

# PROGRAMMING ASSIGNMENT 7: REINFORCEMENT LEARNING

10-301/10-601 Introduction to Machine Learning (Summer 2025)

<https://www.cs.cmu.edu/~hchai2/courses/10601/>

OUT: Tuesday, June 10th

DUE: Friday, June 13th

TAs: Andy, Canary, Michael, Sadrishya, and Neural the Narwhal

## START HERE: Instructions

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., “Jane explained to me what is asked in Question 2.1”). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information: <https://www.cs.cmu.edu/~hchai2/courses/10601/>
- **Late Submission Policy:** See the late submission policy here: <https://www.cs.cmu.edu/~hchai2/courses/10601/>
- **Submitting your work:**
  - **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 3.12.9) and versions of permitted libraries (e.g. `numpy` 2.2.4, `matplotlib` 3.10.0) match those used on Gradescope. You have a **total of 10 Gradescope programming submissions**. Use them wisely. In order to not waste code submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before any Gradescope coding submission.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. You must typeset your submission using  $\text{\LaTeX}$ . If your submission is misaligned with the template, 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). Each derivation/proof should be completed in the boxes provided. Do not move or resize any of the answer boxes. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.

For multiple choice or select all that apply questions, shade in the box or circle in the template document corresponding to the correct answer(s) for each of the questions. For  $\text{\LaTeX}$  users, replace `\choice` with `\CorrectChoice` to obtain a shaded box/circle, and don't change anything else.

## Instructions for Specific Problem Types

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

**Select One:** Who taught this course?

- ☒ 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?

- ☒ 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
- ☐ I don't know

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

**Select all that apply:** Which are scientists?

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

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

**Fill in the blank:** What is the course number?

10-601

10-~~6~~301

## Written Problems (14 points)

### 1 Empirical Questions

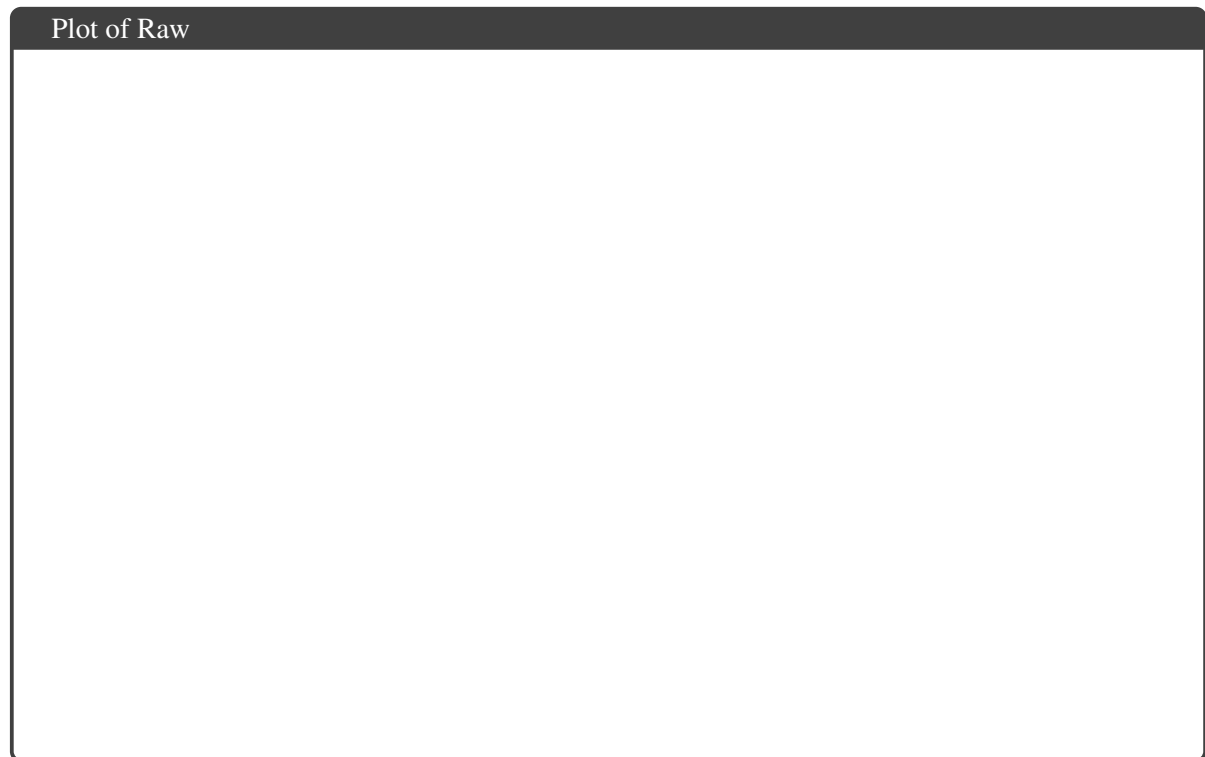
The following parts should be completed after you work through the programming portion of this assignment (Section 3).

1. (4 points) Run Q-learning on the mountain car environment using both tile and raw features. You should not use the replay buffer for this question.

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

For the tile features: run for 400 episodes with max iterations of 200,  $\epsilon$  set to 0.05,  $\gamma$  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.



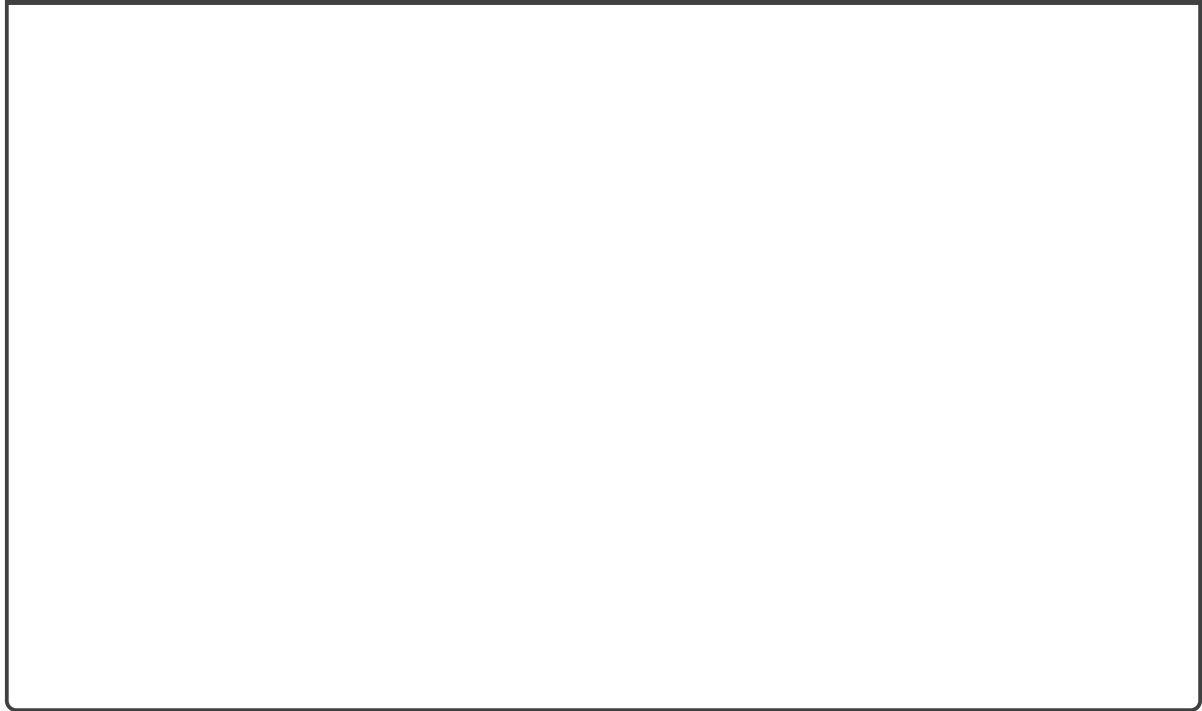
Plot of Tile

Comment

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,  $\epsilon$  set to 0.05,  $\gamma$  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



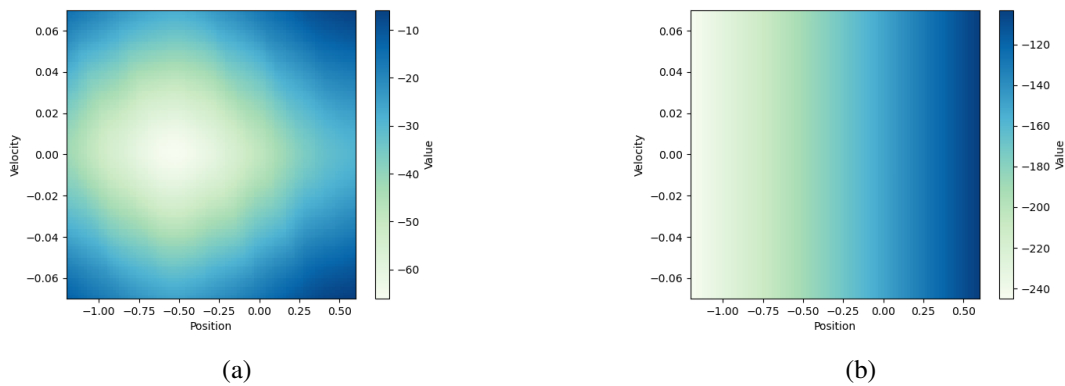


Figure 1: Estimated optimal value function visualizations for both types of features

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 1, 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

4. (2 points) We see that Figure 1b 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  $y = \mathbf{Ax} + \mathbf{b}$ )? Briefly justify your answer in 2 sentences or less.

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

Answer

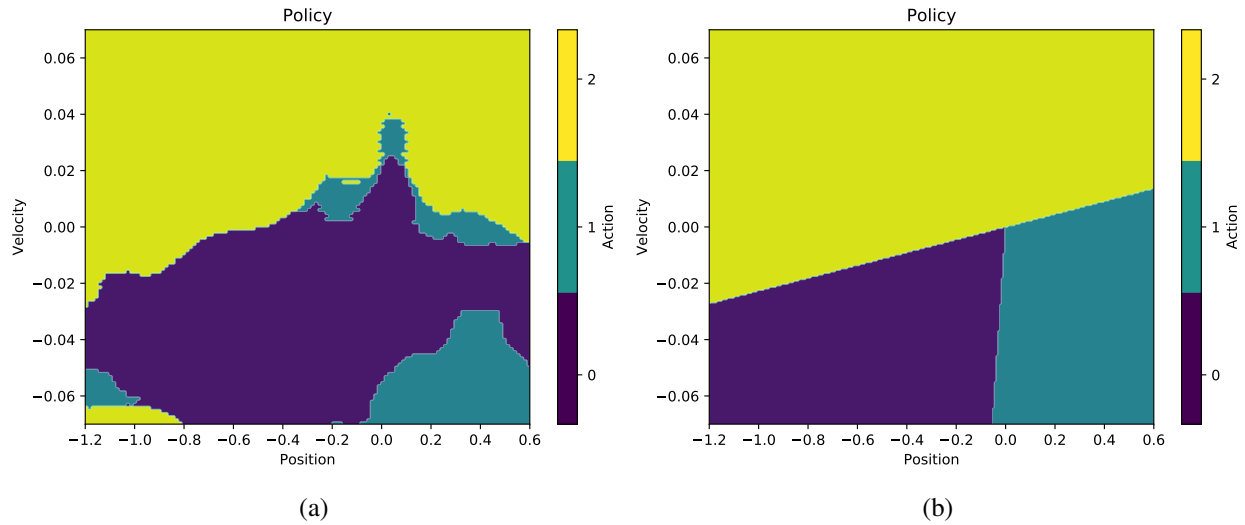


Figure 2: Estimated optimal policy visualizations for both types of features

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 2, write down which features (raw or tile) were likely used for deep Q-learning. Explain your reasoning.

Answer

## 2 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



### 3 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.

#### 3.1 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.

$S_I$	$S_J$	$S_K$	$S_L$ $\xrightarrow{+100}$ Terminal $\uparrow$
$S_E$	$S_F$	$S_G$	$S_H$ $\uparrow +50$
Terminal	$\xleftarrow{-100}$		
$S_A$	$S_B$	$S_C$	$S_D$

### 3.2 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 3. 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.

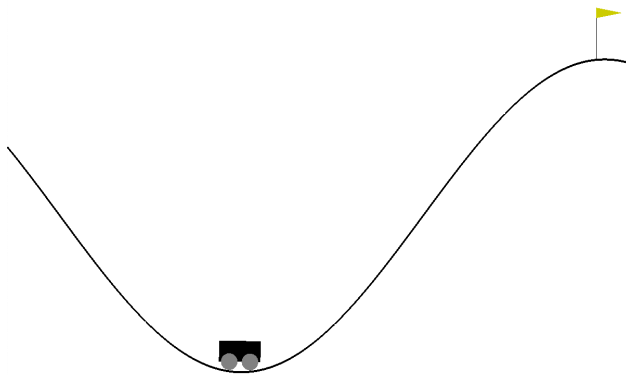


Figure 3: An example Mountain Car environment where the goal is to get to the top right flag.

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.

### 3.3 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  $\alpha$  and discount factor be  $\gamma$ . 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 (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  $\epsilon$ . 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.

### 3.4 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.

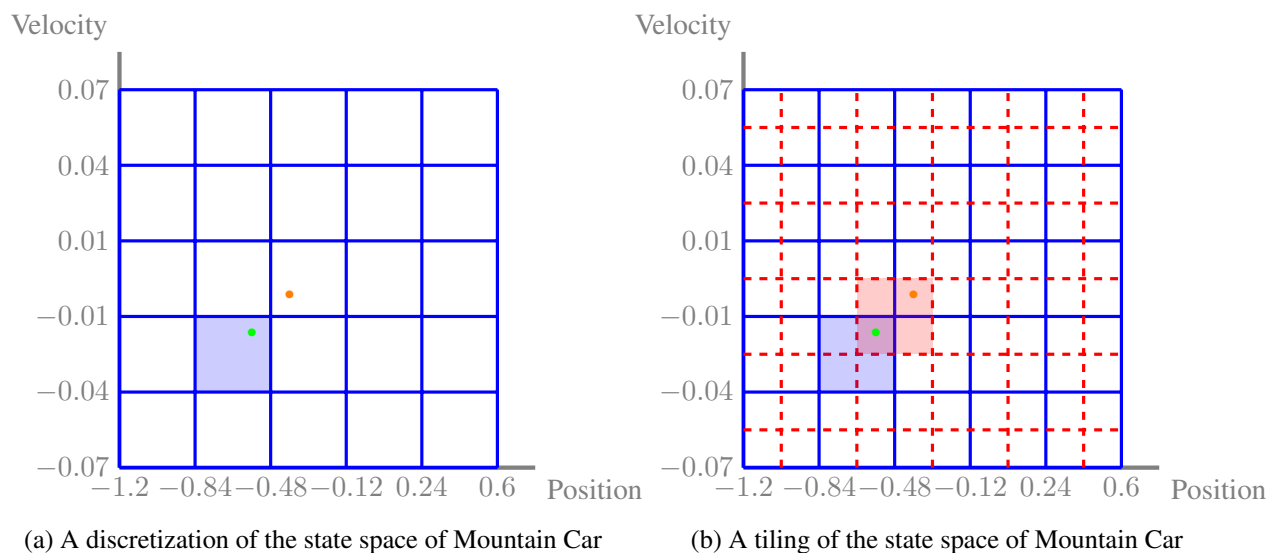


Figure 4: State representations for the states of Mountain Car

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 4a. 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 4b. 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.

### 3.5 Replay Buffer

As detailed in Section 3.3, 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.

### 3.6 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  $[\text{position}, \text{velocity}]^T$ .

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 4b would become  $[0, 0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots, 0]^T$ , 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 3.7 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, 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 `(state, reward, done)` which is the next state, the reward observed, and a boolean indicating if you reached the goal or not, ending the episode. The `state` will be either a raw or tile representation, as defined above, depending on how you initialized Mountain Car. If you observe `done = True` then you should `reset` the environment and end the episode. Failure to do so will result in undefined behavior.
- `render()`: Visualize the environment (not graded). Requires the installation of `pyglet`<sup>1</sup>. 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<sup>2</sup> 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"`.

<sup>1</sup>You can install it by typing `pip install pyglet` in your shell.

<sup>2</sup>You don't need to know how decorators work for this class, but you can read more about them [here](#) if you're interested.

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. `<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  $\epsilon$  for the epsilon-greedy strategy.
8. `<gamma>`: the discount factor  $\gamma$ .
9. `<learning_rate>`: the learning rate  $\alpha$  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
```

### 3.7 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
```

### 3.8 Gradescope Submission

You should only submit your `q_learning.py` to Gradescope. Please do not use any other file name for your implementation. This will cause problems for the autograder to correctly detect and run your code.

**Note:** For this assignment, you may make up to **10** submissions to Gradescope before the deadline, but only your last submission will be graded.