HOMEWORK 8: REINFORCEMENT LEARNING

10-601 Introduction to Machine Learning (Fall 2020)
https://www.cs.cmu.edu/~10601/
DUE: Thursday, Dec 03, 2020 11:59 PM

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 1, 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: https://www.cs.cmu.edu/~10601
- **Late Submission Policy:** See the late submission policy here: https://www.cs.cmu.edu/~10601
- **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, we will be using Gradescope (https://gradescope.com/). Please use the provided template. Submissions must be written in LaTeX. Regrade requests can be made, however this gives the staff the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted. Each derivation/proof should be completed in the boxes provided. If you do not follow the t 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 3.6.9, OpenJDK 11.0.5, g++ 7.4.0) and versions of permitted libraries (e.g. `numpy` 1.17.0 and `scipy` 1.4.1) match those used on Gradescope. You have unlimited Gradescope programming submissions. However, we recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting you 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 Piazza.

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 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 0.38. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.38-libs/*:linalg_lib/nd4j-v1.0.0-beta7-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-beta7. When compiling and running your code, we will add the additional command line argument `-cp "linalg_lib/ejml-v0.38-libs/*:linalg_lib/nd4j-v1.0.0-beta7-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 3.3.7. 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 Piazza Resources page 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://deeplearning4j.org/docs/latest/nd4j-overview
[c] http://eigen.tuxfamily.org/

Written Questions (32 points)

1. Value Iteration [6 points]

In this question you will carry out value iteration by hand to solve a maze. A map of the maze is shown in the table below, where 'G' represents the goal of the agent (it's the terminal state); 'H' represents an obstacle; the zeros are the state values V(s) that are initialized to zero.

0	0	G
H	0	H
0	0	0

Table 1: Map of the maze

The agent can choose to move up, left, right, or down at each of the 6 states (the goal and obstacles are not valid initial states, "not moving" is not a valid action). The transitions are deterministic, so if the agent chooses to move left, the next state will be the grid to the left of the previous one. However, if it hits the wall (edge) or obstacle (H), it stays in the previous state. The agent receives a reward of -1 whenever it takes an action. The discount factor γ is 1.

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

Answer

d

(b) (3 points) Compute the state values after each round of synchronous value iteration updates on the map of the maze before convergence. For example, after the first round, the values should look like this:

-1	-1	G
H	-1	H
-1	-1	-1

Table 2: Value function after round 1

??	??	??
??	??	??
??	??	??

Table 3: Round 2

??	??	??
??	??	??
??	??	??

Table 4: Round 3

??	??	??
??	??	??
??	??	??

Table 5: Round 4

(c) (2 points) Which of the following changes will result in the same optimal policy as the settings above?

Select all that apply:

☐ The agent receives a reward of -10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 0.9.

☐ The agent receives a reward of 10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 0.9.

☐ The agent receives a reward of 10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 1.

☐ The agent receives a reward of 10 when it takes an action that reaches G and receives a reward of -1 whenever it takes an action that doesn't reach G. Discount factor is 1.

☐ None of the above.

2. Q-learning [8 Points]

In this question, we will practice using the Q-learning algorithm to play tic-tac-toe. Tic-tac-toe is a simple two-player game. Each player, either X (cross) or O (circle), takes turns marking a location in a 3x3 grid. 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

Table 6: tic-tac-toe board positions

We will model the game as follows: each board location corresponds to an integer between 1 and 9, illustrated in the graph above. Actions are also represented by an integer between 1 and 9. 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. 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.

O	X	
O	O	X
		X

Table 7: State 1 (circle's turn)

To further simplify the question, let's say we are the 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 7. The possible actions we can take are the positions that are unmarked: $\{3, 7, 8\}$. If we select action 7, the game ends and we receive a reward of +1; if we select action 8, the expert will select action 3 to end the game and we'll receive a reward of -1; if we select action 3, the expert will respond by selecting action 7, which results in the state of the game in table 8. In this scenario, our only possible action is 8, which ends the game and we receive a reward of 0.

O	X	O
O	O	X
X		X

Table 8: State 2 (circle's turn)

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

$$Q(1, 3) = 0.5$$
$$Q(1, 7) = -0.4$$
$$Q(1, 8) = -0.6$$
$$Q(2, 8) = 0.7$$

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 7, 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, 7)	Work

(b) (1 point) In the second episode, the agent takes action 8, receives a reward of -1, and the episode terminates. Derive the updated Q-value based on this episode. **Please round to three decimal places**.

Q(1, 8)	Work

(c) (2 points) In the third episode, the agent takes action 3, receives a reward of 0, and arrives at State 2 (8). It then takes action 8, receives a reward of 0, 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, 3)	Q(2, 8)

Work

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

Q(1, 3)	Q(1, 7)	Q(1, 8)	Q(2, 8)

Work

(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**.

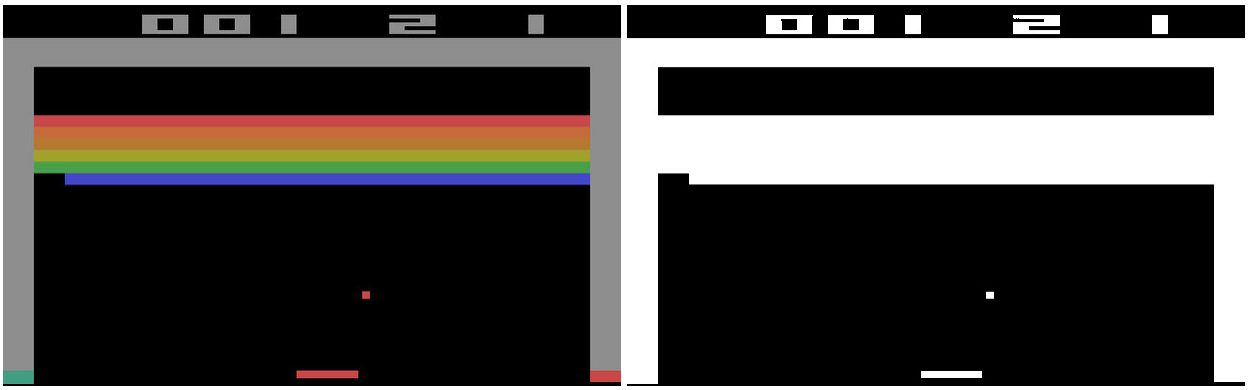
Q(1, 3)	Q(1, 7)	Q(1, 8)	Q(2, 8)

Work

3. 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.

(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

(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?

[1] Play a Google-Doodle version here

Answer

(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

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:

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

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

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

[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.

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:

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

(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

(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

4. Empirical questions [10 Points]

The following parts should be completed after you work through the programming portion of this assignment (Section 1).

(a) (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.

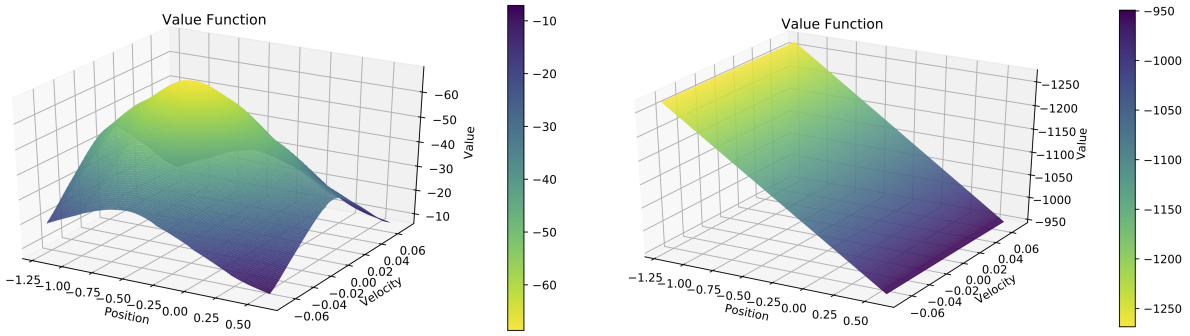
For the tile features: run for 400 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.99, and a learning rate of 0.00005.

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



(a) (b)

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

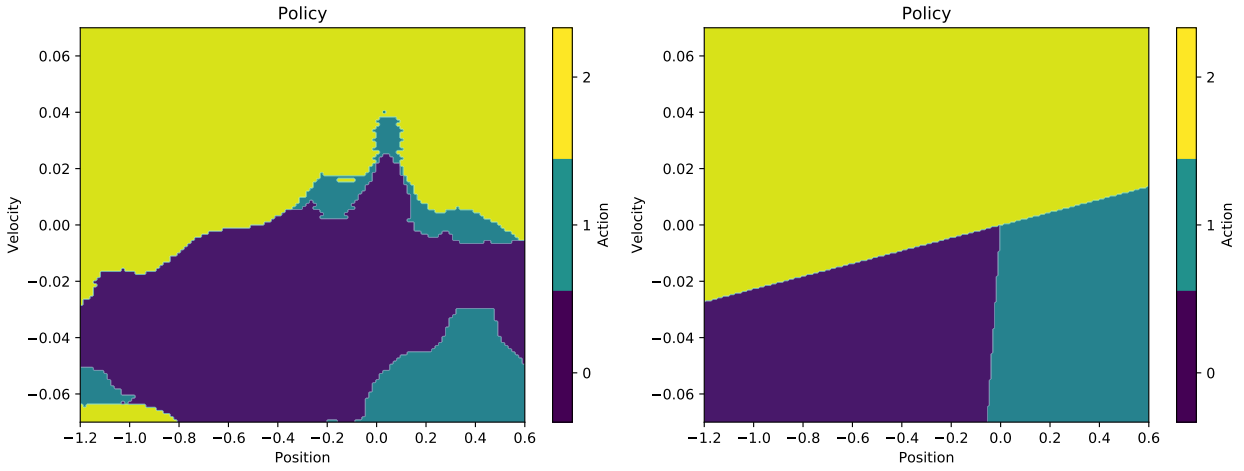
(b) (2 points) For both raw and tile features, we have run Q-learning with some good[5] parameters and created visualizations of the value functions after many episodes. For each plot in Figure 2, write down which features (raw or tile) were likely used in Q-learning with function approximation. Explain your reasoning. In addition, interpret each of these plots in the context of the mountain car environment.

Answer

(c) (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

[5]For some sense of good.



(a) (b)

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

(d) (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

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

Answer

1 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.

1.1 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.

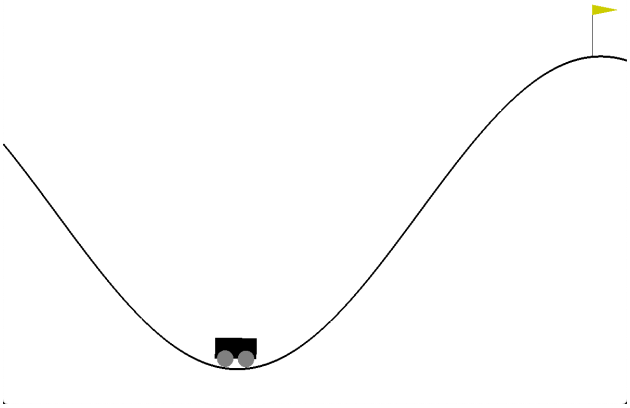


Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

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

The actions that you may take at any state are $\{0, 1, 2\}$ which respectively correspond to (0) pushing the car left, (1) doing nothing, and (2) pushing the car right.

1.2 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 we're 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 get 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:

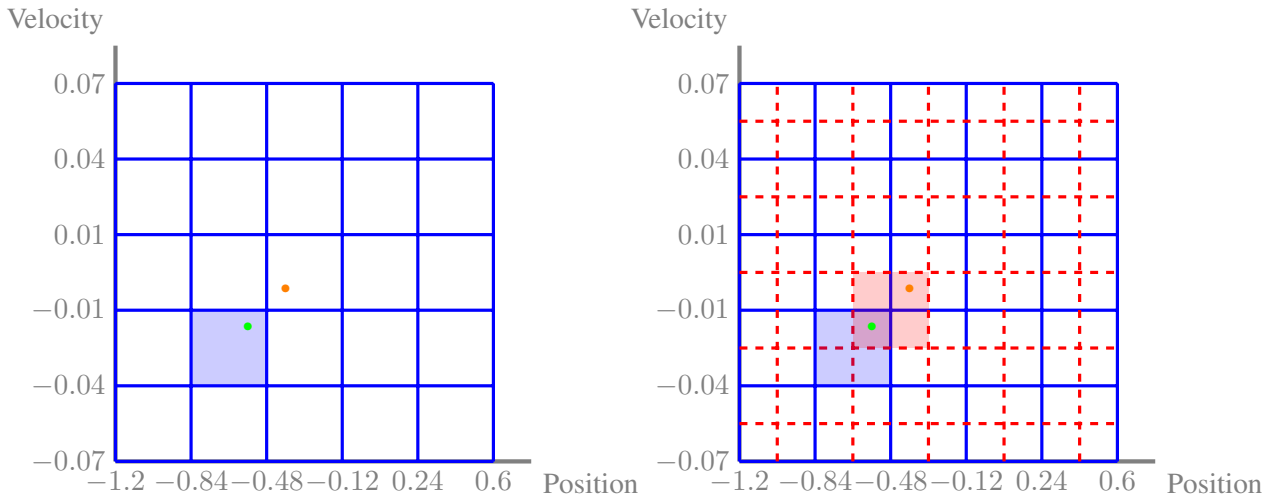
$$q(\mathbf{s},a;\mathbf{w}) = \mathbf{s}^T\mathbf{w}_a + b$$

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 as well. For the purpose of testing, we will test two cases: $\epsilon = 0$ and $0 < \epsilon < 1$. When $\epsilon = 0$, the program becomes deterministic and your output have to match our reference output accurately. In this case, if there is a draw in the greedy action selection process, pick the action represented by the smallest number. For example, if we're at state s and $Q(s,0) = Q(s,2)$, then take action 0. And when $0 < \epsilon < 1$, your reference output will need to fall in a certain range that we determine by running exhaustive experiments based on the input parameters.

1.3 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.

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



(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 $\{6\}$. 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 $\{6, 39\}$. 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.

1.4 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 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.

`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.

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

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
-0.00077704809877990149
0.0011306483117300896
0.0017559989206646666
```

<returns_out>

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

1.5 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)
```

1.6 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 unlimited submissions to Gradescope before the deadline, but only your last submission will be graded.