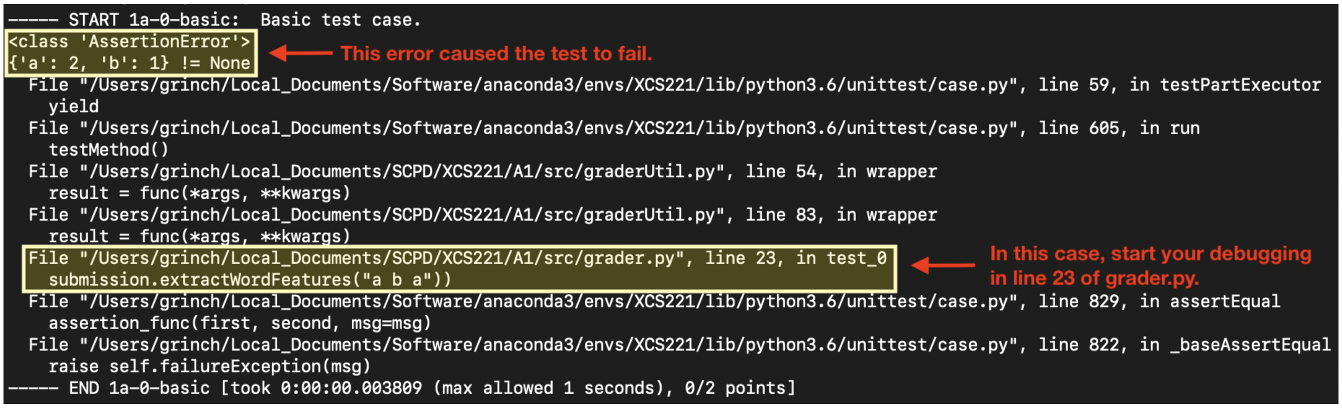
Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:



```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:



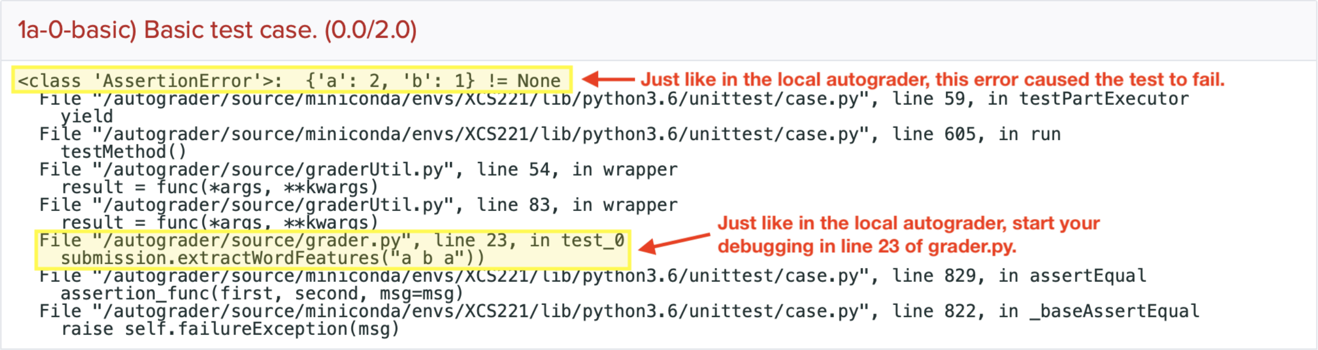
```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

This error caused the test to fail.

In this case, start your debugging in line 23 of grader.py.

Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

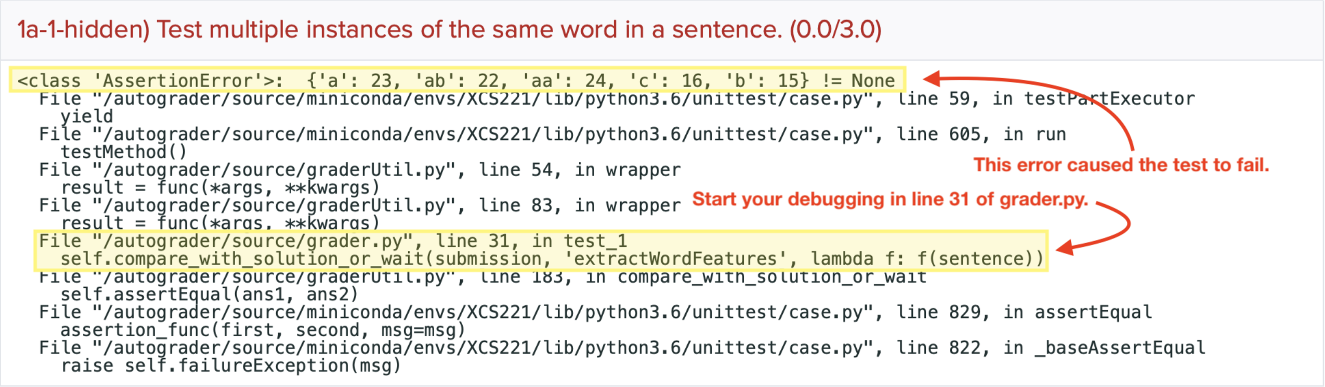


1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.



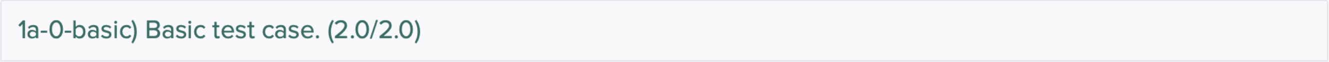
1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

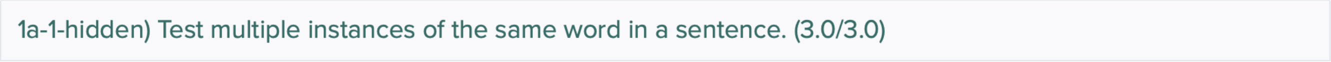
This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.



1a-0-basic) Basic test case. (2.0/2.0)



1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Introduction

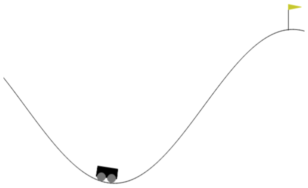


Figure 1: Mountain Car

Markov decision processes (MDPs) can be used to model situations with uncertainty (which is most of the time in the real world). In this assignment, you will implement algorithms to find the optimal policy, even you know the transitions and rewards (value iteration) and when you don't (reinforcement learning). You will use these algorithms on Mountain Car, a classic control environment where the goal is to control a car to go up a mountain.

1. Value Iteration

In this problem, you will perform value iteration updates manually on a basic game to build your intuitions about solving MDPs. The set of possible states in this game is $\text{States} = \{-2, -1, 0, +1, +2\}$ and the set of possible actions is $\text{Actions}(s) = \{a_1, a_2\}$ for all states that are not end states. The starting state is 0 and there are two end states, -2 and $+2$. Recall that the transition function $T : \text{States} \times \text{Actions} \to \Delta(\text{States})$ encodes the probability of transitioning to a next state s' after being in state s and taking action a as $T(s'|s, a)$. In this MDP, the transition dynamics are given as follows:

$\forall i \in \{-1, 0, 1\} \subseteq \text{States}$,

- $T(i - 1|i, a_1) = 0.8$ and $T(i + 1|i, a_1) = 0.2$
- $T(i - 1|i, a_2) = 0.7$ and $T(i + 1|i, a_2) = 0.3$

Think of this MDP as a chain formed by states $\{-2, -1, 0, +1, +2\}$. In words, action a_1 has a 80% chance of moving the agent backwards in the chain and a 20% chance of moving the agent forward. Similarly, action a_2 has a 70% of sending the agent backwards and a 30% chance of moving the agent forward. We will use a discount factor $\gamma = 1$. The reward function for this MDP is

$$\text{Reward}(s, a, s') = \begin{cases} 10 & \text{if } s' = -2, \\ 50 & \text{if } s' = +2, \\ -5 & \text{otherwise.} \end{cases}$$

(a) **[3 points (Written)]**

What is the value of $V_i^\star(s)$ for each $s \in \text{States}$ after each iteration $i = \{1, 2\}$ of value iteration? Recall that $\forall s \in \text{States}$, $V_0^\star(s) = 0$ and, for any i, end state s_{end}, we have $V_i^\star(s_{\text{end}}) = 0$.

What we expect: The $V_i^\star(s)$ of all 5 states after each iteration. In total, 10 values should be reported.

(b) **[1 point (Written)]**

Using $V_2^\star(\cdot)$, what is the corresponding optimal policy $\pi^\star$ for all non-end states?

What we expect: Optimal policy $\pi^\star(s)$ for each non-end state.

2. Transforming MDPs

In computer science, the idea of a reduction is very powerful: say you can solve problems of type A, and you want to solve problems of type B. If you can convert (reduce) any problem of type B to type A, then you automatically have a way of solving problems of type B potentially without doing much work! We saw an example of this for search problems when we reduce A* to UCS. Now let's do it for solving MDPs.

(a) [**4 points (Written)**]

Suppose we have an MDP with states S and a discount factor $\lambda < 1$, but we have an MDP solver that can only solve MDPs with discount factor of 1. How can we leverage the MDP solver to solve the original MDP?

Let us define a new MDP with states $S' = S \cup \{o\}$, where o is a new state. Let's use the same actions ($A'(s) = A(s)$), but we need to keep the discount $\lambda' = 1$. Your job is to define new transition probabilities $T'(s, a, s')$ and rewards $R'(s, a, s')$ in terms of the old MDP such that the optimal values $V_{\text{opt}}(s)$ for all $s \in S$ are equal under the original MDP and the new MDP.

Hint: If you're not sure how to approach this problem, go back to the first MDP lecture and read closely the slides on convergence.

3. Value Iteration on Mountain Car

Now that we have gotten a bit of practice with general-purpose MDP algorithms, let's use them for some control problems. Mountain Car is a classic example in robot control [1] where you try to get a car to the goal located on the top of a steep hill by accelerating left or right. We will use the implementation provided by The Farama Foundation's Gymnasium, formerly OpenAI Gym.

The state of the environment is provided as a pair (position, velocity). The starting position is randomized within a small range at the bottom of the hill. At each step, the actions are either to accelerate to the left, to the right, or do nothing, and the transitions are determined directly from the physics of a frictionless car on a hill. Every step produces a small negative reward, and reaching the goal provides a large positive reward.

To get the feel for the environment, test with an untrained agent which takes random action at each step and see how it performs.

```
python mountaincar.py --agent naive
```

You will see the agent struggling, not able to complete the task within the time limit. In this assignment, you will train this agent with different reinforcement learning algorithms so that it can learn to climb the hill. As the first step, we have designed two MDPs for this task. The first uses the car's continuous (position, velocity) state as is, and the second discretizes the position and velocity into bins and uses indicator vectors.

Carefully examine `ContinuousGymMDP` and `DiscreteGymMDP` classes in `util.py` and make sure you understand.

If we want to apply value iteration to the `DiscreteGymMDP` (think about why we can't apply it to ContinuousGymMDP), we require the transition probabilities $T(s, a, s')$ and rewards $R(s, a, s')$ to be known. But oftentimes in the real world, T and R are unknown, and the gym environments are set up in this way as well, only interfacing through the `.step()` function. One method to still determine the optimal policy is model-based value iteration, which runs Monte Carlo simulations to estimate $\hat{T}$ and $\hat{R}$, and then runs value iteration. This is an example of model-based RL. Examine `RLAlgorithm` in `util.py` to understand the `getAction` and `incorporateFeedback` interface and peek into the `simulate` function to see how they are called repeatedly when training over episodes.

(a) **[5 points (Coding)]**

As a warm up, we will start with implementing value iteration as you learned in lectures, and run it on the number line MDP from Problem 1.

Complete `valueIteration` function in `submission.py`.

(b) **[9 points (Coding)]**

Now in `submission.py`, implement `ModelBasedMonteCarlo` which runs `valueIteration` every `calcValIterEvery` steps that the RL agent operates. The `getAction` method controls how we use the latest policy as determined by value iteration, and the `incorporateFeedback` method updates the transition and reward estimates, calling value iteration when needed. Implement the `getAction` and `incorporateFeedback` methods for `ModelBasedMonteCarlo`.

(c) **[2 points (Written)]**

Run `python train.py --agent value-iteration` to train the agent using model based value iteration you implemented above and see the plot of reward per episode. Comment on the plot, and discuss when might model based value iteration perform poorly. You can also run `python mountaincar.py --agent value-iteration` to visually observe how the trained agent performs the task now.

What we expect: 2-3 sentences describing the plot and discussion of when model based value iteration may fail.

4. **Q-Learning Mountain Car**

In the previous question, we've seen how value iteration can take an MDP which describes the full dynamics of the game and return an optimal policy, and we've also seen how model-based value iteration with Monte Carlo simulation can estimate MDP dynamics if unknown at first and then learn the respective optimal policy. But suppose you are trying to control a complex system in the real world where trying to explicitly model all possible transitions and rewards is intractable. We will see how model-free reinforcement learning can nevertheless find the optimal policy.

(a) **[9 points (Coding)]**

For a discretized MDP, we have a finite set of `(state, action)` pairs. We learn the Q-value for each of these pairs using the Q-learning update learned in class. In the `TabularQLearning` class, implement the `getAction` method which selects action based on `explorationProb`, and the `incorporateFeedback` method which updates the Q-value given a (state, action) pair.

(b) **[4 points (Coding)]**

For Q-learning in continuous states, we need to use function approximation. The first step of function approximation is extracting features given state. Feature extractors of different complexities work well with different problems: linear and polynomial feature extractors that work well with simpler problems may not be suitable for other problems. For the mountain car task, we are going to use a Fourier feature extractor. As background, any continuous periodic function can be approximated as a Fourier Series

$$f(x) = \frac{a_0}{2} + \sum_{j=1}^{n} [a_j \cos(2\pi jx/T) + b_j \sin(2\pi jx/T)]$$

with a_j and b_j sequences of coefficients determined by integrating f. To apply this to Q-learning with function approximation, we want the learned weights w to emulate a_j and b_j and the output of ϕ to provide the basis of varying sinusoid periods as seen in $\cos(2\pi jx/T)$ for $j = 1, 2, \ldots, n$. Thus, for state $s = [s_1, s_2, \ldots, s_k]$, action a, and maximum coefficient c, the feature extractor ϕ is:

$$\begin{aligned}\phi(s, a, c) &= [\cos(0), \cos(s_1), \ldots, \cos(s_k), \cos(2s_1), \cos(s_1 + s_2), \ldots, \cos(cs_1 + cs_2 + \ldots + cs_k)] \\ &= \left\{ \cos\left(\sum_{i=1}^{k} c_i s_i \right) : \forall c_1 \in \mathscr{C}, \ldots, c_k \in \mathscr{C} \right\}\end{aligned}$$

where $\mathscr{C} = \{0, 1, \ldots, c\}$ is the set of non-negative integers from 0 to c. Note that $\phi(s, a, c) \in \mathbb{R}^{(c+1)^k}$.

Implement `fourierFeatureExtractor` in `submission.py`. Looking at `util.polynomialFeatureExtractor` may be useful for familiarizing oneself with numpy.

(c) **[9 points (Coding)]**

Now we can find the Q-value of each `(state, action)` by multiplying the extracted features from this pair with weights. Unlike the tabular Q-learning in which Q-values are updated directly, with function approximation, we are updating weights associated with each feature. Using `fourierFeatureExtractor` from the previous part, complete the implementation of `FunctionApproxQLearning`.

(d) **[2 points (Written)]**

Run `python train.py --agent tabular` and `python train.py --agent function-approximation` to see the plots for how the TabularQLearning and FunctionApproxQLearning work respectively, and comment on the plots. You should expect to see that tabular Q-learning performs better than function approximation Q-learning on this task. What might be some of the reasons? You can also run `python mountaincar.py --agent tabular` and `python mountaincar.py --agent function-approximation` to visualize the agent trained with continuous and discrete MDP respectively.

What we expect: Plots and 2-3 sentences comparing the performances of `TabularQLearning` and `FunctionApproxQLearning`, discussing possible reasons as well.

(e) **[2 points (Written)]**

What are some possible benefits of function approximation Q-learning for mountain car or other environments? Consider the situations where the exploration period is limited, where the state space is very high dimensional and difficult to explore, and/or you have space constraints.

What we expect: 2-3 sentences discussing why `FunctionApproxQLearning` can be better in various scenarios.

5. **Safe Exploration**

We learned about different state exploration policies for RL in order to get information about `(s,a)`. The method implemented in our MDP code is epsilon-greedy exploration, which balances both exploitation (choosing the action a that maximizes $\hat{Q}_{\text{opt}}(s,a)$) and exploration (choosing the action a randomly).

$$\pi_{\text{act}}(s) = \begin{cases} \arg\max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s,a) & \text{probability } 1-\epsilon \\ \text{random from Actions}(s) & \text{probability } \epsilon \end{cases}$$

In real-life scenarios when safety is a concern, there might be constraints that need to be set in the state exploration phase. For example, robotic systems that interact with humans should not cause harm to humans during state exploration. [2] in RL is thus a critical research question in the field of AI safety and human-AI interaction.

Assume there are harmful consequences for the driver of a mountain-car if the car exceeds a certain velocity. One very simple approach of constrained RL is to restrict the set of potential actions that the agent can take at each step. We want to apply this approach to restrict the states that the agent can explore in order to prevent reaching unsafe speeds.

(a) **[1 point (Written)]**

The implementation of the Mountain Car MDP you built in the previous questions actually already has velocity constraints in the form of a `self.max_speed` parameter. Read through OpenAI Gym's Mountain Car implementation and explain how `self.max_speed` is used.

What we expect: One sentence on how `self.max_speed` parameter is used in `custom_mountain_car.py`

(b) **[1 point (Written)]**

Run Function Approximation Q-Learning without the `max_speed` constraint by running `python train.py --agent function-approximation --max_speed=100000`. We are changing `max_speed` from its original value of 0.07 to a very large float to approximate removing the constraint entirely. Notice that running the MDP unconstrained doesn't really change the output behavior. Explain in 1-2 sentences why this might be the case.

What we expect: One sentence explaining why the Q-Learning result doesn't necessarily change.

(c) **[2 points (Written)]**

Consider a different way of setting the constraint where you limit the set of actions the agent can take in the action space. In ConstrainedQLearning, implement constraints on the set of actions the agent can take each step such that `velocity_(t+1) < velocity_threshold`. Remember to handle the case where the set of valid actions is empty. Below is the equation for calculating the velocity at time step t+1.

$$\text{velocity}_{t+1} = \text{velocity}_t + (\text{action} - 1) * \text{force} - \cos(3 * \text{position}_t) * \text{gravity}$$

We've determined that in the mountain car environment, a velocity of 0.065 is considered unsafe. After implementing the velocity constraint, run grader test 5c-0-helper to examine the optimal policy for two continuous MDPs running Q-Learning (one with max_speed of 0.065 and the other with a very large max_speed constraint). Provide 1-2 sentences explaining why the output policies are now different.

What we expect: Complete the implmentation of `ConstrainedQLearning` in `submission.py`, then run `python grader.py 5c-0-helper`. Include 1-2 sentences explaining why there is a difference between the two scenarios.

(d) **[1 point (Written)]**

The concept of safe exploration in RL extends to other real-world contexts where some amount of learning through trial and error might be expected. One example of this is clinical research: conducting epsilon-greedy exploration examining the effects of a drug that has serious or fatal side effects may harm a lot of people and be ethically questionable. For example, one of the most notorious cases of abuse in clinical trials was the Tuskegee Syphilis Study conducted between 1932 and 1972 by the US Public Health Service and the CDC. During the study, 400 African American men with Syphilis were not treated so that researchers could study the progression of the disease, costing many of these individuals their lives. As a result, ethical guidelines have been put in place to guide research involving human-beings, like clinical research. Many of these guidelines apply to other research contexts beyond medicine. In many ways, these guardrails for science are similar to the constraints you set for Mountain Car to keep its velocity within a safe range.

The Belmont report, one of the list of guidelines for conducting human-subjects research that arose after the Tuskegee Syphilis Study, lists three principles: Respect for Persons, Beneficence, and Justice. In 2012, the US Department of Homeland Security applied these principles to describe how to conduct ethical information and communication technology research in the Menlo Report. The principles are as follows:

- **Respect for persons:** Participation as a research subject is voluntary, and follows from informed consent; Treat individuals as autonomous agents and respect their right to determine their own best interests; Respect individuals who are not targets of research yet are impacted; Individuals with diminished autonomy, who are incapable of deciding for themselves, are entitled to protection.
- **Beneficence:** Do not harm; Maximize probably benefits and minimize probable harms; Systematically assess both risk of harm and benefit.
- **Justice:** Each person deserves equal consideration in how to be treated, and the benefits of research should be fairly distributed according to individual need, effort, societal consideration, and merit; Selection of subjects should be fair, and burdens should be allocated equitably across impacted subjects.
- **Respect for Law and Public Interest:** Engage in legal due diligence; Be transparent in methods and results; Be accountable for actions.

Imagine a company is working on testing and improving their autonomous driving technology through RL. They want to run their experiments on streets in residential neighborhoods. Describe an MDP (specifically define States, Actions(s), and Reward(s, a, s')) and a state exploration policy that would potentially cause harm to humans. Which of the 4 Menlo Report principles might be violated by running this experiment and why? Explain in 1-3 sentences.

What we expect: Definition of an MDP's States, Actions(s), and Reward(s, a, s'), a state exploration policy, and a relevant Menlo Report Principle that might be violated with an explanation.

References

[1] Moore. efficient memory-based learning for robot control. 1990.

[2] Alex Ray, Joshua Achiam, and Dario Amodei. Openai. benchmarking safe exploration in deep reinforcement learning. 2018.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.a

1.b

2.a

3.c

4.d

4.e

5.a

5.b

5.c

5.d