10301/601 Homework 8 (Written)

Joong Ho Choi

TOTAL POINTS

43.167 / 45

QUESTION 1

1 Latex Bonus Point 1 / 0

✓ **+ 1 pts** **$$\LaTeX$$**

+ 0 pts Not all $$\LaTeX$$

QUESTION 2

2.1 Synchronous Value Iteration 6 pts

2.1 SJ 0.334 / 0.334

✓ **- 0 pts** **Correct: 0**

- 0.334 pts Blank/Incorrect

2.2 SL 0.333 / 0.333

✓ **- 0 pts** **Correct: 0**

- 0.333 pts Incorrect

2.3 SK 0.333 / 0.333

✓ **- 0 pts** **Correct: 80**

- 0.333 pts Blank/Incorrect

2.4 SE 0.333 / 0.333

✓ **- 0 pts** **Correct: 0**

- 0.333 pts Incorrect

2.5 SF 0.333 / 0.333

✓ **- 0 pts** **Correct: 0**

- 0.333 pts Blank/Incorrect

2.6 SH 0.333 / 0.333

✓ **- 0 pts** **Correct: 40**

- 0.333 pts Blank/Incorrect

2.7 SB 0.333 / 0.333

✓ **- 0 pts** **Correct: 0**

- 0.333 pts Blank/Incorrect

2.8 SC 0.334 / 0.334

✓ **- 0 pts** **Correct: 0**

- 0.334 pts Blank/Incorrect

2.9 SD 0.334 / 0.334

✓ **- 0 pts** **Correct: 0**

- 0.334 pts Blank/Incorrect

2.10 SJ 0 / 0.333

- 0 pts Correct: down

✓ **- 0.333 pts** **Blank/Incorrect**

2.11 SH 0.333 / 0.333

✓ **- 0 pts** **Correct: up**

- 0.333 pts Blank/Incorrect

2.12 SF 0.333 / 0.333

✓ **- 0 pts** **Correct: right**

- 0.333 pts Blank/Incorrect

2.13 SE 0.334 / 0.334

✓ **- 0 pts** **Correct: terminal**

- 0.334 pts Blank/Incorrect

2.14 SK 0.333 / 0.333

✓ **- 0 pts** **Correct: right**

- 0.333 pts Blank/Incorrect

2.15 SL 0.334 / 0.334

✓ **- 0 pts** **Correct: terminal**

- 0.334 pts Blank/Incorrect

2.16 SB 0.334 / 0.334

✓ **- 0 pts** **Correct: down**

- 0.334 pts Blank/Incorrect

2.17 SC **0.333 / 0.333**

✓ **- 0 pts Correct: down**

- 0.333 pts Blank/Incorrect

2.18 SD **0 / 0.333**

- 0 pts Correct: down

✓ **- 0.334 pts Blank/Incorrect**

QUESTION 3

2.2 Asynchronous Value Iteration 12 pts

3.1 SJ **0.334 / 0.334**

✓ **- 0 pts Correct: 57.6**

- 0.334 pts Blank/Incorrect

3.2 SL **0.333 / 0.333**

✓ **- 0 pts Correct: 0 or Blank**

- 0.333 pts Incorrect

3.3 SK **0.333 / 0.333**

✓ **- 0 pts Correct: 80**

- 0.333 pts Blank/Incorrect

3.4 SE **0.333 / 0.333**

✓ **- 0 pts Correct: 0/Blank**

- 0.333 pts Incorrect

3.5 SF **0.334 / 0.334**

✓ **- 0 pts Correct: 31.472**

- 0.334 pts Blank/Incorrect

3.6 SH **0.333 / 0.333**

✓ **- 0 pts Correct: 40**

- 0.333 pts Blank/Incorrect

3.7 SB **0.333 / 0.333**

✓ **- 0 pts Correct: 24.5**

- 0.333 pts Blank/Incorrect

3.8 SC **0.333 / 0.333**

✓ **- 0 pts Correct: 20.736**

- 0.333 pts Blank/Incorrect

3.9 SD **0.334 / 0.334**

✓ **- 0 pts Correct: 28.8**

- 0.334 pts Blank/Incorrectr

3.10 SJ **0.333 / 0.333**

✓ **- 0 pts Correct: right**

- 0.333 pts Blank/Incorrect

3.11 SK **0.334 / 0.334**

✓ **- 0 pts Correct (Right)**

- 0.334 pts Incorrect/blank

3.12 SL **0.333 / 0.333**

✓ **- 0 pts Correct (terminal)**

- 0.333 pts Incorrect/blank

3.13 SE **0.333 / 0.333**

✓ **- 0 pts Correct (terminal)**

- 0.333 pts Incorrect/blank

3.14 SF **0.333 / 0.333**

✓ **- 0 pts Correct (up)**

- 0.333 pts incorrect/blank

3.15 SH **0.334 / 0.334**

✓ **- 0 pts Correct (up)**

- 334 pts Incorrect/blank

3.16 SB **0.334 / 0.334**

✓ **- 0 pts Correct (up)**

- 0.334 pts blank/incorrect

3.17 SC **0.333 / 0.333**

✓ **- 0 pts Correct (right)**

- 0.333 pts Incorrect/blank

3.18 SD **0.333 / 0.333**

✓ **- 0 pts Correct (up)**

- 0.333 pts Incorrect/blank

3.19 SJ **0.333 / 0.333**

✓ **- 0 pts Correct (82.2/82.1865)**

- **0.333 pts** Incorrect/blank

3.20 SK **0.334 / 0.334**

✓ **- 0 pts Correct (97.5/97.4586)**

- **0.334 pts** Incorrect/blank

3.21 SL **0.333 / 0.333**

✓ **- 0 pts Correct (0/blank/terminal)**

- **0.333 pts** Incorrect

3.22 SE **0.334 / 0.334**

✓ **- 0 pts Correct (0/blank/terminal)**

- **0.334 pts** incorrect

3.23 SF **0.333 / 0.333**

✓ **- 0 pts Correct (53.9/53.9051)**

- **0.333 pts** Incorect/blank

3.24 SH **0.333 / 0.333**

✓ **- 0 pts Correct (48.7 / 48.7293)**

- **0.333 pts** Incorrect/blank

3.25 SB **0.334 / 0.334**

✓ **- 0 pts Correct (46.3 / 46.2815)**

- **0.334 pts** Incorrect/blank

3.26 SC **0.333 / 0.333**

✓ **- 0 pts Correct (38.4 / 38.4399)**

- **0.333 pts** Incorrect/blank

3.27 SD **0.333 / 0.333**

✓ **- 0 pts Correct (42/41.9723)**

- **0.333 pts** Incorrect/blank

3.28 SJ **0.333 / 0.333**

✓ **- 0 pts Correct (right)**

- **0.333 pts** Incorrect/blank

3.29 SK **0.333 / 0.333**

✓ **- 0 pts Correct (right)**

- **0.333 pts** Incorrect/blank

3.30 SL **0 / 0.334**

- **0 pts** Correct (terminal)

✓ **- 0.334 pts Incorrect/blank**

3.31 SE **0 / 0.333**

- **0 pts** Correct (terminal)

✓ **- 0.333 pts Incorrect/blank**

3.32 SF **0.333 / 0.333**

✓ **- 0 pts Correct (up)**

- **0.333 pts** Incorrect/blank

3.33 SH **0.334 / 0.334**

✓ **- 0 pts Correct (up)**

- **0.334 pts** Incorrect/blank

3.34 SB **0.334 / 0.334**

✓ **- 0 pts Correct (up)**

- **0.334 pts** Incorrect/blank

3.35 SC **0.333 / 0.333**

✓ **- 0 pts Correct (left)**

- **0.333 pts** Incorrect/blank

3.36 SD **0.333 / 0.333**

✓ **- 0 pts Correct (up)**

- **0.333 pts** Incorrect/blank

QUESTION 4

3 Q-Learninig 9 pts

4.1 3.1 **1 / 1**

✓ **- 0 pts Correct (All of the above)**

- **0.5 pts** Only selected 2 options
- **0.25 pts** Only selected 3 options
- **0.75 pts** Only selected 1 option
- **1 pts** blank

4.2 3.2 **1 / 1**

✓ **- 0 pts Correct (0)**

- **1 pts** Incorrect/blank

4.3 3.3 **1 / 1**

✓ **- 0 pts Correct (0)**

- 1 pts Incorrect/blank

4.4 3.4 **1 / 1**

✓ **- 0 pts Correct (all of the above)**

- 0.75 pts only selected 1 option

- 0.25 pts Only selected 3 options

- 1 pts Blank

- 0.5 pts Only selected 2 options

4.5 3.5 **1 / 1**

✓ **- 0 pts Correct (50)**

- 1 pts Blank/incorrect

4.6 3.6 **1 / 1**

✓ **- 0 pts Correct (5)**

- 1 pts Incorrect/blank

4.7 3.7 **1 / 1**

✓ **- 0 pts Correct (all of the above)**

- 0.75 pts Only selected 1

- 0.5 pts Only selected 2

- 0.25 pts Only selected 3

- 1 pts Blank/incorrect

4.8 3.8 **1 / 1**

✓ **- 0 pts Correct (-100)**

- 1 pts incorrect/blank

4.9 3.9 **1 / 1**

✓ **- 0 pts Correct (-10)**

- 1 pts blank/incorrect

QUESTION 5

4 Function Approximation 8 pts

5.1 4.1 **1 / 1**

✓ **- 0 pts Correct ($$2^{30720} = 2^{160 * 92}$$)**

- 1 pts Incorrect

5.2 4.2 **1 / 1**

✓ **- 0 pts Correct $$3(2^{30720})$$ or equivalent**

- 0.5 pts Correct given incorrect previous part (ie: you had 3*previous answer)

- 1 pts Blank/Incorrect

5.3 4.3 **1 / 1**

✓ **- 0 pts Correct (both infinite [uncountable])**

- 1 pts Incorrect / blank (needed to specify it was infinite)

- 0.5 pts Only one correct

5.4 4.4 **1 / 2**

- 0 pts Correctly mentions construction of a (huge) state vector s and how each state-action pair would have its own weight, encoding a table lookup (not feasible given how large it is)

- 0.5 pts Mostly correct with slightly insufficient explanation for construction of $$s$$ OR how it encodes a table lookup OR how it is a special case of Q-learning with a linear function approximator

- 1 pts Doesn't describe construction of $$s$$ correctly or
sufficiently

✓ **- 1 pts Describes construction of $$s$$ but doesn't sufficiently explain how it encodes a table lookup OR how it is a special case of Q-learning with a linear function approximator**

- 2 pts Incorrect

- 2 pts Blank

5.5 4.5 **2.5 / 3**

- 0 pts Correct, the samples are not independent and identically distributed because they are highly correlated. Experience replay will fix this.

- 1 pts Did not say that samples were not independent and identically distributed

- 1 pts Did not say why the observations are not iid, ie: because samples are highly correlated (because $$s'$$ depends on $$s$$ and $$a$$), or some similar explanation

- 1 pts Did not provide a relevant solution in context of RL, did not say experience replay or some similar

setup would solve this, or included an incorrect solution to their answer. Common incorrect answers include taking larger samples (you're still only visiting each state-action pair a finite number of times since every state isn't equally probable), using gradient or minibatch gradient descent, adding tiling, or using a higher value of $$\epsilon$$ (none of these decorrelate the update at $$s$$ and at $$s'$$).

- 0.5 pts Incorrect reason for highly correlated states (greedy actions imply heavily correlated states, state space representation implies heavily correlated states, etc)

✓ **- 0.5 pts Incorrect interpretation of experience replay (ie: mentioned that experience replay is a replacement for SGD)**

QUESTION 6

5 Empirical Questions 10 pts

6.1 5.1.1 1.5 / 1.5

✓ **- 0 pts Reasonable**

- 1 pts Incorrect returns

- 0.5 pts Incorrect rolling mean (common error: plotting the average for every 25 episodes)

- 1.5 pts Blank

6.2 5.1.2 1.5 / 1.5

✓ **- 0 pts Reasonable**

- 1 pts Incorrect returns

- 0.5 pts Incorrect rolling mean (common error: plotting the average for every 25 episodes)

- 0.01 pts Minor error

- 1.5 pts Blank

6.3 5.1.3 1 / 1

✓ **+ 1 pts Tile does better than raw**

+ 0 pts Incorrect (very little insight about tiling, such as "rolling mean smooths the plot", "converges faster (does not mention to a better value, what if it just converges to something bad very fast?)", "the variance is larger for tile", etc, or simply explaining the plots without any insight)

+ 0 pts Blank

6.4 5.2 2 / 2

✓ **- 0 pts Correct**

- 0.5 pts Incorrect feature assignment

- 0.5 pts Incorrect feature assignment explanation (independent of item 2)

- 0.25 pts Vague feature assignment justification (e.g. mentions complexity vs. simplicity or describes raw vs. tile, but not how it relates to linearity of raw)

- 1 pts Incorrect or missing interpretation of plots

- 0.5 pts Partial or incomplete interpretation of plots (e.g. does not discuss how value function increases with position)

- 0.5 pts Minor error

- 2 pts Blank

6.5 5.3 2 / 2

✓ **- 0 pts Correct**

- 1 pts Incorrect answer

- 1 pts Incorrect example or invalid justification (independent of item 2)

- 0.5 pts Vague example or justification that hints at understanding of a correct response (e.g. notes that maxing may introduce nonlinearities, but does not describe a potential shape)

- 0.5 pts Minor error

- 2 pts Blank

6.6 5.4 2 / 2

✓ **- 0 pts Correct [(a) is tiles, (b) is raw (with explanation), gives reasonable explanation of policy]**

- 2 pts Blank/completely incorrect

- 0.5 pts Did not explain why they learn the patches at these specific locations or gave incorrect/insufficient explanation

- 0.5 pts Incorrect or insufficient explanation of why (a) is tiled and (b) is raw features

QUESTION 7

7 6 Collaboration Questions 0 / 0

✓ **- 0 pts** **Correct**

HOMEWORK 8: REINFORCEMENT LEARNING

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (SPRING 2022)
http://www.cs.cmu.edu/~mgormley/courses/10601/

OUT: April 12, 2022
DUE: April 21, 2022
TAs: Sana, Chu, Hayden, Tori, Prasoon

Summary In this assignment, you will implement a reinforcement learning algorithm for solving the classic mountain-car environment. As a warmup, the first section will lead you through an on-paper example of how value iteration and Q-learning work. Then, in Section 7, you will implement Q-learning with function approximation to solve the mountain car environment.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. If your scanned submission misaligns the template, there will be a 5% penalty. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment Python 3.9.6) and versions of permitted libraries (`numpy` 1.21.2 and `scipy` 1.7.1) match those used on Gradescope. You have 10 free Gradescope programming submissions. After 10 submissions, you will begin to lose points from your total programming score. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Instructions for Specific Problem Types

For "Select One" questions, please fill in the appropriate bubble completely:

Select One: Who taught this course?

- ● Matt Gormley
- ○ Marie Curie
- ○ Noam Chomsky

If you need to change your answer, you may cross out the previous answer and bubble in the new answer:

Select One: Who taught this course?

- ● Matt Gormley
- ○ Marie Curie
- ⊗ Noam Chomsky

For "Select all that apply" questions, please fill in all appropriate squares completely:

Select all that apply: Which are scientists?

- ■ Stephen Hawking
- ■ Albert Einstein
- ■ Isaac Newton
- □ I don't know

Again, if you need to change your answer, you may cross out the previous answer(s) and bubble in the new answer(s):

Select all that apply: Which are scientists?

- ■ Stephen Hawking
- ■ Albert Einstein
- ■ Isaac Newton
- ⊠ I don't know

For questions where you must fill in a blank, please make sure your final answer is fully included in the given space. You may cross out answers or parts of answers, but the final answer must still be within the given space.

Fill in the blank: What is the course number?

10-601	10-~~7~~601

Written Questions (46 points)

1 LaTeX Bonus Point (1 points)

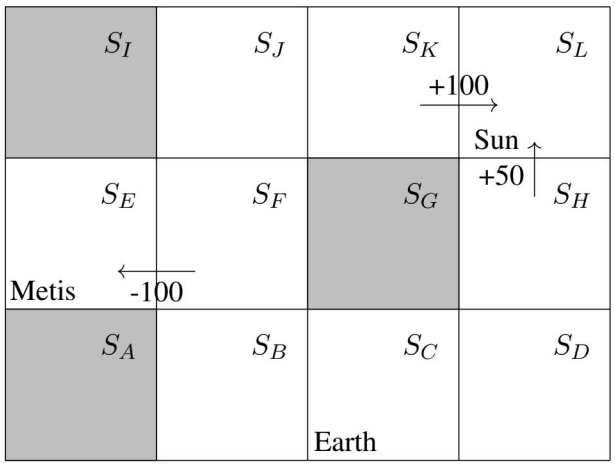
1. (1 point) **Select one:** Did you use LaTeX for the entire written portion of this homework?

   ● Yes

   ○ No

2 Value Iteration (18 points)

While attending an ML conference, you meet scientists at NASA who ask for your help sending spacecraft to the surface of the Sun. Specifically, they ask you to develop a reinforcement learning agent capable of carrying out the space-flight from Earth to the Sun. You model this problem as a Markov decision process (MDP). The figure below depicts the state space.



Here are the details:

1. Each grid cell is a state $S_A, S_B, ..., S_L$ corresponding to a position in the solar system.
2. The action space includes movement `up`/`down`/`left`/`right`. Transitions are **non-deterministic**. With probability 80% the agent transitions to the intended state. With probability 10% the agent slips left *of the intended direction*. With probability 10% the agent slips right *of the intended direction*. For example, if the agent is in state S_F and takes action `left`, it moves to state S_E with 80% probability, it moves to state S_B (left of the intended direction) with 10% probability, and it moves to state S_J (right of the intended direction) with 10% probability,.
3. It is not possible to move into blocked states, which are shaded grey, since they contain other planets. If the agent's action would move them out of bounds of the board or a blocked state, it remains in the same state.
4. The start state is S_C (Earth). The terminal states include both the S_L (Sun) and S_E (asteroid Metis).
5. Non-zero rewards are depicted with arrows. Flying into the Sun from the left gives positive reward $R(S_K, \texttt{a}, S_L) = +100 \ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$, since it is more fuel-efficient than flying into the sun from below (the agent can use the gravitational field of the planets in S_A and S_I in addition to S_G). However, approaching the Sun from the left has its risks, as flying into Metis is inadvisable and gives negative reward $R(S_F, \texttt{a}, S_E) = -100 \ \forall \texttt{a} \in$

$\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. Note that flying into the Sun from below still achieves the goal and gives positive reward $R(S_H, \texttt{a}, S_L) = +50 \; \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. All other rewards are zero.

Below, let $V^*(s)$ denote the value function for state s using the optimal policy $\pi^*(s)$.

2.1 Synchronous Value Iteration

1. (3 points) Report the value of each state after a single round of **synchronous** value iteration in the table below. Initialize the value table $V^0(s) = 0, \forall s \in \{S_A \dots S_L\}$ and assume $\gamma = 0.9$. Visit each state in *reverse alphabetical order*. Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round intermediate values when calculating your answers.**

	S_J 0	S_K 80	S_L 0
S_E 0	S_F 0		S_H 40
	S_B 0	S_C 0	S_D 0

2. (3 points) What is the policy, $\pi(s)$, that corresponds to the value table you calculated above? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

	S_J `right`	S_K `right`	S_L `terminal`
S_E `terminal`	S_F `right`		S_H `up`
	S_B `down`	S_C `down`	S_D `up`

2.2 Asynchronous Value Iteration

1. (3 points) Starting over, report the value of each state for a single round of **asynchronous** value iteration in the table below. Initialize the value table $V^0(s) = 0, \forall s \in \{S_A \dots S_L\}$ and assume $\gamma = 0.9$. Visit each state in *reverse alphabetical order*. Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round intermediate values when calculating your answers.**

	S_J 57.6	S_K 80	S_L 0
S_E 0	S_F 31.5		S_H 40
	S_B 24.5	S_C 20.7	S_D 28.8

2. (3 points) What is the policy, $\pi(s)$, that corresponds to the value table you calculated above? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

	S_J `right`	S_K `right`	S_L `terminal`
S_E `terminal`	S_F `up`		S_H `up`
	S_B `up`	S_C `right`	S_D `up`

3. (3 points) Below, we give you the value of each state one round before the convergence of **asynchronous** value iteration.[1] What is the final value of each state, $V^*(s)$? Be sure to use **asynchronous** value iteration, and visit each state in *reverse alphabetical order*. Ignore the blocked states. Round your *answers only* to the first decimal place. **Do not round intermediate values when calculating your answers.**

S_I	S_J 80.9516	S_K 96.9920	S_L
S_E	S_F 52.5638	S_G	S_H 48.4960
S_A	S_B 44.5586	S_C 35.3208	S_D 41.2039

Your solution:

S_I	S_J 82.2	S_K 97.5	S_L
S_E	S_F 53.9	S_G	S_H 48.7
S_A	S_B 46.3	S_C 38.4	S_D 42.0

[1] This is actually one round before the *policy* convergence, not the *value* convergence. The values we provide are the values after the third iteration.

4. (3 points) What is the policy, $\pi^*(s)$, that corresponds to $V^*(s)$? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

	S_J `right`	S_K `right`	S_L
S_E	S_F `up`		S_H `up`
	S_B `up`	S_C `left`	S_D `up`

3 Q-Learning (9 points)

Let's consider the same grid world as before:

S_I	S_J	S_K	S_L Sun
S_E Metis	S_F	S_G	S_H
S_A	S_B	S_C Earth	S_D

This time, however, suppose we **don't know** the reward function or the transition probability between states. Some rules for this setup are:

1. Each grid cell is a state $S_A, S_B, \ldots, S_L$ corresponding to a position in the solar system.
2. The action space of the agent is: $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$.
3. If the agent hits the edge of the board, it remains in the same state. It is not possible to move into blocked states, which are shaded grey, since they contain other planets.
4. The start state is S_C (Earth). The terminal states include both the S_L (Sun) and S_E (asteroid Metis).
5. Use the discount factor $\gamma = 0.9$, $\epsilon = 0.5$, and learning rate $\alpha = 0.1$.

We will go through three iterations of Q-learning in this section. Initialize $Q(s, a) = 0, \forall s \in \{S_A, \ldots, S_L\}$, $\forall a \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$.

1. (1 point) **Select all that apply:** If the agent were to act greedily, what action would it take at this time?

   - ■ `up`
   - ■ `down`
   - ■ `left`
   - ■ `right`

2. (1 point) Beginning at state S_C, you take the action `right` and receive a reward of 0. You are now in state S_D. What is the new value for $Q(S_C, \texttt{right})$, assuming the update for deterministic transitions? Round your answer to the fourth decimal place.

$Q(S_C, \texttt{right})$
0

3. (1 point) What is the new value for $Q(S_C, \texttt{right})$, using the temporal difference error update? Round your answer to the fourth decimal place.

$Q(S_C, \texttt{right})$
0

4. (1 point) **Select all that apply:** Continue to update your Q-function (as calculated by the temporal difference error update) from above. This time, though, assume your run has brought you to state S_H with no updates to the Q-function in the process. If the agent were to act greedily, what action would it take at this time?

   - ■ `up`
   - ■ `down`
   - ■ `left`
   - ■ `right`

5. (1 point) Beginning at state S_H, you take the action `up`, receive a reward of +50, and the run terminates. What is the new value for $Q(S_H, \texttt{up})$, assuming the update for deterministic transitions? Round your answer to the fourth decimal place.

$Q(S_H, \texttt{up})$
50

6. (1 point) What is the new value for $Q(S_H, \texttt{up})$, using the temporal difference error update? Round your answer to the fourth decimal place.

$Q(S_H, \texttt{up})$
5

7. (1 point) **Select all that apply:** Continue to update your Q-function (as calculated by the temporal difference error update) from above. You start from state S_C since the previous run terminated, but manage to make it to state S_F with no updates to the Q-function. If the agent were to act greedily, what action would it take at this time?

   - ■ `up`
   - ■ `down`
   - ■ `left`
   - ■ `right`

8. (1 point) Beginning at state S_F, you take the action `left`, receive a reward of -100, and the run terminates. What is the new value for $Q(S_F, \texttt{left})$, assuming the update for deterministic transitions? Round your answer to the fourth decimal place.

$Q(S_F, \texttt{left})$
-100

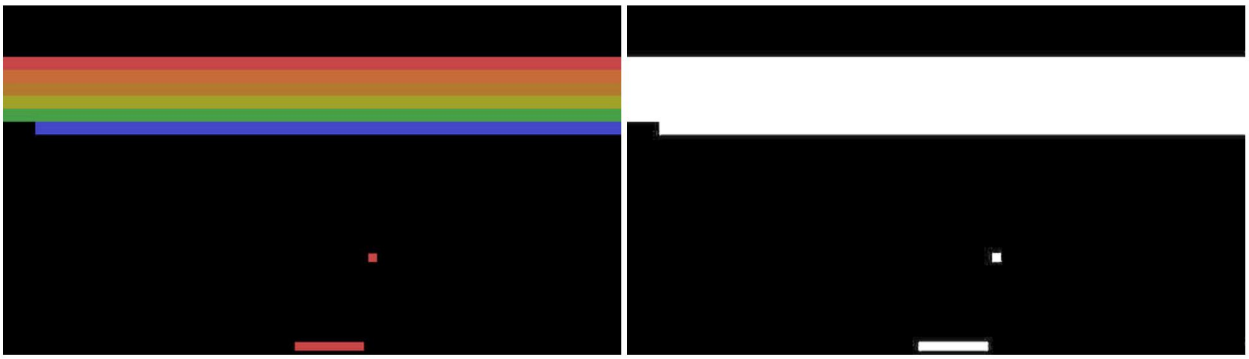
9. (1 point) What is the new value for $Q(S_F, \texttt{left})$, using the temporal difference error update? Round your answer to the fourth decimal place.

$Q(S_F, \texttt{left})$
-10

4 Function Approximation (8 points)

In this question we will motivate function approximation for solving Markov Decision Processes by looking at Breakout, a game on the Atari 2600. The Atari 2600 is a gaming system released in the 1980s, but nevertheless is a popular target for reinforcement learning papers and benchmarks. The Atari 2600 has a resolution of 160×192 pixels. In the case of Breakout, we try to move the paddle to hit the ball in order to break as many tiles above as possible. We have the following actions:

- Move the paddle left
- Move the paddle right
- Do nothing



(a) Atari Breakout (b) Black and white Breakout

Figure 1: Atari Breakout. 1a is what Breakout looks like. We have the paddle in the bottom of the screen aiming to hit the ball in order to break the tiles at the top of the screen. 1b is our transformation of Atari Breakout into black and white pixels for the purpose of some of the following problems.

1. (1 point) Suppose we are dealing with the black and white version of Breakout[2] as in Figure 1b. Furthermore, suppose we are representing the state of the game as just a vector of pixel values without considering if a certain pixel is always black or white. Since we are dealing with the black and white version of the game, these pixel values can either be 0 or 1.

   What is the size of the state space?

   Answer

   2^{30720}

2. (1 point) In the same setting as the previous part, suppose we wish to apply Q-learning to this problem. What is the size of the Q-value table we will need?

   Answer

   $(2^{30720}) * 3$

[2]Play a "Google"-Doodle version here

3. (1 point) Now assume we are dealing with the colored version of Breakout as in Figure 1a. Now each pixel is a tuple of real valued numbers between 0 and 1. For example, black is represented as $(0, 0, 0)$ and white is $(1, 1, 1)$.

   What is the size of the state space and Q-value table we will need?

   Answer

   Inf

   By now you should see that we will need a huge table in order to apply Q-learning (and similarly value iteration and policy iteration) to Breakout given this state representation. This table would not even fit in the memory of any reasonable computer! Now this choice of state representation is particularly naïve. If we choose a better state representation, we could drastically reduce the table size needed.

   On the other hand, perhaps we don't want to spend our days feature engineering a state representation for Breakout. Instead we can apply function approximation to our reinforcement algorithms! The whole idea of function approximation is that states nearby to the state of interest should have *similar* values. That is, we should be able to generalize the value of a state to nearby and unseen states.

   Let us define $q_\pi(s, a)$ as the true action value function of the current policy π. Assume $q_\pi(s, a)$ is given to us by some oracle. Also define $q(s, a; \mathbf{w})$ as the action value predicted by the function approximator parameterized by $\mathbf{w}$. Here $\mathbf{w}$ is a matrix of size $\dim(S) \times |\mathcal{A}|$, where $\dim(S)$ denotes the dimension of the state space. Clearly we want to have $q(s, a; \mathbf{w})$ be close to $q_\pi(s, a)$ for all (s, a) pairs we see. This is just our standard regression setting. That is, our objective function is just the Mean Squared Error:

$$J(\mathbf{w}) = \frac{1}{2}\frac{1}{N}\sum_{s\in\mathcal{S}, a\in\mathcal{A}} \left(q_\pi(s, a) - q(s, a; \mathbf{w})\right)^2. \tag{1}$$

   Because we want to update for each example stochastically[3], we get the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left(q(s, a; \mathbf{w}) - q_\pi(s, a)\right) \nabla_{\mathbf{w}} q(s, a; \mathbf{w}). \tag{2}$$

   However, more often than not we will not have access to the oracle that gives us our target $q_\pi(s, a)$. So how do we get the target to regress $q(s, a; \mathbf{w})$ on? One way is to bootstrap an estimate of the action value under a greedy policy using the function approximator itself. That is to say

$$q_\pi(s, a) \approx r + \gamma \max_{a'} q(s', a'; \mathbf{w}) \tag{3}$$

   where r is the reward observed from taking action a at state s, γ is the discount factor and s' is the state resulting from taking action a at state s. This target is often called the Temporal Difference (TD) target, and gives rise to the following update for the parameters of our function approximator in lieu of a tabular update:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \underbrace{\Bigg(q(s, a; \mathbf{w}) - \underbrace{\left(r + \gamma \max_{a'} q(s', a'; \mathbf{w})\right)}_{\text{TD Target}} \Bigg)}_{\text{TD Error}} \nabla_{\mathbf{w}} q(s, a; \mathbf{w}). \tag{4}$$

[3] This is not really stochastic, you will be asked in a bit why.

4. (2 points) Consider the setting where we can represent our state by some vector $\mathbf{s}$, action $a \in \{0, 1, 2\}$ and we choose a linear approximator. That is:

$$q(\mathbf{s}, a; \mathbf{w}) = \mathbf{s}^T \mathbf{w}_a \tag{5}$$

Again, assume we are in the black and white setting of Breakout as in Figure 1b. Show that tabular Q-learning is just a special case of Q-learning with a linear function approximator by describing a construction of $\mathbf{s}$. (**Hint**: Engineer features such that Eq. (5) encodes a table lookup)

Answer

This approach uses one-hot encoding. For example, (s1,left) is 1 whereas everything else is zero. Thus, when $\mathbf{s}^T \mathbf{w}_a, it would return Q(s1, left)$.
[(s1,left),(s1,right),(s1,do nothing)....sn,do nothing]^T * w would return Real values.

5. (3 points) Stochastic Gradient Descent works because we can assume that the samples we receive are independent and identically distributed. Is that the case here? If not, why and what are some ways you think you could combat this issue?

Answer

No it isn't. The samples we receive are not independent, because some pixels require pixels below them to be wiped out before they themselves can be wiped out. A way to combat this issue would be to use experience replay instead of using SGD, which relies on i.i.d. assumption.

5 Empirical Questions (10 points)

The following parts should be completed after you work through the programming portion of this assignment (Section 7).

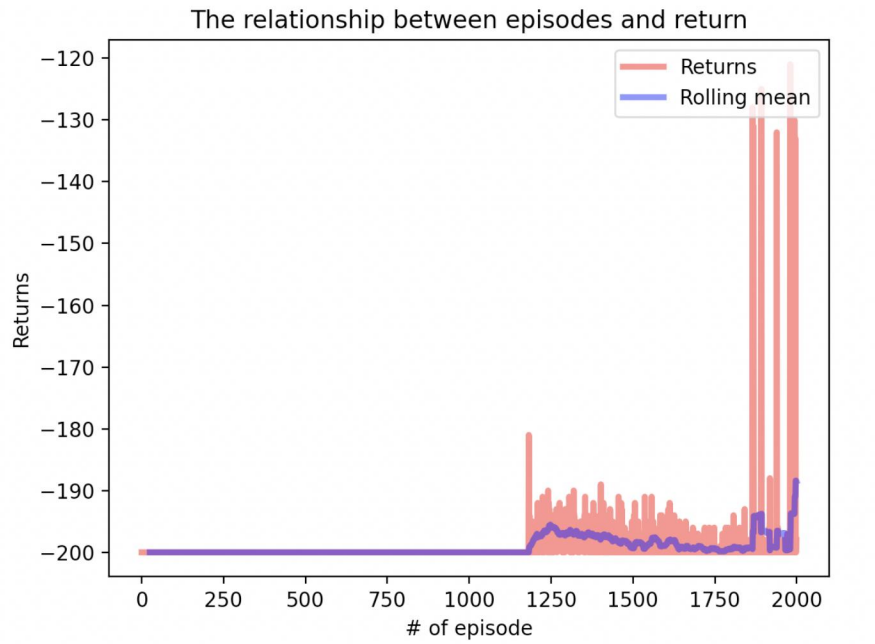
1. (4 points) Run Q-learning on the mountain car environment using both tile and raw features.

   For the raw features: run for 2000 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.999, and a learning rate of 0.001.

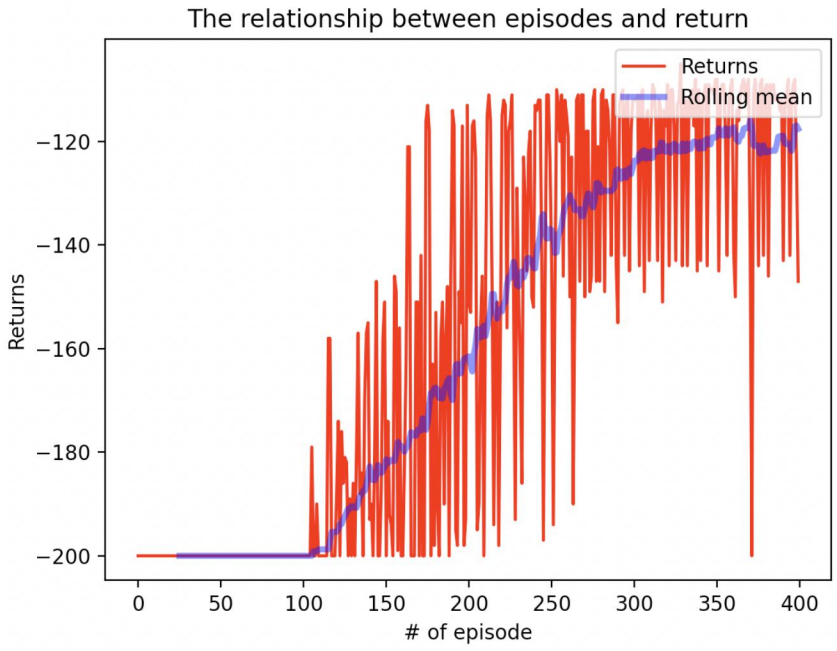
   For the tile features: run for 400 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.99, and a learning rate of 0.00005.

   For each set of features, plot the return (sum of all rewards in an episode) per episode on a line graph. On the same graph, also plot the rolling mean over a 25 episode window. Comment on the difference between the plots.

Plot of Raw

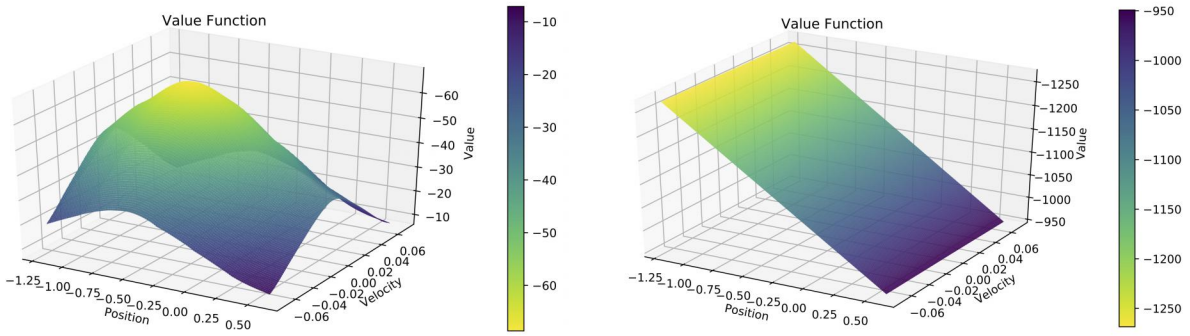


Plot of Tile



Comment

With raws, the returns stay around -200 for the first 1000 episodes and improve towards -120 in the second half. The rolling mean also improves in the 2nd half from -200 to -190ish.
On the other hand, with tile, both returns and rolling mean fluctuate heavily. While returns stay around -200 in the first 100 episodes, returns improve to -120 while fluctuating heavily as of episode increases. Similarly, the rolling mean improves from -200 to -120 while fluctuating as of episode increases.



(a) (b)

Figure 2: Estimated optimal value function visualizations for both types of features

2. (2 points) For both raw and tile features, we have run Q-learning with some good parameters and created visualizations of the value functions after many episodes. For each plot in Figure 2, write down which features (raw or tile) were likely used in Q-learning with function approximation. Explain your reasoning. In addition, interpret each of these plots in the context of the mountain car environment.

Answer

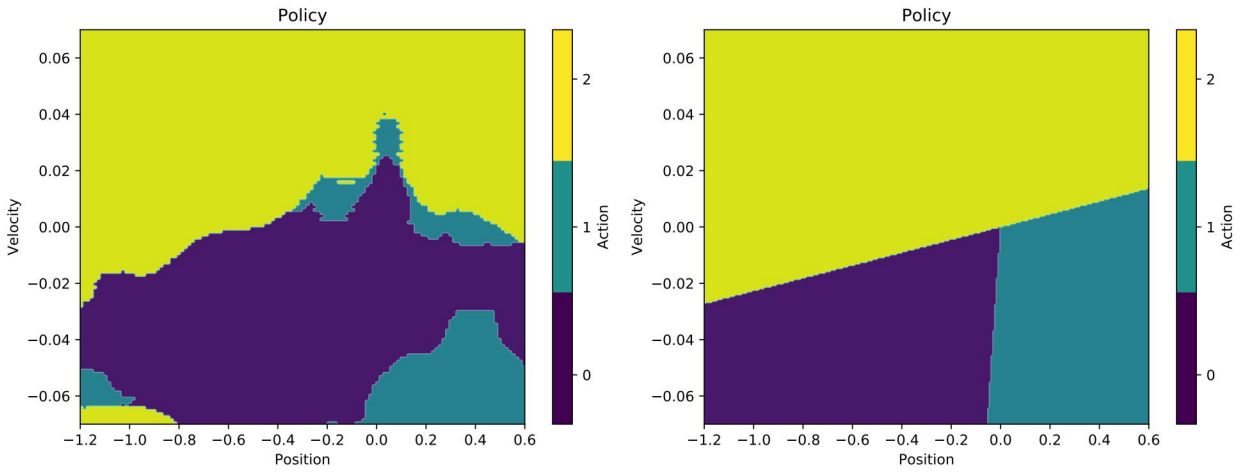
Most likely, raw was used for (b) and tile was used for (a). We know this because raw can't learn non-linear relationship, whereas tile can.
(a) is more representative because it learns the con of being at bottom and pros of being on left hill. This is shown in how color gets darker for more negative value of position. However, (b) does not learn this and this is shown in how it has light green(high negative reward) for more negative values of position.

3. (2 points) We see that Figure 2b seems to look like a plane. Can the value function depicted in this plot ever be nonlinear (linear here *strictly* refers to a function that can be expressed in the form of $y = \mathbf{A}\mathbf{x} + \mathbf{b}$)? If so, describe a potential shape. If not, explain why.

Hint: How do we calculate the value of a state given the Q-values?

Answer

The value function depicted in the plot can be nonlinear. For example, when there are two planes (one with negative slope and one with positive slope), value function depicted in the plot can have a piecewise shape.



(a) (b)

Figure 3: Estimated optimal policy visualizations for both types of features

4. (2 points) In a similar fashion to the previous question, we have created visualizations of the potential policies learned. For each plot in Figure 3, write down which features (raw or tile) were likely used in Q-learning with function approximation. Explain your reasoning. In addition, interpret each of these plots in the context of the mountain car environment. Specifically, why are the edges linear v.s. non-linear? Why do they learn these patches at these specific locations?

Answer

we know (b) for raw and (a) for tile because raw can't learn non-linear relationship, whereas tile can. Hence, edges are linear for (b) and non-linear for (a).
For raw, the car learns that it's best to not do anything when its (position,velocity) is bounded in (-0.09,-0.07),(0,-0.01),(0.6,0.01). Also, the car learns to move left when its (position,velocity) is bounded in (-0.09,-0.07),(0,-0.01),(-1.2,-0.03). Also, the car learns to move right when its (position,velocity) is not in the two regions previously described.
On the other hand, for tile, the boundaries are much less clear. There is huge mix when the car learns to stay or move left. The region for deciding to move right is also slightly mixed in.
They learn these patches at these specific locations because (a) is a tile. It can learn non-linear relationship and be more representative because it learns the con of being at bottom and pros of being on left hill, whereas (b)—the raw— doesn't and can only learn linear relationship

6 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found here.

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details.
3. Did you find or come across code that implements any part of this assignment ? If so, include full details.

Your Answer

No,no,no

7 Programming [68 Points]

Your goal in this assignment is to implement Q-learning with linear function approximation to solve the mountain car environment. You will implement all of the functions needed to initialize, train, evaluate, and obtain the optimal policies and action values with Q-learning. In this assignment we will provide the environment for you. The program you write will be automatically graded using the Gradescope system.

7.1 Specification of Mountain Car

In this assignment, you will be given code that fully defines the Mountain Car environment. In Mountain Car you control a car that starts at the bottom of a valley. Your goal is to reach the flag at the top right, as seen in Figure 4. However, your car is under-powered and cannot climb up the hill by itself. Instead you must learn to leverage gravity and momentum to make your way to the flag. It would also be good to get to this flag as fast as possible.

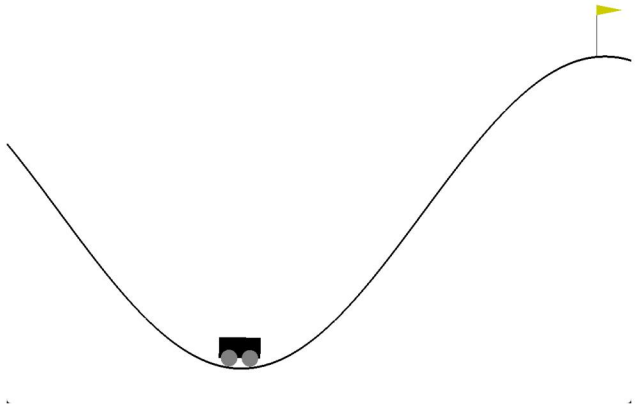


Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

The state of the environment is represented by two variables, `position` and `velocity`. `position` can be between $[-1.2, 0.6]$ (inclusive) and `velocity` can be between $[-0.07, 0.07]$ (inclusive). These are just measurements along the x-axis.

The actions that you may take at any state are $\{0, 1, 2\}$, where each number corresponds to an action: (0) pushing the car left, (1) doing nothing, and (2) pushing the car right.

7.2 Q-learning with Linear Approximations

The Q-learning algorithm is a model-free reinforcement learning algorithm, where we assume we don't have access to the model of the environment the agent is interacting with. We also don't build a complete model of the environment during the learning process. A learning agent interacts with the environment solely based on calls to **step** and **reset** methods of the environment. Then the Q-learning algorithm updates the q-values based on the values returned by these methods. Analogously, in the approximation setting the algorithm will instead update the parameters of q-value approximator.

Let the learning rate be α and discount factor be γ. Recall that we have the information after one interaction with the environment, (s, a, r, s'). The tabular update rule based on this information is:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a')\right).$$

Instead, for the function approximation setting we use the following update rule derived from the Function Approximation Section (Section 4). Note that we have made the bias term explicit here, where before it was implicitly folded into $\mathbf{w}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left(q(\mathbf{s}, a; \mathbf{w}) - (r + \gamma \max_{a'} q(\mathbf{s}', a'; \mathbf{w}) \right) \nabla_{\mathbf{w}} q(\mathbf{s}, a; \mathbf{w}),$$

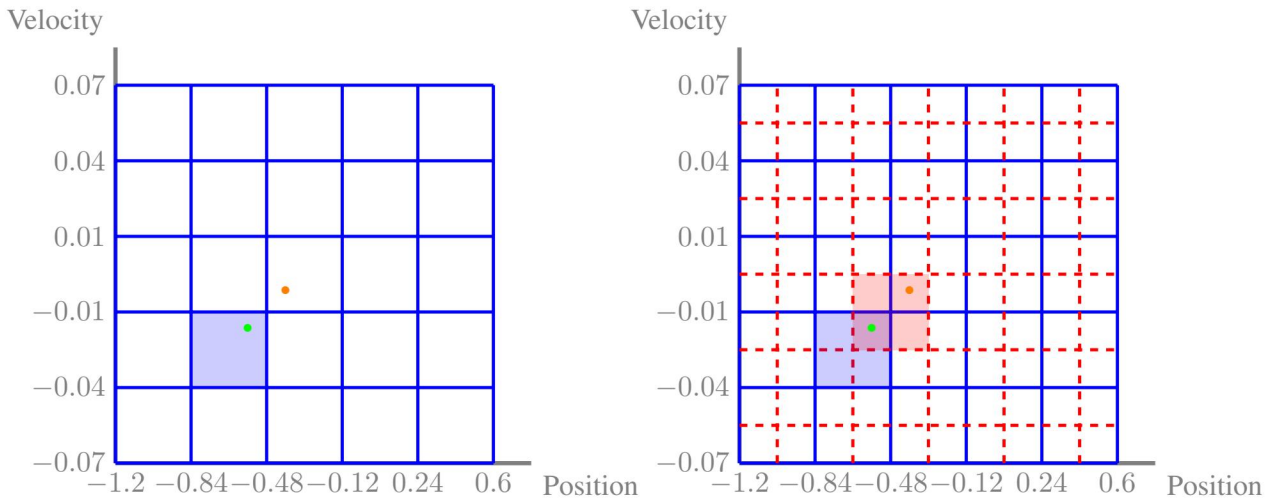
where

$$q(\mathbf{s}, a; \mathbf{w}) = \mathbf{s}^T \mathbf{w}_a + b.$$

The epsilon-greedy action selection method selects the optimal action with probability $1 - \epsilon$ and selects uniformly at random from one of the 3 actions (0, 1, 2) with probability ϵ. The reason that we use an epsilon-greedy action selection is we would like the agent to do explorations by stochastically selecting random actions with small probability. For the purpose of testing, we will test two cases: $\epsilon = 0$ and $0 < \epsilon < 1$. When $\epsilon = 0$ (no exploration), the program becomes deterministic and your output have to match our reference output accurately. In this case, **pick the action represented by the smallest number if there is a draw in the greedy action selection process**. For example, if we are at state s and $Q(s, 0) = Q(s, 2)$, then take action 0. When $0 < \epsilon < 1$, your output will need to fall in a certain range within the reference determined by running exhaustive experiments on the input parameters.

7.3 Feature Engineering

Linear approximations are great in their ease of use and implementations. However, there sometimes is a downside; they're *linear*. This can pose a problem when we think the value function itself is nonlinear with respect to the state. For example, we may want the value function to be symmetric about 0 velocity. To combat this issue we could throw a more complex approximator at this problem, like a neural network. But we want to maintain simplicity in this assignment, so instead we will look at a nonlinear transformation of the "raw" state.



(a) A discretization of the state space of Mountain Car

(b) A tiling of the state space of Mountain Car

Figure 5: State representations for the states of Mountain Car

For the Mountain Car environment, we know that `position` and `velocity` are both bounded. What we can do is draw a grid over the possible `position-velocity` combinations as seen in Figure 5a. We then enumerate the grid from bottom left to top right, row by row. Then we map all states that fall into a grid

square with the corresponding one-hot encoding of the grid number. For efficiency reasons we will just use the index that is non-zero. For example the green point would be mapped to $\{6\}$ and the orange point to $\{12\}$. This is called a *discretization* of the state space.

The downside to the above approach is that although observing the green point will let us learn parameters that generalize to other points in the shaded blue region, we will not be able to generalize to the orange point even though it is nearby. We can instead draw two grids over the state space, each offset slightly from each other as in Figure 5b. Now we can map the green point to two indices, one for each grid, and get $\{6, 39\}$ (note the index for orange grid starts from the end of blue index, i.e. 25). Now the green point has parameters that generalize to points that map to $\{6\}$ (the blue shaded region) in the first discretization and parameters that generalize to points that map to $\{39\}$ (the red shaded region) in the second. We can generalize this to multiple grids, which is what we do in practice. This is called a *tiling* or a *coarse-coding* of the state space.

7.4 Implementation Details

Here we describe the API to interact with the Mountain Car environment available to you.

- `__init__(mode, debug)`: Initializes the environment to the a mode specified by the value of `mode`. This can be a string of either "raw" or "tile".

  "raw" mode tells the environment to give you the state representation of raw features encoded in a sparse format: $\{0 \to \texttt{position}, 1 \to \texttt{velocity}\}$.

  In "tile" mode you are given indices of the tiles which are active in a sparse format: $\{T_1 \to 1, T_2 \to 1, \ldots T_n \to 1\}$ where T_i is the tile index for the ith tiling. All other tile indices are assumed to map to 0. For example the state representation of the example in Figure 5b would become $\{6 \to 1, 39 \to 1\}$.

  The dimension of the state space of the "raw" mode is 2. The dimension of the state space of the "tile" mode is 2048. These values can be accessed from the environment through the `state_space` property, and similarly for other languages.

  `debug` is an optional argument for debugging. See Section 7.5 for more details.

- `reset()`: Reset the environment to starting conditions.

- `step(action)`: Take a step in the environment with the given action. `action` must be either 0, 1 or 2. This will return a tuple of (`state`, `reward`, `done`) which is the next state, the reward observed, and a boolean indicating if you reached the goal or not, ending the episode. The `state` will be either a raw' or tile representation, as defined above, depending on how you initialized Mountain Car. If you observe `done = True` then you should `reset` the environment and end the episode. Failure to do so will result in undefined behavior.

- `render()`: Visualize the environment (not graded). Requires the installation of `pyglet`[4]. We highly recommend you to use this only after you implement everything. Do *not* use this as a tool for debugging—this should rather be used as a tool for understanding Q-learning better. It is computationally intensive to render graphics, so only call the function once every 100 or 1000 episodes. This will be a no-op in Gradescope.

You should now implement your Q-learning algorithm with linear approximations in `q_learning.py`. The program will assume access to a given environment file(s) which contains the Mountain Car environment which we have given you. **Initialize the parameters of the linear model with all 0 (and don't forget**

[4] You can install it by typing `pip install pyglet` in your shell.

to include a bias!) and use the epsilon-greedy strategy for action selection.

Your program should write a output file containing the total rewards (the returns) for every episode after running Q-learning algorithm. There should be one return per line.

Your program should also write an output file containing the weights of the linear model. The first line should be the value of the bias. Then the following $|\mathcal{S}| \times |\mathcal{A}|$ lines should be the values of weights, outputted in row major order[5], assuming your weights are stored in a $|\mathcal{S}| \times |\mathcal{A}|$ matrix.

The autograder will use the following commands to call your function:

```
$ python q_learning.py [args...]
```

where above `[args...]` is a placeholder for command-line arguments: `<env>` `<mode>` `<weight_out>` `<returns_out>` `<episodes>` `<max_iterations>` `<epsilon>` `<gamma>` `<learning_rate>`. These arguments are described in detail below:

1. `<env>`: the environment that you are running, either `mc` for Mountain Car or `gw` for Grid World.
2. `<mode>`: mode to run the environment in. Should be either `raw` or `tile`. Note that Grid World operates only in `tile` mode.
3. `<weight_out>`: path to output the weights of the linear model.
4. `<returns_out>`: path to output the returns of the agent.
5. `<episodes>`: the number of episodes your program should train the agent for. One episode is a sequence of states, actions and rewards, which ends with terminal state or ends when the maximum episode length has been reached.
6. `<max_iterations>`: the maximum of the length of an episode. When this is reached, we terminate the current episode.
7. `<epsilon>`: the value ϵ for the epsilon-greedy strategy.
8. `<gamma>`: the discount factor γ.
9. `<learning_rate>`: the learning rate α of the Q-learning algorithm.

Example command:

```
$ python q_learning.py mc raw mc_raw_weight.out mc_raw_returns.out \
 4 200 0.05 0.99 0.01
```

Example output from the above command (may not be exactly the same, but should be close up to 0.01):

`<weight_out>`

```
-7.66116708660012
1.3411763263964611
1.3419332653944924
1.3370748857368524
-0.0013201697867872468
0.0010668243394517697
0.0012565450062079566
```

[5] https://en.wikipedia.org/wiki/Row-_and_column-major_order

<returns_out>

```
-200.0
-200.0
-200.0
-200.0
```

7.5 Debugging Tips

To help with debugging, we have provided the option for printing each step of the Q-learning train function based on the reference output for the Grid World environment. We created this output by adding the `debug=True` argument when initializing the Grid World environment. You may do the same to compare your output against ours.

We recommend first checking your outputs based on a run with extremely simple parameters. Remember to set <epsilon>=0 so the program is run without the epsilon-greedy strategy.

We have provided output on the Grid World for the following simple command:

```
$ python q_learning.py gw tile gw_simple_weight.out \
  gw_simple_returns.out 1 1 0.0 1 1
```

Once this works, you can change the parameters to be slightly more complex (such as the ones we have below), and check with our calculations again:

```
$ python q_learning.py gw tile gw_weight.out gw_returns.out \
  3 5 0.0 0.9 0.01
```

The logs for both of the above commands should be in `reference_output/gw_simple.log` and `reference_output/gw.log`, respectively.

In addition, we have provided `mc_weight.out` and `mc_returns.out` in the handout, which are generated using the following parameters:

- <env>: mc
- <mode>: tile
- <episodes>: 25
- <max_iterations>: 200
- <epsilon>: 0.0
- <gamma>: 0.99
- <learning_rate>: 0.005

Example command:

```
$ python q_learning.py mc tile mc_tile_weight.out \
 mc_tile_returns.out 25 200 0.0 0.99 0.005
```

7.6 Gradescope Submission

You should submit your `q_learning.py` to Gradescope. **Any other files uploaded will be discarded or reverted back to the original version provided in the handout.** Do *not* use other file names.