HOMEWORK 8
REINFORCEMENT LEARNING *

10-301/10-601 INTRODUCTION TO MACHINE LEARNING(SPRING 2024)
https://www.cs.cmu.edu/~mgormley/courses/10601/

OUT: Sunday, November 17
DUE: Monday, November 25
TAs: Shivi, Max, Vianna, Albert, Andra, Markov, Neural

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. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope. After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). You are only permitted to use the Python Standard Library modules and `numpy`. Ensure that the version number of your programming language environment (i.e. Python 3.9.12) and versions of permitted libraries (i.e. `numpy` 1.23.0) match those used on Gradescope. You have 10 free Gradescope programming submissions, after

*Compiled on Saturday 23rd November, 2024 at 16:25

which 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 and reference output 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?

- ● 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 instructors for this course?

- ■ Matt Gormley
- ■ Henry Chai
- □ 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 the instructors for this course?

- ■ Matt Gormley
- ■ Henry Chai
- ⊠ 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-~~6~~301

Written Questions (43 points)

1 LaTeX Point and Template Alignment (1 points)

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

   ● Yes

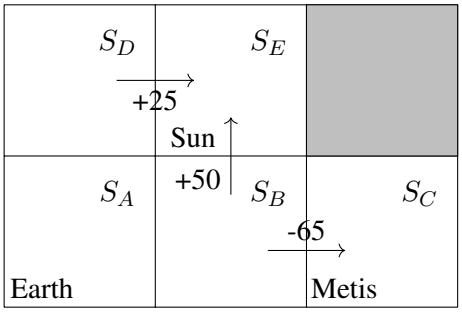
   ○ No

2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment.
   Note: Failing to answer this question will not exempt you from the 2% misalignment penalty.

   ● Yes

2 Value Iteration (12 points)

While attending an ML conference, you meet scientists at NASA who ask you to develop a reinforcement learning agent capable of carrying out a 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 $S_A, S_B, ..., S_E$ corresponding to a position in the solar system. The start state is S_A (Earth). The terminal states include both the S_E (Sun) and S_C (Metis).

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_B and takes action `left`, it moves to state S_A with 80% probability, it moves to state S_B (left of the intended direction is off the board, so the agent remains where it was) with 10% probability, and it moves to state S_E (right of the intended direction) with 10% probability.

3. It is not possible to move to the blocked state (shaded grey) since it contains another planet. If the agent's action moves them off the board or to the blocked state, it remains in the same state.

4. Non-zero rewards are depicted with arrows. Flying into the Sun from below gives positive reward $R(S_B, \texttt{a}, S_E) = +50 \ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$, since it is more fuel-efficient than flying into the sun from the left (the agent can use the gravitational field of the planet

in the blocked state and Metis). However, approaching the Sun from below has risks, as flying too close to Metis is inadvisable and gives negative reward $R(S_B, \texttt{a}, S_C) = -65\ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. Note that flying into the Sun from the left still achieves the goal and gives positive reward $R(S_D, \texttt{a}, S_E) = +25\ \forall \texttt{a} \in \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$. 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 (including terminal states) after a single round of **synchronous** value iteration in the table below. Initialize the value table $V^0(s) = 0, \forall s \in \{S_A \dots S_E\}$ 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_D 20	S_E 0	
S_A 0	S_B 33.5	S_C 0

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 $V(s) = 0, \forall s \in \{S_A \dots S_E\}$ 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 any intermediate values, including state values, when calculating your answers.**

S_D 20	S_E 0	
S_A 25.9	S_B 33.5	S_C 0

2. (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 any intermediate values, including state values, when calculating your answers.**

S_D 25	S_E 0	
S_A 30	S_B 36	S_C 0

Your solution:

S_D 26.3	S_E 0	
S_A 31.1	S_B 36.2	S_C 0

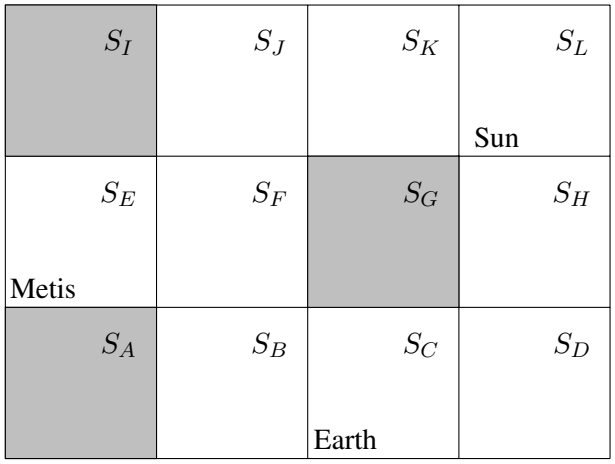
3. (3 points) What is the policy, $\pi^*(s)$, that corresponds to $V^*(s)$? Write one of `up`, `down`, `left`, or `right` for each state. If multiple actions are acceptable, choose the one that comes alphabetically first. For terminal states, write `terminal`. Ignore the blocked states.

S_D *down*	S_E *terminal*	
S_A *right*	S_B *up*	S_C *terminal*

[1] This is actually one round before the *policy* convergence, not the *value* convergence. The values we provide are the values after the second iteration, rounded to the nearest whole number for ease of calculation.

3 Q-Learning (9 points)

Let's consider an environment that is similar to the grid world we saw before, but has more states:



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 $\gamma = 0.9$ and learning rate $\alpha = 0.1$.

We will go through three iterations of Q-learning in this section. Initialize $Q(s, a)$ as below:

$a \setminus s$	S_A	S_B	S_C	S_D	S_E	S_F	S_G	S_H	S_I	S_J	S_K	S_L
Up	0.4	0.1	0.1	0.7	0.0	0.9	0.7	0.8	0.0	0.1	0.8	0.8
Down	1.0	0.8	0.2	0.5	0.1	0.2	0.7	0.2	1.0	0.9	0.1	0.3
Left	0.9	0.4	0.3	0.4	0.9	0.6	0.5	0.1	0.2	0.3	0.9	0.1
Right	0.3	0.8	0.3	0.2	0.0	0.2	0.2	0.3	0.9	0.4	0.2	0.3

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

   ☐ `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? If needed, round your answer to the fourth decimal place.

$Q(S_C, \texttt{right})$
0.6300

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

$Q(S_C, \texttt{right})$
0.3330

4. (1 point) **Select all that apply:** 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 +25, and the run terminates. What is the new value for $Q(S_H, \texttt{up})$, assuming the update for deterministic transitions? If needed, round your answer to the fourth decimal place.

$Q(S_H, \texttt{up})$
25.7200

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

$Q(S_H, \texttt{up})$
3.2920

7. (1 point) **Select all that apply:** You start from state S_C again since the previous run terminated. Assume you 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 -50, and the run terminates. What is the new value for $Q(S_F, \texttt{left})$, assuming the update for deterministic transitions? If needed, round your answer to the fourth decimal place.

$Q(S_F, \texttt{left})$
-49.1900

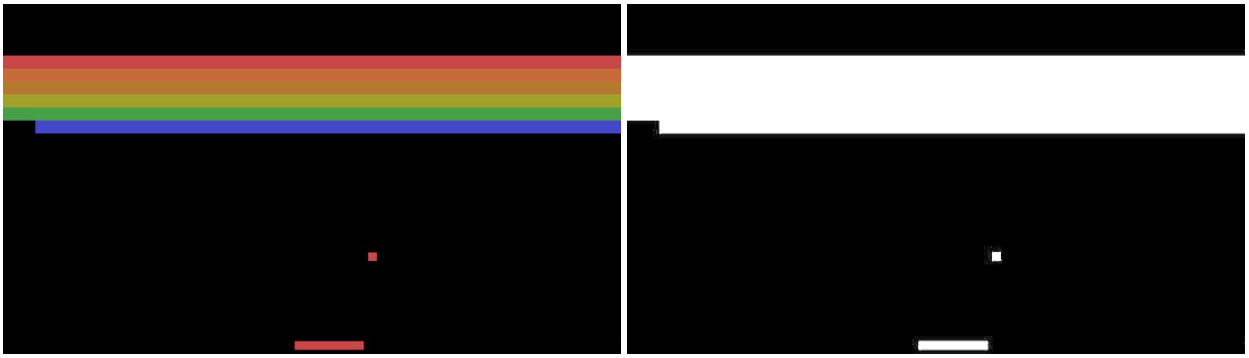
9. (1 point) What is the new value for $Q(S_F, \texttt{left})$, using the temporal difference error update? If needed, round your answer to the fourth decimal place.

$Q(S_F, \texttt{left})$
-4.3790

4 Deep Q-Learning (7 points)

In this question we will motivate learning a parametric form 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? Express your answer in terms of exponents if needed.

   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? Express your answer in terms of exponents if needed.

   Answer

   $3 \cdot 2^{30720}$

[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 $(1, 1, 1)$.

   Is it possible to represent all our Q-values with a table holding one value for every (state, action) pair?

   Answer

   No, it is not possible, as each pixel has infinite number of possible pixel values, there exists infinite number of states.

   Suppose rather than storing many separate Q-values for similar states, we want to share information between states. Instead of individual entries in a table, we can learn parameters $\mathbf{w}$ that parameterize some approximation $q(s, a; \mathbf{w})$ of the true Q-values.

   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 $\mathbf{w}$. 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:

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

[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 $\mathbf{s}$, and for each action, we learn a linear approximation from states to Q-values. That is:

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

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

We can set state s to be a one-hot vector where each position corresponds to a unique state, and weight vector w_a stores the Q values for all states when undertaking action 'a'. For example, set s = [0, 1, 0, 0, 0], representing the second state, and w_a = [3.0, 1.0, 9.0, 5.0, -1.0], then q(s, a; w) = $w_a^T s$ = 1.0, representing the Q value for the second state. This describes how the linear approximation implements the table lookup.

5. (2 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 here does not support independent and identically distributed, as consecutive states are highly correlated within the assumption of Markov Decision Process, where the next state s_{t+1} depends on current state s_t and action a_t. To combat the issue of SGD, we can use the replay buffer technique which maintains a replay buffer D = $\{e_1, e_2, ..., e_N\}$. Every time we sample an example, we pick it up from D unifromly at random, and apply Q-learning update, then add a new experience to D. In this way, we can use uncorrelated samples and get better performance.

5 Empirical Questions (14 points)

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

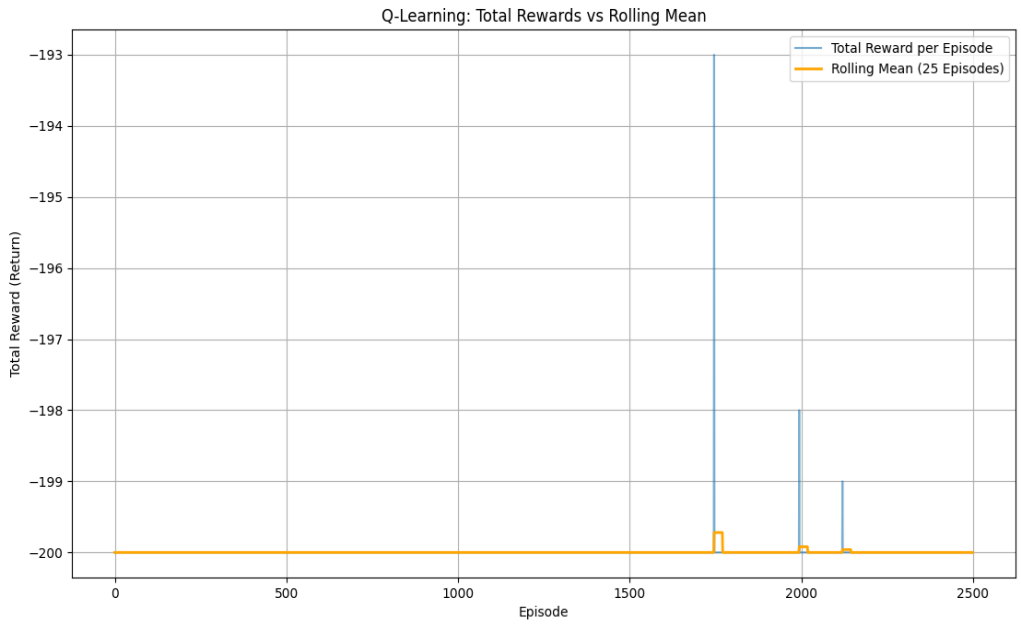
1. (4 points) Run Q-learning on the mountain car environment using both tile and raw features.

   For the raw features: run for 2500 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.999, and a learning rate of 0.001.

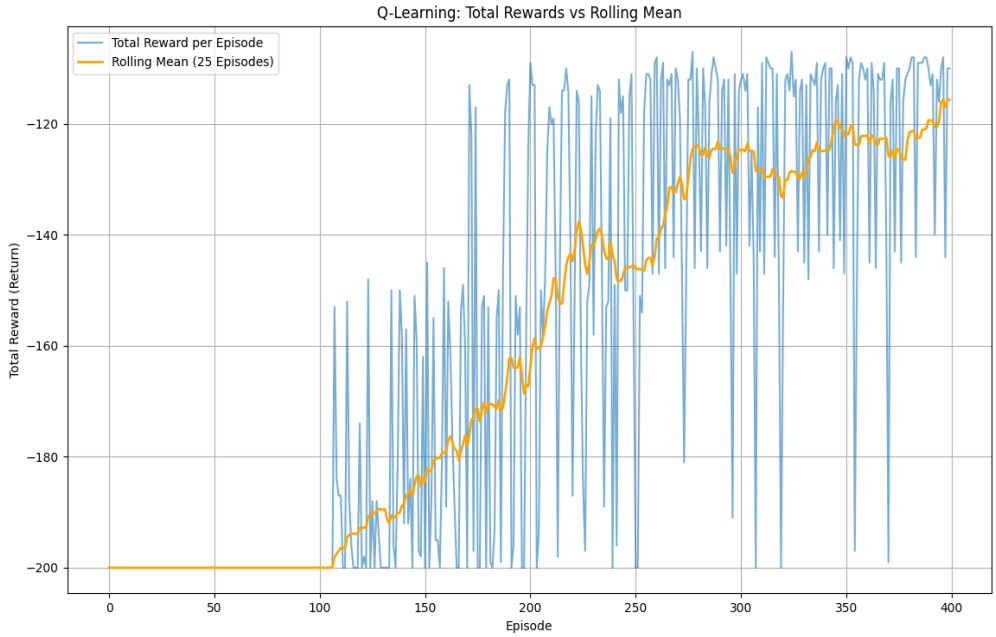
   For the tile features: run for 400 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.99, and a learning rate of 0.00005.

   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



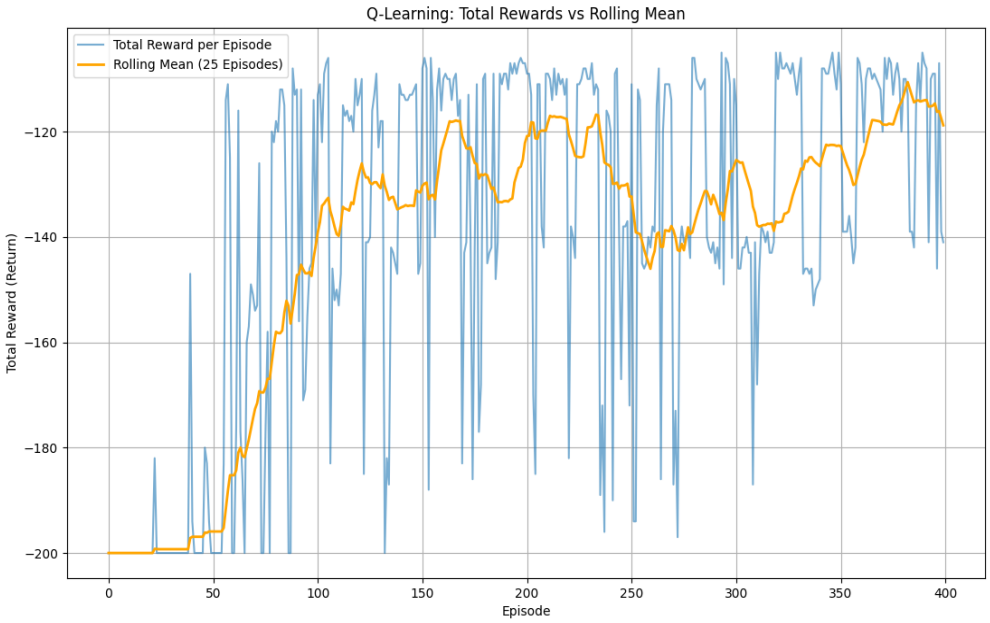
Comment

The curve of raw feature shows that the rewards remain consistently low with sporadic spikes, and the rolling mean is nearly flat, indicating that the agent is not able to learn an effective policy for solving the task with raw features. On the other hand, the curve of tile feature shows obvious upward trend in both total rewards and the rolling mean, indicating that the agent's performance on the task is improving. Besides, the fluctuations of total rewards decrease as the episode going, showing the performance is becoming more stable as learning continues.

2. (4 points) Now, run Q-learning on the mountain car environment (tiled features) with the replay buffer. Use the following hyperparameters: 400 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.99, learning rate of 0.00005, buffer size of 1000, and batch size of 3.

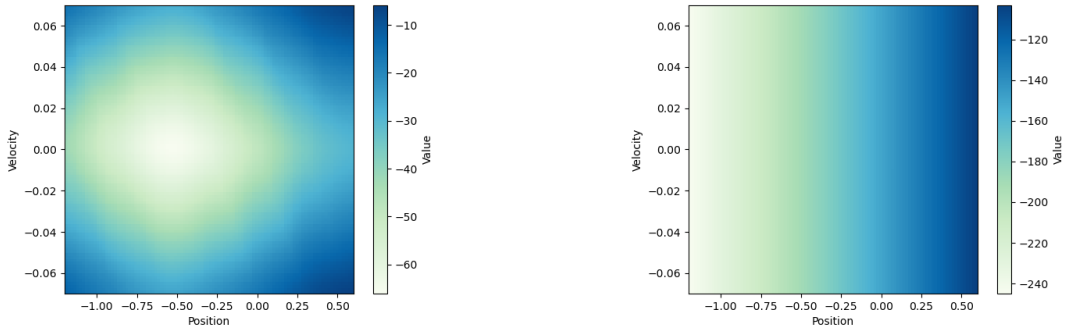
   Plot the return (sum of all rewards in an episode) per episode on a line graph. On the same graph, also plot the rolling mean over a 25 episode window. Comment on the difference between your plots for the tiled Mountain Car environment rewards with and without the replay buffer.

Plot of Tile with Replay Buffer



Comment

The curve of the total reward with replay buffer shows faster convergence than the curve without replay buffer. Besides, the fluctuations in total reward becomes smaller with replay buffer, and the rolling mean starts the upward trend at earlier episode than the one without replay buffer. I think it is because the replay buffer allows the agent to repeatedly sample batches of uncorrelated experiences to provide more diverse examples for learning, achieving better generalization by holding independent and identical distribution property needed by SGD optimization and avoiding over-fitting to the most recent episodes.



(a) (b)

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

3. (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 for deep Q-learning. Explain your reasoning. In addition, interpret each of these plots in the context of the mountain car environment.

Answer

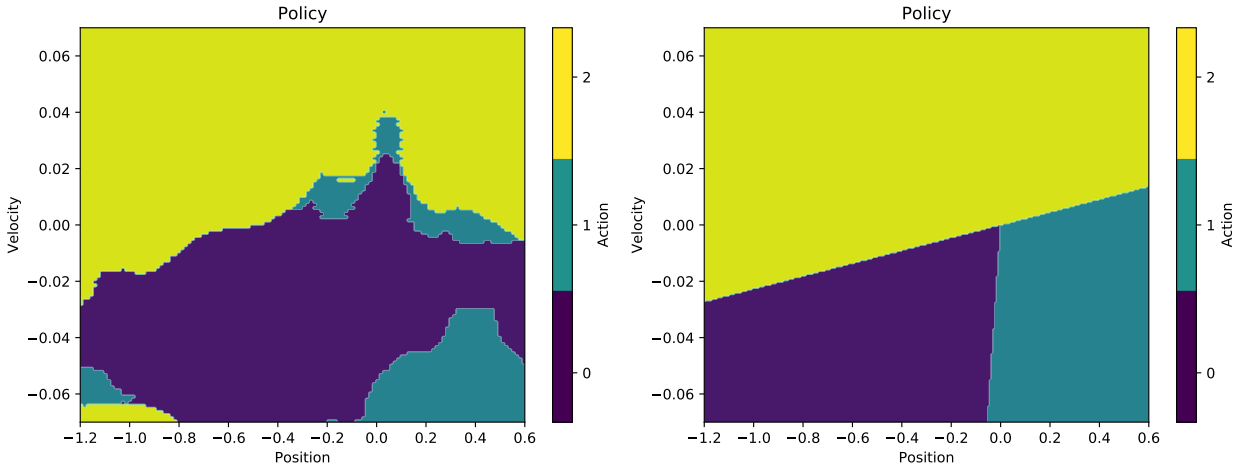
For plot (a), it is likely to use the tile features because it shows smooth gradient across the state space, allowing the agent to learn how to distinguish the importance of different positions and velocities and thus reach the goal mountain car task.
For plot (b), it is more likely to use the raw features because it shows ambiguous and unorganized goal that could let the agent to learn and reach the goal. Without clear optimization direction because of the lack of generalization in feature representation, the agent is unable to understand which states are good or bad, resulting in poor learning and results.

4. (2 points) We see that Figure 2b seems to look like a linear function of the position and velocity. 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 $y = \mathbf{A}\mathbf{x} + \mathbf{b}$)? Briefly justify your answer in 2 sentences or less.

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

Answer

No, the value function cannot be nonlinear because the Q-values is formulated by $W_a \cdot s$ (with intercept term folded into W_a), which is a linear function, and the value function, $V(s) = \texttt{max}_\texttt{a} Q(s, a)$, would be piecewise-linear function that comprises linear segments according to the action with the highest Q-value in state space.



(a) (b)

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

5. (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 for deep Q-learning. Explain your reasoning.

Answer

For plot (a), it is more likely be generated using tile features, as it shows detailed and clear decision boundaries for actions among the state spaces, even the small regions have clear decision boundary, suggesting that the agent has learned a robust policy using informative feature representation provided by the tile features.
For plot (b), it is more likely be generated using raw features, as the decision boundaries are much simpler and coarse, meaning that the agent only learns basic approximation of the optimal policy, failing to capture the complex state-action relationship due to the lack of informative and representative features for the task.

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

1. No.
2. No.
3. No.

7 Programming [68 Points]

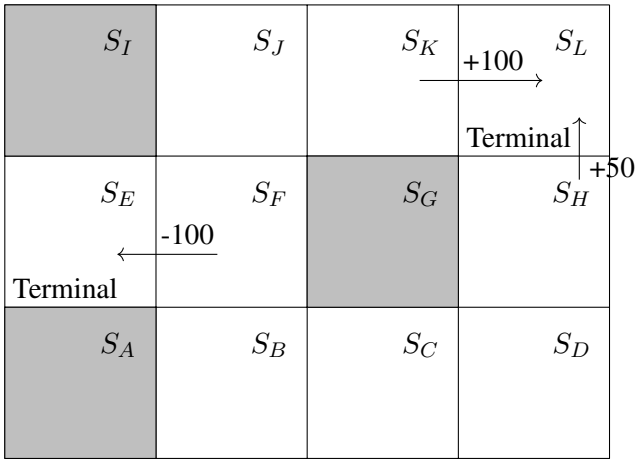
Your goal in this assignment is to implement Q-learning with linear function approximation to solve the mountain car environment. You will implement all of the functions needed to initialize, train, evaluate, and obtain the optimal policies and action values with Q-learning. In this assignment we will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

7.1 Specification of Grid World

In this assignment, you are provided with code that fully defines the GridWorld environment. In GridWorld, you navigate a 3x4 grid to reach specified goal states, navigating around obstacles and possibly encountering rewards or penalties along the way. Your objective is to find the optimal path to the goal state while maximizing your total reward.

The GridWorld environment is represented by a grid of cells, where each cell corresponds to a different state in the environment. The agent can move in four directions: up, down, left, and right. Certain cells are designated as blocked, and the agent cannot move through these. There are also special terminal states that end the episode when reached.

The state of the environment is represented by the agent's current position on the grid, defined by its row and column indices. Actions that the agent can take at any state are defined as $0, 1, 2, 3$, corresponding to the actions: (0) move up, (1) move down, (2) move left, and (3) move right.



7.2 Specification of Mountain Car

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.

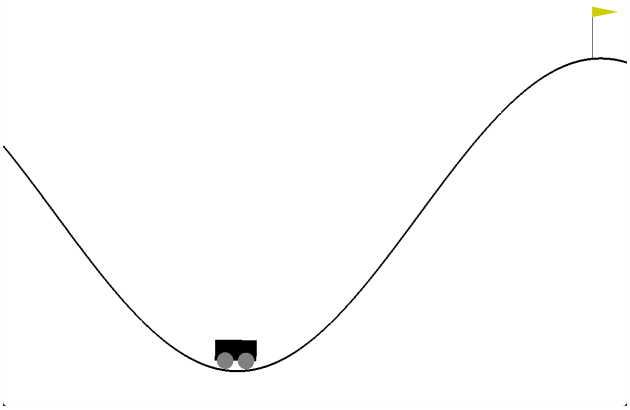


Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

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.

7.3 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

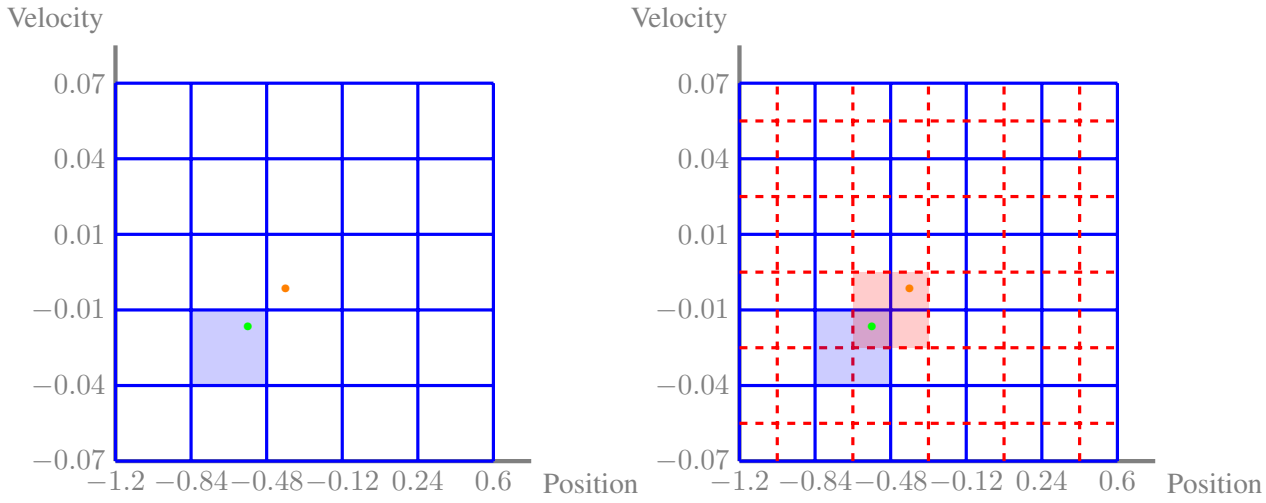
$$q(\mathbf{s}, a; \mathbf{w}) = \mathbf{w}_a^T \mathbf{s} + b_a.$$

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.

7.4 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.

7.5 Replay Buffer

As detailed in Section 7.3, we can update our parameters in each iteration by having our agent take a step in the environment, observing the results, and then applying a stochastic update rule to $\mathbf{w}$. The downside to this approach is that the samples which we use to update $\mathbf{w}$ are not identically and independently distributed, an assumption that allows for a more reliable estimation of the true gradient. For example, suppose the agent starts in state s_0 and takes action a_0, resulting in a reward of r_0 and a new state of s_1. We can update $\mathbf{w}$ with this sample, but in the next iteration, the agent's state will be s_1. Since s_1 is dependent on the state and action taken in the previous iteration, then the two samples are highly correlated.

To combat this, we can add an experience replay buffer. After taking a step in the environment, rather than immediately updating $\mathbf{w}$, we can **store the experience consisting of (s, a, r, s') to our replay buffer**. Then instead of using the experience from the current iteration to update $\mathbf{w}$, we can **use a randomly sampled experience from the replay buffer instead**. In this way, consecutive updates to $\mathbf{w}$ no longer use correlated samples and we can achieve a better estimation of the true gradient.

In practice, we can **randomly sample a batch of experiences from the replay buffer** and sequentially update $\mathbf{w}$ using each experience in the batch. Thus it is important that **batch sampling, as well as any parameter updates, only occur when the replay buffer has accumulated enough experiences for a full batch**. We also implement the replay buffer as a queue with a maximum buffer size, so that the first experiences added are also the first ones evicted to make room for new experiences.

7.6 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 as a vector $[\texttt{position}, \texttt{velocity}]^T$.

  In "tile" mode you are given a binary vector where the i-th index is 1 if the i-th tile is active in the tiling. All other tile indices are assumed to map to 0. For example the state representation of the example in Figure 5b would become $[0, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0]^T$, where indices 6 and 39 are 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.

  `debug` is an optional argument for debugging. See Section 7.7 for more details.

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

- `step(action)`: Take a step in the environment with the given action. `action` must be an integer in the range $[0, \texttt{env.action\_space})$, where `env` is the environment instance. For the Mountain Car environment, `env.action_space` is 3, since the valid actions are 0, 1, and 2. `step(action)` returns a tuple of $(\texttt{state}, \texttt{reward}, \texttt{done})$ 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 to include a bias!) and use the epsilon-greedy strategy for action selection. Update the parameters with or without experience replay as indicated by a command line argument.**

Additionally, to avoid numerical precision errors, please ensure that your Q-values throughout your program are rounded to 5 decimal places. This is already handled for you in the starter code by the `@round_output(5)` decorator[5] above the `Q(W, state, action)` function; the body of this function is left for you to complete. If you choose not to use the starter code, make sure that your code still does this rounding:

```
Qvalue = <some code to calculate Q-values>
Qvalue = np.round(Qvalue, 5)
```

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 the weights of the model to a file. This output file should have the following format:

```
bias_action_0 weight_action_0_1 weight_action_0_2 ...
bias_action_1 weight_action_1_1 weight_action_1_2 ...
...
```

Above, each line corresponds to the weights for that action. For example, the first line contains the bias and the weights for action 0, the second line contains the bias and the weights for action 1, and so on. A space separates the parameters in each line, and each line is terminated by a newline character `"\n"`.

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>` `<replay_enabled>` `<buffer_size>` `<batch_size>`. 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] You can install it by typing `pip install pyglet` in your shell.

[5] You don't need to know how decorators work for this class, but you can read more about them here if you're interested.

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.
10. <replay_enabled>: flag that indicates whether experience replay should be used during the parameter update. A 1 indicates that replay is enabled, and a 0 indicates that replay is disabled.
11. <buffer_size>: the replay buffer size. If replay is enabled, then the replay buffer should hold a maximum of buffer_size experiences.
12. <batch_size>: the batch size for experience replay. If replay is enabled, then this is the number of experiences that should be sampled for the parameter update in each iteration.

Example command:

```
$ python q_learning.py mc raw mc_params1_weight.txt mc_raw_returns.txt \
 4 200 0.05 0.99 0.01 1 500 3
```

7.7 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 and without experience replay. 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_params1_weight.txt \
  gw_params1_returns.txt 1 1 0.0 1 1 0 0 0
```

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_params2_weight.txt gw_params2_returns.txt \
  3 5 0.0 0.9 0.01 0 0 0
```

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.txt and mc_returns.txt 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
- <replay_enabled>: 0
- <buffer_size>: 0
- <batch_size>: 0

Example command:

```
$ python q_learning.py mc tile mc_params2_weight.txt \
 mc_params2_returns.txt 25 200 0.0 0.99 0.005 0 0 0
```

For your convenience, we have provided a file `check.py` in the handout that will generate and compare your reward and weight outputs to all the reference outputs provided in the `reference_output` folder. See the comment at the top of the file for instructions on running these checks.

For all checks:

```
$ python -m unittest check
```

For a specific example (in this case Mountain Car with tile features, the command given earlier on this page):

```
$ python -m unittest check.MCTile
```

7.8 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.