2 Programming [68 Points]

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

The program you write will be automatically graded using the Gradescope system. You may write your program in **Python, Java, or C++**. However, you should use the same language for all parts below.

2.1 Specification of Mountain Car

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

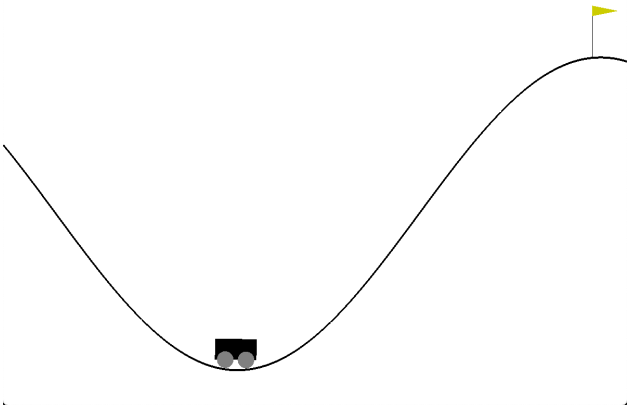


Figure 4: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

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

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

2.2 Q-learning With Linear Approximations

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

Let the learning rate be α and discount factor be γ. Recall that we have the information after one interaction

with the environment, (s, a, r, s'). The tabular update rule based on this information is:

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

Instead, for the function approximation setting we use the following update rule derived from the Function Approximation Section[6]:

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

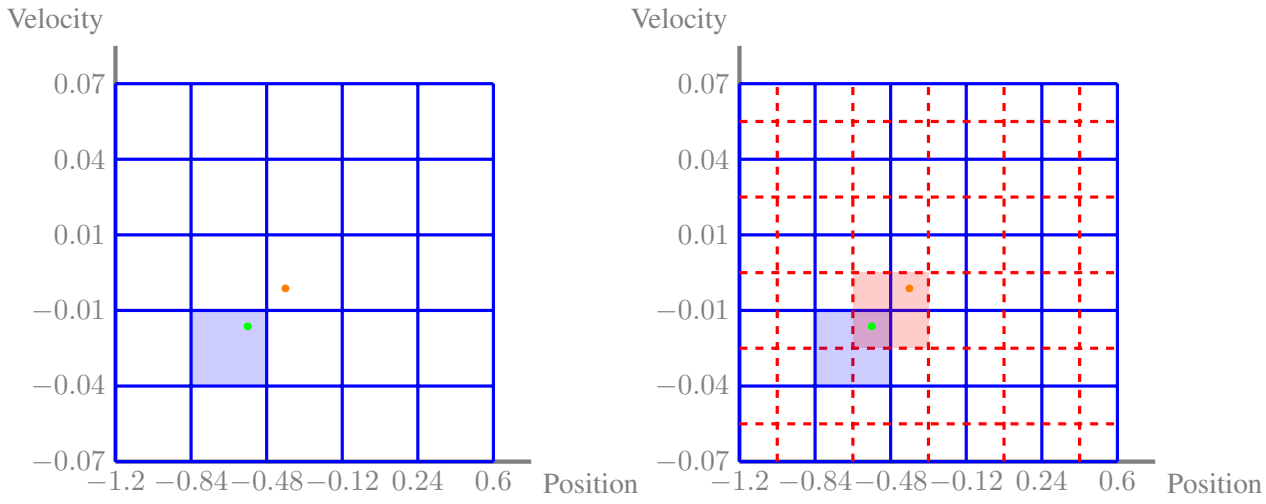
Where:

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

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

2.3 Feature Engineering

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



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

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

Figure 5: State representations for the states of Mountain Car

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

For the Mountain Car environment, we know that `position` and `velocity` are both bounded. What we can do is draw a grid over the possible `position-velocity` combinations as seen in Figure 5a. We then enumerate the grid from bottom left to top right, row by row. Then we map all states that fall into a grid square with the corresponding one-hot encoding of the grid number. For efficiency reasons we will just use the index that is non-zero. For example the green point would be mapped to $\{6\}$ and the orange point to $\{12\}$. This is called a *discretization* of the state space.

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

2.4 Implementation Details

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

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

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

  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.
6. `<epsilon>`: the value ϵ for the epsilon-greedy strategy
7. `<gamma>`: the discount factor γ.
8. `<learning_rate>`: the learning rate α of the Q-learning algorithm

Example command for python users:

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

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

`<weight_out>`

```
-7.6610506220312296
1.3440159024460183
1.344872959883069
1.340055578403996
```

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

```
-0.0007770480987990149
0.0011306483117300896
0.0017559989206646666
```

<returns_out>

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

2.5 Debugging Tips

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

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

Example command for python users:

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

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

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

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

Example command for python users:

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

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

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