HOMEWORK 8
REINFORCEMENT LEARNING *

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (FALL 2021)
http://mlcourse.org

OUT: Nov. 12, 2021
DUE: Nov. 21, 2021
TAs: Gopi Krishna, Roshan, Justin, Youngjoo, Jingyun

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 2, 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 (e.g. Python 3.9.6, OpenJDK 11.0.11, g++ 7.5.0) and versions of permitted libraries (e.g. 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

---

*Compiled on Wednesday 24th November, 2021 at 14:29

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.

Linear Algebra Libraries When implementing machine learning algorithms, it is often convenient to have a linear algebra library at your disposal. In this assignment, Java users may use EJML[a] or ND4J[b] and C++ users may use Eigen[c]. Details below. (As usual, Python users have NumPy.)

EJML for Java EJML is a pure Java linear algebra package with three interfaces. We strongly recommend using the SimpleMatrix interface. The autograder will use EJML version 0.41. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.41-libs/*:linalg_lib/nd4j-v1.0.0-M1.1-libs/*:./"` to ensure that all the EJML jars are on the classpath as well as your code.

ND4J for Java ND4J is a library for multidimensional tensors with an interface akin to Python's NumPy. The autograder will use ND4J version 1.0.0-M1.1. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.41-libs/*:linalg_lib/nd4j-v1.0.0-M1.1-libs/*:./"` to ensure that all the ND4J jars are on the classpath as well as your code.

Eigen for C++ Eigen is a header-only library, so there is no linking to worry about—just `#include` whatever components you need. The autograder will use Eigen version 3.4.0. The command line arguments above demonstrate how we will call you code. When compiling your code we will include, the argument `-I./linalg_lib` in order to include the `linalg_lib/Eigen` subdirectory, which contains all the headers.

We have included the correct versions of EJML/ND4J/Eigen in the `linalg_lib.zip` posted on the Coursework page of the course website for your convenience. It contains the same `linalg_lib/` directory that we will include in the current working directory when running your tests. Do **not** include EJML, ND4J, or Eigen in your homework submission; the autograder will ensure that they are in place.

---

[a] https://ejml.org
[b] https://javadoc.io/doc/org.nd4j/nd4j-api/latest/index.html
[c] http://eigen.tuxfamily.org/

Instructions for Specific Problem Types

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

Select One: Who taught this course?

      ● Matt Gormley / Henry Chai

      ○ 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 / Henry Chai

      ○ 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

      ☐ None of the above

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

1   Written Questions (34 points)

1.1   Non-Deterministic Value Iteration

1. In this question we will explore value iteration with a non-deterministic transition function. Suppose the agent is in a grid world. The action space of the agent is: Up, Down, Left and Right. In any state, there is the following **non-deterministic** transition function: 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. If the agent hits the edge of the board, it remains in the same state.

    For example, If the agent were in state B and choose action down, there is an 80% chance of moving to state E, a 10% chance of moving to state C, and a 10% chance of moving to state A.

    If at any point, the agent transitions into a state labeled P, it is given a reward of -100 and terminates. Similarly, if it transitions to a state G, it is given a reward of +1 and terminates. The agent is never initialized in state P or G. All other rewards are 0 and $\gamma = 1$.

P	P	P	P
A	B	C	G
D	E	F	G
P	P	P	P

    Table 1: Depiction of Grid World

    Notice that in this problem, the agent's immediate reward R, is a function of its current state s, its action a, and its next state s' (which could be different than the intended next state). Therefore, during value iteration, we need to account for the agent's expected reward given a state-action pair, and use a slightly different formula than the one we saw in lecture: $\forall s \in \mathcal{S}$

    $$v_0(s) = 0$$
    $$v_{k+1}(s) = \max_a \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[R(s, a, s') + \gamma V_k(s') \right]$$

    (a) (1 point) How many possible deterministic policies are there in this environment, including both optimal and non-optimal policies?

    > Answer
    >
    > 4096

(b) (1 point) After initializing the board to all zeros, in state B, which of the following actions are optimal.

Select all that apply:

☐ Up

☒ Down

☐ Left

☐ Right

☐ None of the Above

(c) (1 point) What is the value of state B in the next round (round 1) of synchronous value iteration? Round your answer to three decimal places.

Answer
0

(d) (1 point) What is the value of state C in round 1 of synchronous value iteration? Round your answer to three decimal places.

Answer
0.1

(e) (1 point) What is the value of state C in round 3 of synchronous value iteration? Round your answer to three decimal places.

Answer
0.245

(f) (1 point) What is the final value of state A once value iteration converges. Round your answer to three decimal places.

Answer
1

(g) (2 points) What is the optimal action for each state? Fill in the values in the table given below.

P	P	P	P
down	down	down	G
up	up	up	G
P	P	P	P

Table 2: Place the corresponding action in each blank cell

1.2   Q-Learning

1. In this question, we will practice using the Q-learning algorithm to play a simple two-player board game called "Connect 3". Each player, either a blue circle ◯ or a red circle ◯, takes turns marking a location in a 4x4 grid. Each time, a player has to either start from the bottom of the grid (if it is empty), or place symbols above locations with markers already placed (i.e., you can only stack the circles vertically starting from the bottom). The player who first succeeds in placing three of their marks in a column, a row, or a diagonal wins the game.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Table 3: Connect 3 Board Positions

We will model the game as follows: each board location corresponds to an integer between 1 and 16, illustrated in the graph above. Actions are also represented by an integer between 1 and 16. Playing action a results in marking the location a and an action a is only valid if the location a has not been marked by any of the players and there are no empty positions below a. We train the model by playing against an expert. The agent only receives a possibly nonzero reward when the game ends. Note a game ends when a player wins or when every location in the grid has been occupied. The reward is +1 if it wins, -1 if it loses and 0 if the game draws.

Table 4: State 1 (blue circle's turn)

To further simplify the question, let's say we are the blue circle player and it's our turn. Our goal is to try to learn the best end-game strategy given the current state of the game illustrated in table 4. The possible actions we can take are: $\{3, 9, 10, 12\}$. If we select action 3, 9, or 12, the expert will select action 10 to end the game and we'll receive a reward of -1; if we select action 10, the expert will respond by selecting action 9, which results in the state of the game in table 5. In the scenario in table 5, we can select actions $\{3, 5, 6, 12\}$. If we select actions 5, 6, or 12, then we end the game and receive a reward of +1; if we select action 3, then the expert will select action 5 to end the game and we'll receive a reward of -1.


Table 5: State 2 (blue circle's turn)

Suppose we apply a learning rate $\alpha = 0.01$ and discount factor $\gamma = 1$. The Q-values are initialized as:

$Q(1,3) = -0.4$	$Q(1,9) = -0.3$	$Q(1,10) = 0.5$	$Q(1,12) = -0.3$
$Q(2,3) = -0.4$	$Q(2,5) = 0.5$	$Q(2,6) = 0.3$	$Q(2,12) = 0.3$

Note: Showing your work in these questions is optional, but it is recommended to help us understand where any misconceptions may occur. Only your answer in the left box will be graded.

(a) (1 point) In the first episode, the agent takes action 3, receives -1 reward, and the episode terminates. Derive the updated Q-value after this episode. Remember that given the sampled experience (s, a, r, s') of (state, action, reward, next state), the update of the Q value is:

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

Note if s' is the terminal state, $Q(s', a') = 0$ for all a'. **Please round to three decimal places**.

Q(1, 3)	Work
-0.406	$Q(s,a) = Q(s,a) + \alpha\,(r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a))$ In this case, the episode is at State 1 and takes action 3 to reach a terminal state with a reward of -1. $Q(1,3) = Q(1,3) + (0.01)\,(-1 + (1)\max_{a' \in A} 0 - Q(1,3))$ $Q(1,3) = -0.4 + (0.01)\,(-1 + 0.4) = -0.406$

(b) (1 point) In the second episode, the agent takes action 10, receives a reward of 0, and arrives at State 2 (5). It then takes action 3, receives a reward of -1, and the episode terminates. Derive the updated Q-values after each of the two experiences in this episode. Suppose we update the corresponding Q-value right after every single step. **Please round to three decimal places**.

Q(1, 10)	Q(2, 3)
0.5	-0.406

Work
$Q(1,10) = Q(1,10) + (0.01)\,(0 + (1)\max_{a' \in A} 0.5 - 0.5) = 0.5$ $Q(2,3) = Q(2,3) + (0.01)\,(-1 + (1)\max_{a' \in A} Q(s', a') - (-0.4))$ $Q(2,3) = -0.4 + (0.01)\,(-1 + 0.4) = -0.406$

(c) (2 points) In the third episode, the agent takes action 10, receives a reward of 0, and arrives at State 2 (5). It then takes action 5, receives a reward of +1, and the episode terminates. Derive the updated Q-values after each of the two experiences in this episode. Suppose we update the corresponding Q-value right after every single step. **Please round to three decimal places**.

Q(1, 10)	Q(2, 5)
0.5	0.505

Work

$Q(1, 10) = Q(1, 10) + (0.01)\left(0 + (1)\max_{a' \in A} 0.5 - 0.5\right) = 0.5$

$Q(2, 5) = Q(2, 5) + (0.01)\left(1 + (1)\max_{a' \in A} Q(s', a') - Q(2, 5)\right)$
$Q(2, 5) = 0.5 + (0.01)(1 - 0.5)) = 0.505$

(d) (2 points) If we run the three episodes in cycle forever, what will be the final values of the following four Q-values. **Please round to three decimal places**.

Q(1, 3)	Q(1, 10)	Q(2, 3)	Q(2, 5)
-1	1	-1	1

Work

$Q^{t+1}(2, 5) = Q^t(2, 5) + (0.01)\left(1 + (1)\max_{a' \in A} Q(s', a') - Q^t(2, 5)\right)$
At convergence, $Q^{t+1} = Q^t = Q$
$Q(2, 5) = Q(2, 5) + (0.01)(1 + (1)(0 - Q(2, 5)))$
$(0.01)(1 + (1)(0 - Q(2, 5))) = 0$
$Q(2, 5) = 1$
Similarly,
$Q(1, 10) = 1$
$Q(1, 3) = -1$
$Q(2, 3) = -1$

(e) (2 points) What will happen if the agent adopts the greedy policy (always pick the action that has the highest current Q-value) during training? Calculate the final four Q-values in this case. **Please round to three decimal places**.

Q(1, 3)	Q(1, 10)	Q(2, 3)	Q(2, 5)
-0.4	1	-0.4	1

Work

(1, 3) and (2, 3) would never be picked in case of a greedy approach and so, their Q values would never be updated. As for the other two, they will be picked and so, their final convergence values will be similar to the previous question.

1.3   Function Approximation

1. 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
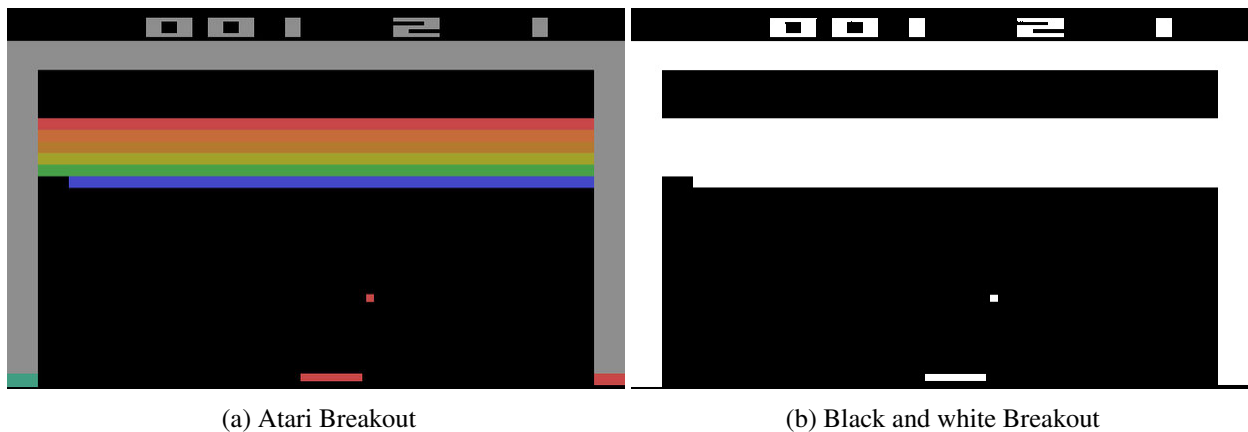
   - Move the paddle right

   - 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.

   (a) (1 point) Suppose we are dealing with the black and white version of Breakout[1] 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}

   (b) (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**
   >
   > $2^{30720} * 3$

---

[1]Play a Google-Doodle version here

(c) (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**
>
> State space size - ∞
> Q-value table - ∞

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:

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

Because we want to update for each example stochastically[2], we get the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left(q(s, a; \mathbf{w}) - q_\pi(s, a) \right) \nabla_{\mathbf{w}} q(s, a; \mathbf{w}) \tag{3}$$

However, more often then not[3] 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[4] an estimate of the action value under a greedy policy using the function approximator itself. That is to say

$$q_\pi(s, a) \approx r + \gamma \max_{a'} q(s', a'; \mathbf{w}) \tag{4}$$

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:

---

[2]This isn't really stochastic, you'll be asked in a bit why.
[3]Always in real life.
[4]Metaphorically, the agent is pulling itself up by its own bootstraps.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \left(q(s, a; \mathbf{w}) - \underbrace{\left(r + \gamma \max_{a'} q(s', a'; \mathbf{w}) \right)}_{\text{TD Target}} \right) \nabla_{\mathbf{w}} q(s, a; \mathbf{w}) \tag{5}$$

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

(d) (2 points) Let us 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:

$$q(\mathbf{s}, a; \mathbf{w}) = \mathbf{s}^T \mathbf{w}_a \tag{6}$$

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 6 encodes a table lookup)

> **Answer**
>
> Consider $\mathbf{s}$ to represent the state as a one-hot encoded vector. It's essentially a vector of size $|\mathcal{S}|$ with the value 1 correpsonding to the state index it represents. $\mathbf{w}$ is a $|\mathcal{S}|$ x $|\mathcal{A}|$ matrix. In order for the linear approximator to work like tabular Q-learning, this matrix $\mathbf{w}$ could act as the Q value table with each entry corresponding to a state, action pair.
>
> So, the linear approximator ($q(\mathbf{s}, a; \mathbf{w}) = \mathbf{s}^T \mathbf{w}_a$) would end up using the $\mathbf{s}$ to choose the state and $\mathbf{w}_a$ to find the ath entry (column) against that state (row). This shows that it is possible to have linear approximator to act as tabular Q-learning.

(e) (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, the samples received in this case are not independent but instead are highly correlated. In this case, we transition to a new state which completely depends on the previous state and the action taken at that state. Thus, online learning or SGD is not possible as it would lead to the parameters being updated such that they forget about the importance of previous states, actions/decision that brought them to this state in the first place. In order to combat this, experience replay is used. In this, a buffer of all previous states and actions is maintained so that the parameters can be trained on the transitions from the buffer.

1.4   Empirical Questions

The following parts should be completed after you work through the programming portion of this assignment (Section 2).

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
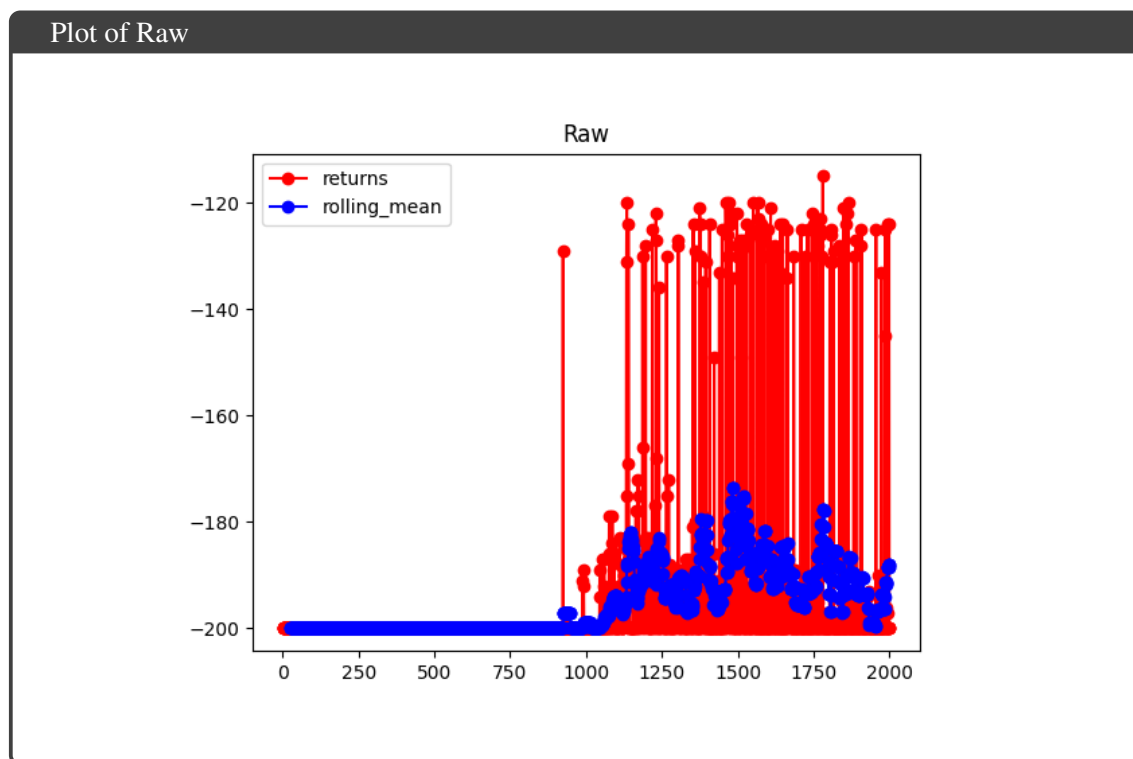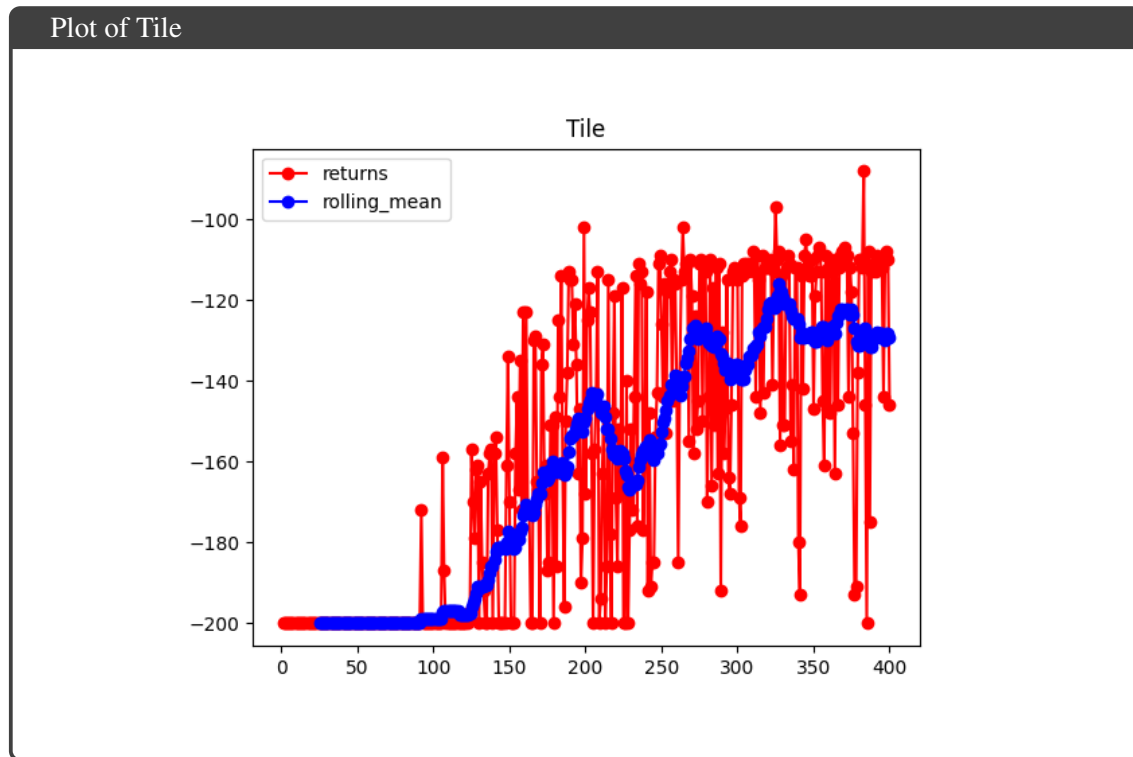
Comment

The rolling mean return for tile increases continuously with the number of episodes whereas that for raw, increases a bit and stagnates. This could be because tiling provides a way of discretizing the continuous state space but at the same time, with multiple grids, helps provide a sense of generalization.
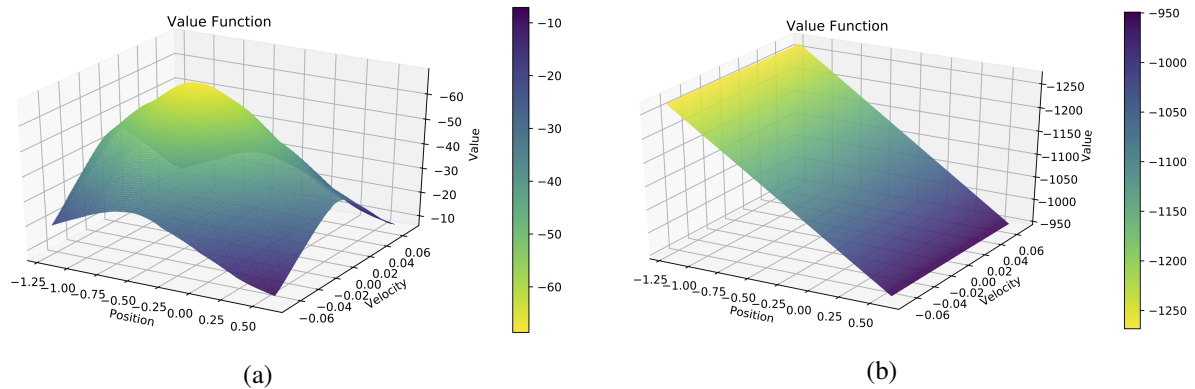
Figure 2: Estimated optimal value function visualizations for both types of features

2. (2 points) For both raw and tile features, we have run Q-learning with some good[5] 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**
>
> (b) is generated using raw features and (a) is generated using tile features. In the mountain car environment, the raw features lead to a **s** containing position and velocity. So, at the time of linear approximation, $\mathbf{s}^T w_a + b$ leads to a linear combination of position and velocity (and value function is just the max Q value for a state) which can be seen in the linear plot shown in b).
>
> In the mountain car environment, tiling leads to a higher dimensional vector representation of position and velocity which introduces non-linearity as depicted in the plot (a).

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? 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**
>
> No, I dont believe that the value function in plot b) can ever be non-linear. Plot (b) is generated using raw features. As explained previously, raw features lead to a Q value that is linearly dependent on position and velocity. Value of a state **s** is given by the maximum Q value across all possible actions from that state. Thus, Value function for raw too is linearly dependent on the position and velocity of the car and so, will always lead to a linear plot.
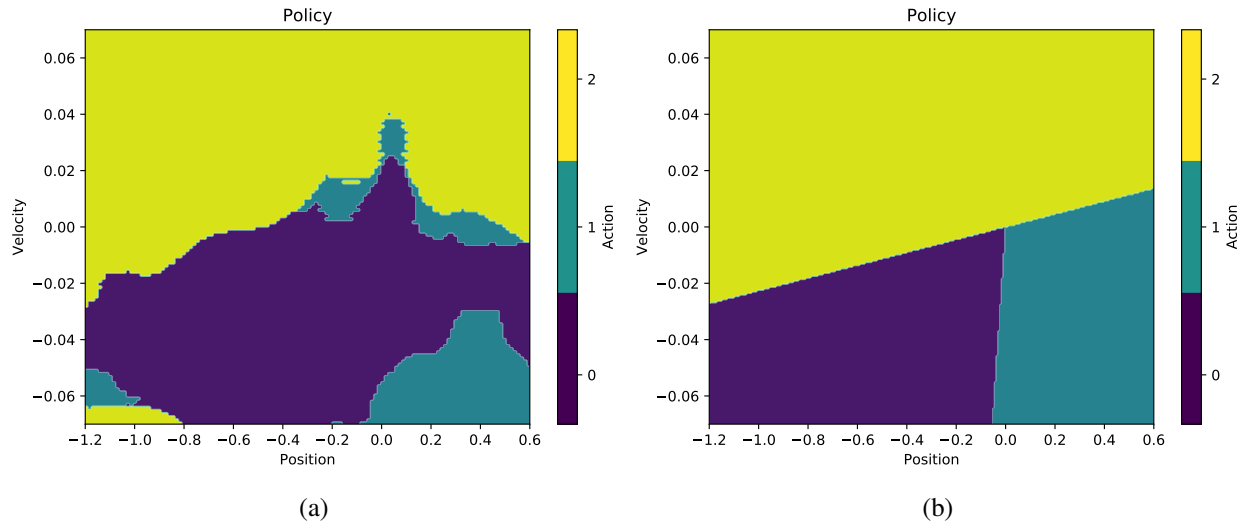
---

[5]For some sense of good.

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**
>
> (b) is using raw and (a) is using tile. Similar to previous reasonings, Q value and similarly V are linearly dependent on position and velocity in case of raw features and non-linearly in case of tile features. Policy is dependent on the sum of V for each state. Thus, in case of raw features, it would be linearly dependent on V leading to a linear edges and in case of tile features, it would be non-linearly dependent on V leading to the non-linear edges. As for intuition regarding some patches, in both cases, when the car's velocity is positive or going towards positive (that is it wants to go towards right), it continuous pushing right as the goal is in that direction. If the position of the car is towards right, in most cases, the car either tries pushing further to the right or does nothing to mazimize the return.

2   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. You may write your program in **Python, Java, or C++**. However, you should use the same language for all parts below.

2.1   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 can not 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.
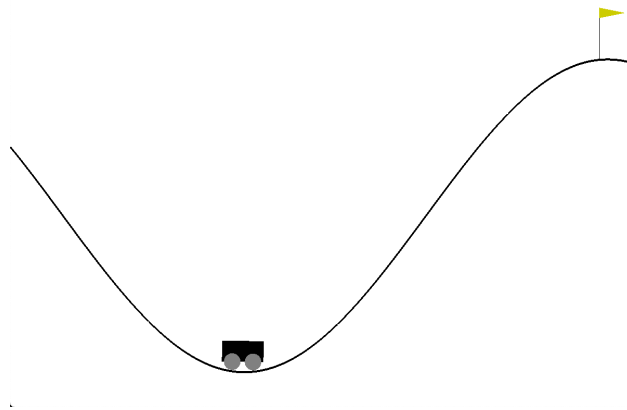

Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

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.

2.2   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[6]:

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

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're 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.

2.3   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
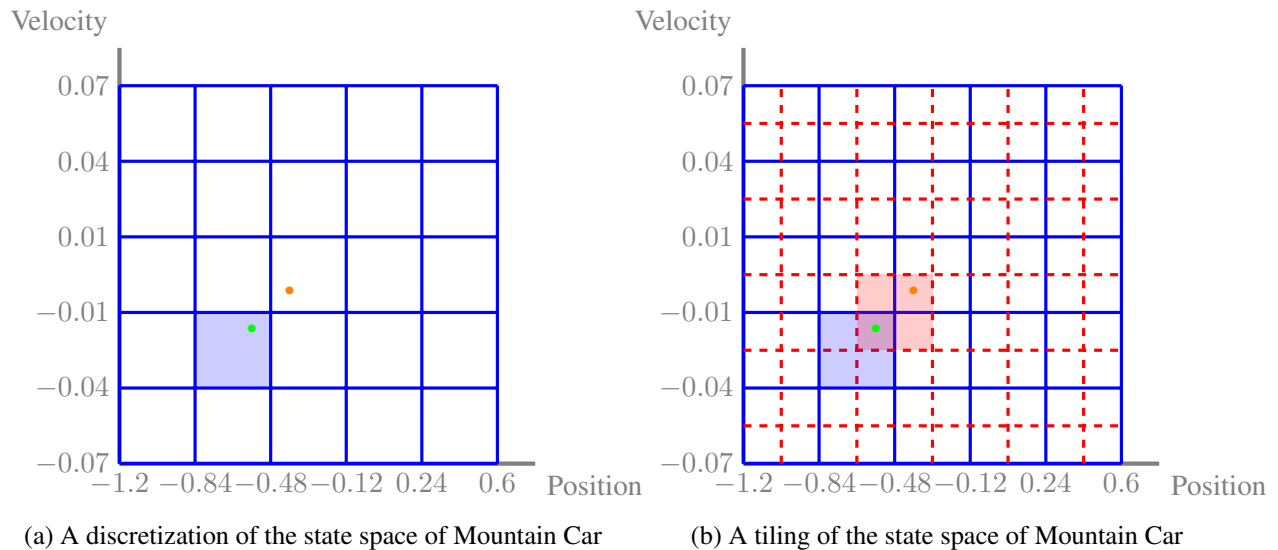
Figure 5: State representations for the states of Mountain Car

---

[6]Note that we have made the bias term explicit here, where before it was implicitly folded into $\mathbf{w}$

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.

2.4   Implementation Details

Here we describe the API to interact with the Mountain Car environment available to you in Python. The other languages will have an analagous API.

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

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

  In "tile" mode you are given indices of the tiles which are active in a sparse format: $\{T_1 \rightarrow 1, T_2 \rightarrow 1, \ldots T_n \rightarrow 1\}$ 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 $\{6 \rightarrow 1, 39 \rightarrow 1\}$.

  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.

  `fixed` is an optional argument for debugging. See Section 1.5 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.

- **[Python Only]** `render(self)`: Optionally render the environment. It is computationally intensive to render graphics, so only render a full episode once every 100 or 1000 episodes. Requires the installation of `pyglet`. This will be a no-op in Gradescope.

You should now implement your Q-learning algorithm with linear approximations as `q_learning.{py|java|cpp}`. 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 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[7], assuming your weights are stored in a $|\mathcal{S}| \times |\mathcal{A}|$ matrix.

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

For Python: `$ python q_learning.py [args...]`
For Java: `$ javac -cp "./lib/ejml-v0.33-libs/*:./" q_learning.java;`
            `java -cp "./lib/ejml-v0.33-libs/*:./" q_learning [args...]`
For C++:   `$ g++ -g -std=c++11 -I./lib q_learning.cpp; ./a.out [args...]`

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

1. `<mode>`: mode to run the environment in. Should be either ``raw'' or ``tile''.

2. `<weight_out>`: path to output the weights of the linear model.

3. `<returns_out>`: path to output the returns of the agent

4. `<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.

5. `<max_iterations>`: the maximum of the length of an episode. When this is reached, we terminate the current episode.

6. `<epsilon>`: the value ϵ for the epsilon-greedy strategy

7. `<gamma>`: the discount factor γ.

8. `<learning_rate>`: the learning rate α of the Q-learning algorithm

Example command for python users:

```
$ python q_learning.py raw weight.out returns.out \
  4 200 0.05 0.99 0.01
```

Example output from running the above command (your code won't match exactly, but should be close).

`<weight_out>`

```
-7.6610506220312296
1.3440159024460183
1.344872959883069
1.340055578403996
```

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

```
-0.0007770480987990149
0.0011306483117300896
0.0017559989206646666
```

`<returns_out>`

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

2.5 Debugging Tips

To help with debugging, we have provided the option for fixing the initialization of Mountain Car. To utilize this option, provide the additional argument `fixed = 1` when initializing Mountain Car. In this setup, the Mountain Car is initialized with `position = 0.8` and `velocity = 0`.

We recommend to first run your program with the most simple parameters and check the outputs against manually calculated values. Remember to set `<epsilon>=0` so the program is run without epsilon-greedy strategy.

Example command for python users:

```
$ python q_learning.py raw simple_weight.out simple_returns.out \
1 1 0.0 1 1
```

Once your program works, you can change one of the parameters to be slightly more complex, e.g. set `<max_iterations>=2` or `<gamma>=0.9`, and check with your manual calculations again.

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

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

Example command for python users:

```
$ python q_learning.py tile fixed_weight.out fixed_returns.out \
 25 200 0.0 0.99 0.005
```

Your output should match with the reference up till the last 4 digits.

Before submitting to Gradescope, do not forget to remove the `fixed` argument when initializing Mountain Car.

Some additional tips: If you get a `"ValueError:  high is out of bounds for int32"` this is due to python version differences. You can either update your python or change the dtype of the randint function to int64. This can be done by changing line 18 to

```
seed = rng.randint(2**32 - 1, dtype=np.int64)
```

2.6 Gradescope Submission

You should submit your `q_learning.{py|java|cpp}` to Gradescope. Note: please do not use other file names. This will cause problems for the autograder to correctly detect and run your code.

Note: For this assignment, you may make upto 30 submissions to Gradescope before the deadline, but only your last submission will be graded.

3 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**
>
> 1 - No
> 2 - No
> 3 - No