

HOMework 6

NEURAL NETWORKS AND REINFORCEMENT LEARNING¹

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (SPRING 2021)

<http://mlcourse.org>

OUT: Mar. 30, 2021

DUE: Apr. 7, 2021 11:59 PM

TAs: Clay Yoo, Everett Knag, Varun Natu

Summary In this assignment, you will implement a reinforcement learning algorithm for solving the classic mountain-car environment. As a warmup, the first section is as a review of neural network and regularisation concepts. Later sections will then lead you through an on-paper example of how non-deterministic value iteration and Q-learning work. Then, in Section 2, you will implement Q-learning with function approximation to solve the mountain car environment.

START HERE: Instructions

- **Collaboration Policy:** Please read the collaboration policy here: <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 30 Gradescope programming submissions. 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.

¹Compiled on Wednesday 7th April, 2021 at 23:08

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 your 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>

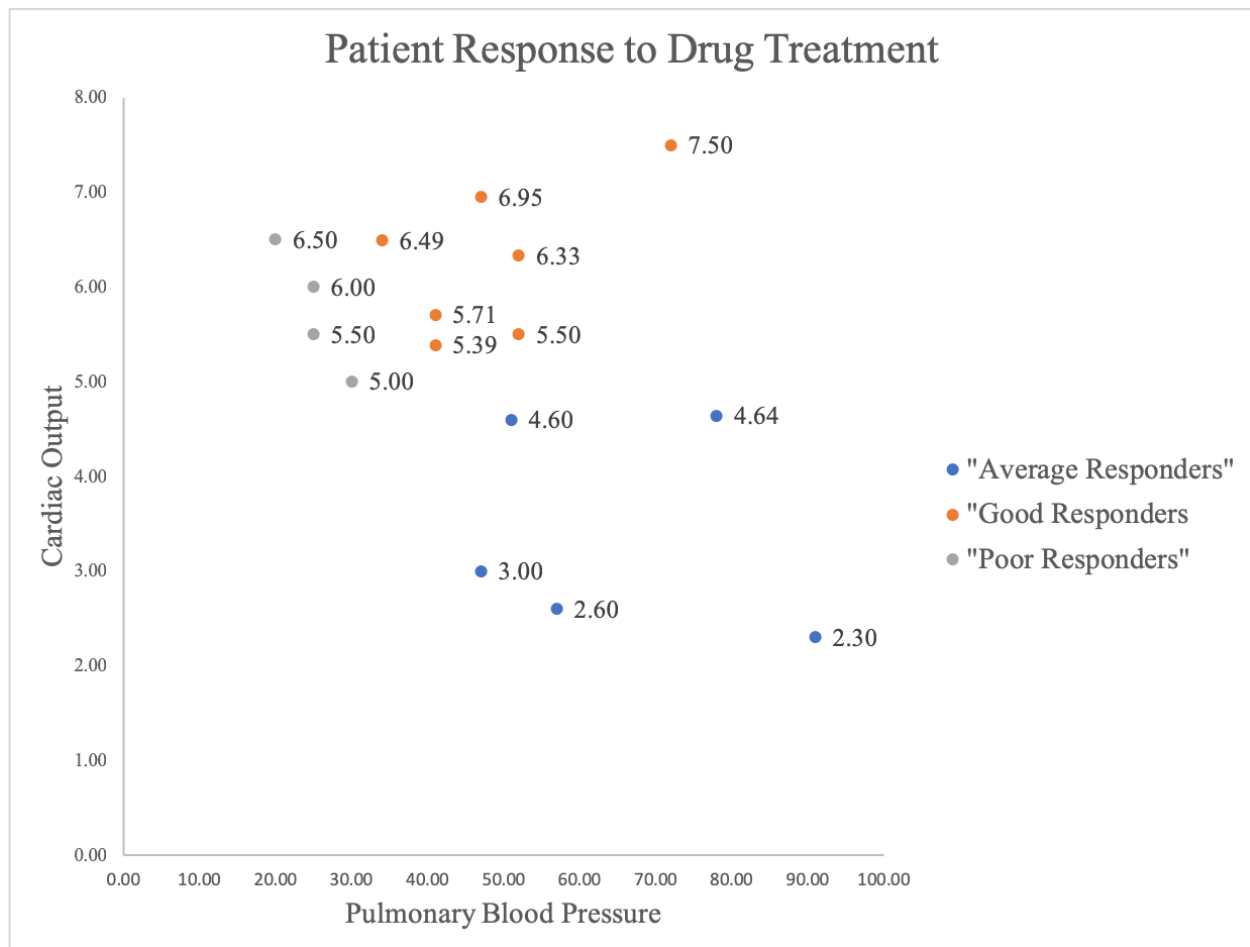
^b<https://deeplearning4j.org/docs/latest/nd4j-overview>

^c<http://eigen.tuxfamily.org/>

1 Written Questions [51 Points]

1.1 Neural Networks, Logistic Regression, Regularization Revisited

Imagine you work for a pharmaceutical company that is trying to predict whether certain patients will respond well to a new drug. Specifically, these patients have high blood pressure in their lungs, a condition known as pulmonary hypertension. Doctors recommend that the best predictors of a treatment effect is the patient's heart function (measured by cardiac output) and the blood pressure in their lungs (pulmonary blood pressure). You plot the data and visualize the following:



- (2 points) Draw on the graph above the decision boundaries of a trained neural network that minimizes the training error when classifying Good Responders vs all others (Poor or Average). Assume the neural network has two hidden units and one hidden layer. What is the smallest training error you can achieve?

Fill in the blank (write answer as a fraction):

Your Answer

0

2. (2 points) Using your decision boundaries above, assuming a logistic activation function, which point has the highest probability of being a Good Responder? Provide the point number as shown on the graph (points are labeled by their y value).

Fill in the blank:

Your Answer

7.5

3. (1 point) True or False: Increasing the number of hidden units of a neural network will always guarantee a lower training error.
- ☐ True
- ☒ **False**
4. (1 point) Convolutional neural networks often consist of convolutional layers, max-pooling layers, and fully-connected layers. Select all the layer types below that have parameters (i.e. weights) which can be learned by gradient descent / backpropagation. **Select all that apply:**
- ☒ **convolutional layer**
- ☐ max-pooling layer
- ☒ **fully-connected layer**
5. (2 points) Consider the black-and-white 5 pixel by 5 pixel image shown below. Black pixels are represented by the value 0 and white pixels by 1.

1	1	0	0	0
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	0	1	1

Next consider the 3 by 3 convolution with weights shown below.

-1	0	1
-1	0	1
-1	0	1

Suppose we apply the above convolution (as one would in a CNN convolutional layer) to the above image to produce a new output image. Assume that we do not permit any padding to be used and the stride of the convolution is 1. What is the value of the pixel in the upper-left corner of the output image?

(Important Note: Convolution is sometimes defined differently in machine learning than in other fields, such as signal processing. So be sure to follow the method of convolution shown in the lectures on CNNs.) **Fill in the blank:**

Your Answer
0

6. (2 points) For the same output image produced in the previous question, what is the value of the pixel in the upper-right corner of the output image?

Fill in the blank:

Your Answer
-2

7. (1 point) True or False: Long Short Term Memory (LSTM) networks partially address the vanishing gradient problem by incorporating input, output, and forget gates.

☒ **True**

☐ False

8. (1 point) True or False: In a recurrent neural network (RNN), the weight vector passed to a hidden unit at the same hidden layer level from a prior unit will have different values.

☐ True

☒ **False**

9. (1 point) True or False: Recurrent neural networks (RNNs) can accept sequence data as input, but can only output a single classification decision for that input sequence.

☐ True

☒ **False**

10. (2 points) Regularization.

Which of the following are true about regularization? **Select all that apply:**

☒ **One of the goals of regularization is combating overfitting.**

☐ A model with regularization fits the training data better than a model without regularization

☒ **The L-0 norm (number of non-zero parameters) is rarely used in practice in part because it is non-differentiable.**

☒ **One way to understand regularization is that it attempts to follow Occam's razor and make the learning algorithm prefer "simpler" solutions.**

11. (2 points) Regularization in Linear Regression.

When performing linear regression, which of the following options will decrease mean-squared training error? **Select all that apply:**

☒ **Adding higher-order functions of the features as separate features**

- ☐ Increasing the regularization weight
- ☐ For the same weight on the regularizer, using an L1 regularizer instead of an L2
- ☒ **For the same weight on the regularizer, using an L1 regularizer instead of an L0**
- ☐ None of the above

1.2 Non-Deterministic Value Iteration

In this question we will explore value iteration with a non-deterministic transition function. Suppose the agent is in a grid world. In any state, there is the following **non-deterministic** transition function: with probability 80% the agent transitions to the intended state. With probability 10% the agent slips left of the intended direction. With probability 10% the agent slips right of the intended direction. If the agent hits the edge of the board, it remains in the same state.

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

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

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

Table 1: Depiction of Grid World

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

$$v_0(s) = 0$$

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

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

Answer

4⁶

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

Select all that apply:

- ☐ Up
- ☒ Down
- ☐ Left
- ☐ Right

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

Answer
0

4. (1 point) What is the value of state C in round 1 of synchronous value iteration? Round your answer to three decimal places.

Answer
0.1

5. (1 point) What is the value of state C in round 3 of synchronous value iteration? Round your answer to three decimal places.

Answer
0.245

6. (1 point) What is the final value of state A once value iteration converges.

Answer
1

7. (2 points) What is the optimal action for each state.

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

Table 2: Place the corresponding action in each blank cell

1.3 Q-Learning Multiple Choice

1. (1 point) True or False: Given the value function $V^*(s)$, we can derive the optimal action-value function $Q^*(s, a)$

Select one choice:

☐ True

☒ False

2. (1 point) True or False: Q-Learning cannot learn from interacting with the environment because it updates its estimates based on its own previously learned estimates

Select one choice:

☐ True

☒ False

3. (2 points) Which of the following is true about Q-Learning?

Select all that apply:

☐ Q-Learning creates an explicit model of the environment and state-transition function to learn state-value estimates

☐ Q-Learning must wait for an episode to end before it can update its estimates.

☐ Q-Learning can update its estimates after the agent takes a single action

☒ Q-Learning can learn without being given an explicit reward function or state-transition function.

☐ None of the above.

4. (2 points) Which of the following is true about the convergence properties of Q-Learning?

Select all that apply:

☐ Q-Learning will converge to different estimates of the action-value function $Q(s, a)$ for ϵ -greedy policies with different non-zero ϵ values.

☐ Q-Learning necessarily converges to the optimal action-value function $Q^*(s, a)$ for all policies

☒ Q-Learning converges to the optimal action-value function $Q^*(s, a)$ under the assumption that all state-action pairs are visited infinitely often by the policy.

☐ Q-Learning guarantees convergence only for deterministic environments.

☐ All of the above.

5. (2 points) Which of the following is the correct expression for the action-value function Q upon taking action a_t in state s_t in terms of the discount factor γ , Reward function $R(s, a)$, State transition function $T(s'|s, a)$ and the value function $V(s)$?

Select one choice:

☒ $Q(s_t, a_t) = R(s_t, a_t) + \gamma * [\sum_{s_{t+1}} T(s_{t+1}|s_t, a_t) * V(s_{t+1})]$

☐ $Q(s_t, a_t) = R(s_t, a_t) + \gamma * [\frac{1}{|S|} \sum_{s_{t+1}} Q(s_{t+1}, a_t)]$, where $|S|$ is the total number of states

☐ $Q(s_t, a_t) = R(s_t, a_t) + \gamma * [\frac{1}{|S|} \sum_{s_{t+1}} V(s_{t+1})]$, where $|S|$ is the total number of states

☐ The given quantities and are not enough to express Q

1.4 Function Approximation

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

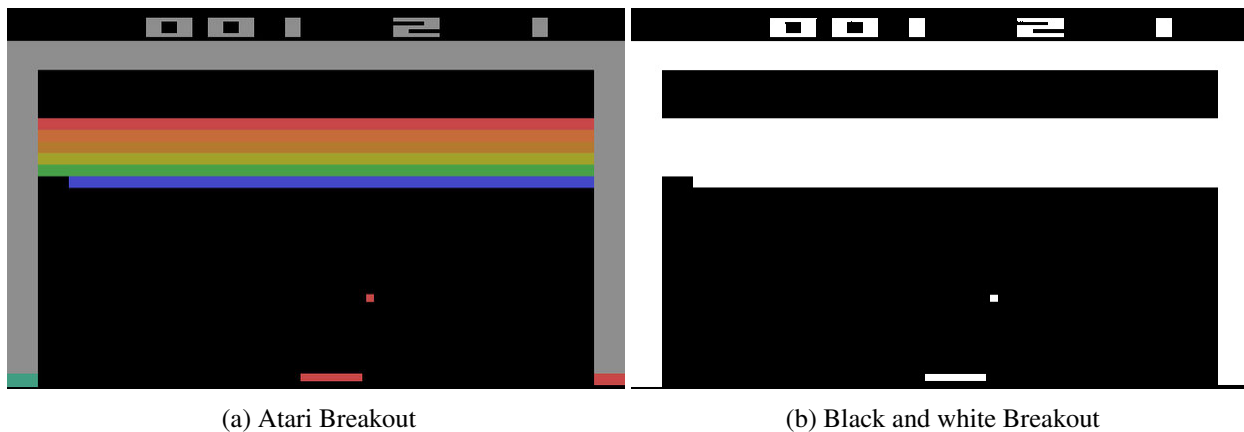


Figure 1: Atari Breakout. **1a** is what Breakout looks like. We have the paddle in the bottom of the screen aiming to hit the ball in order to break the tiles at the top of the screen. **1b** is our transformation of Atari Breakout into black and white pixels for the purpose of some of the following problems.

- (1 point) Suppose we are dealing with the black and white version of Breakout² as in Figure **1b**. Furthermore, suppose we are representing the state of the game as just a vector of pixel values without considering if a certain pixel is always black or white. Since we are dealing with the black and white version of the game, these pixel values can either be 0 or 1.

What is the size of the state space?

Answer
$2^{(160 \times 192)}$

- (1 point) In the same setting as the previous part, suppose we wish to apply Q-learning to this problem. What is the size of the Q-value table we will need?

Answer
$2^{(160 \times 192)} \times 3$

²Play 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).

What is the size of the state space and Q-value table we will need?

Answer

$8^{(160 \times 192)}$, and $8^{(160 \times 192)} \times 3$

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:

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

Because we want to update for each example stochastically³, we get the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha (q(s, a; \mathbf{w}) - q_\pi(s, a)) \nabla_{\mathbf{w}} q(s, a; \mathbf{w}) \quad (2)$$

However, more often than not⁴ 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 bootstrap⁵ an estimate of the action value under a greedy policy using the function approximator itself. That is to say

$$q_\pi(s, a) \approx r + \gamma \max_{a'} q(s', a'; \mathbf{w}) \quad (3)$$

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:

³This isn't really stochastic, you'll be asked in a bit why.

⁴Always in real life.

⁵Metaphorically, the agent is pulling itself up by its own bootstraps.

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

4. (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:

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

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 5 encodes a table lookup)

Answer

In the linear approximation, we approximate the q-value of a state as a linear combination of some feature vector and a vector of weights, i.e., $q(\mathbf{s}, a; \mathbf{w}) = \mathbf{s}^T \mathbf{w}_a$. In the discrete case, we can construct a feature vector \mathbf{s} as a list of dummy variables, each as an indicator corresponding to a discrete state. Then the feature vector \mathbf{s} will have zeros for all the entries except for one, which will be 1. Each $\omega_i \in \mathbf{w}$ will then correspond to a single state s_i , and q value will just return that weight as the value of s_i . Thus a tabular Q-learning is exactly the same as the Q-learning with a linear function approximation with the design of \mathbf{s} described above. Thus it is a special case.

5. (3 points) Stochastic Gradient Descent works because we can assume that the samples we receive are independent and identically distributed. Is that the case here? If not, why and what are some ways you think you could combat this issue?

Answer

No. Since SGD works if the samples are IID, and in the case of RL, the successive samples can be correlated. Thus using the SGD that is based on one data sample may not provide with accurate solutions. However, we can still use stochastic optimization method to update the parameters. We can re-sample the data set by sampling randomly and repeatedly using such as the ϵ -greedy policy. Then we can use the SGD method on the randomly selected sub-data set and then update the parameters.

1.5 Empirical Questions

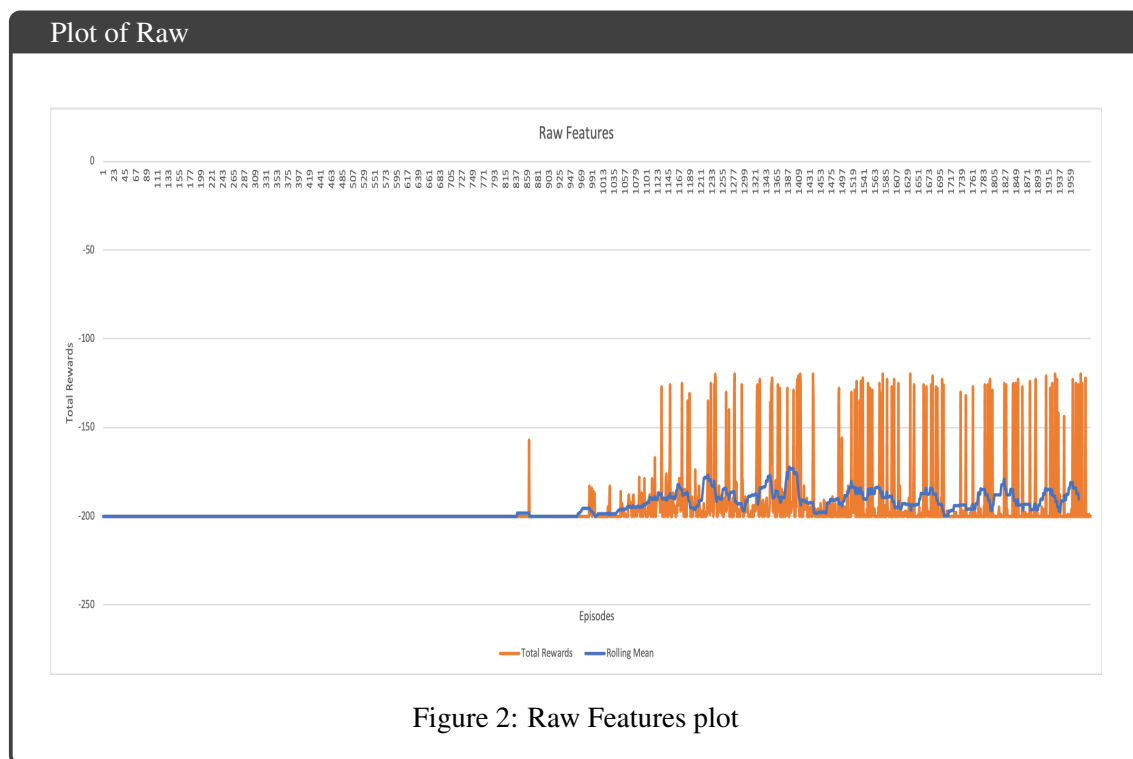
The following parts should be completed after you work through the programming portion of this assignment (Section 2).

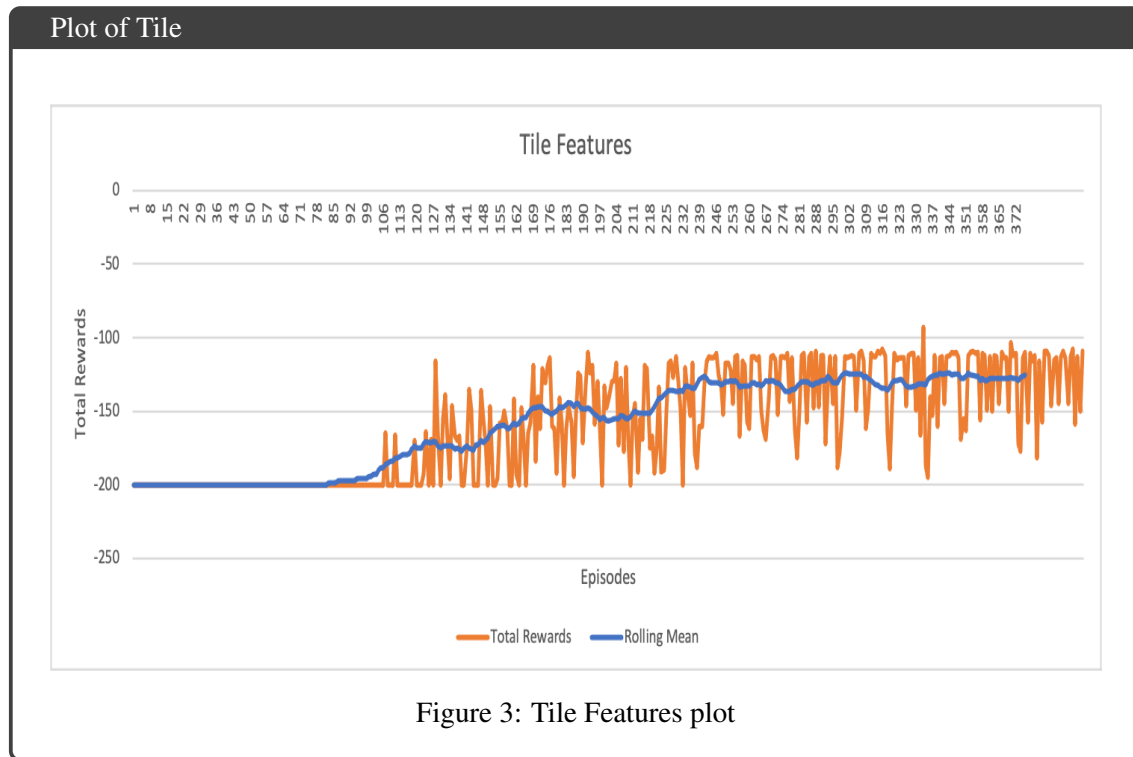
1. (4 points) Run Q-learning on the mountain car environment using both tile and raw features.

For the raw features: run for 2000 episodes with max iterations of 200, ϵ set to 0.05, γ set to 0.999, and a learning rate of 0.001.

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

For each set of features, plot the return (sum of all rewards in an episode) per episode on a line graph. On the same graph, also plot the rolling mean over a 25 episode window. Comment on the difference between the plots.





Comment

As we can see from the above two graphs, Tile Features can reach the goal more quickly and frequently. Such results are easy to understand since Tile Features are more complicated and thus by using such features can result in models with high accuracy. Especially note that for the Tile Features, the learning rate is even lower than the Raw Features, but still the learning is faster and more efficient.

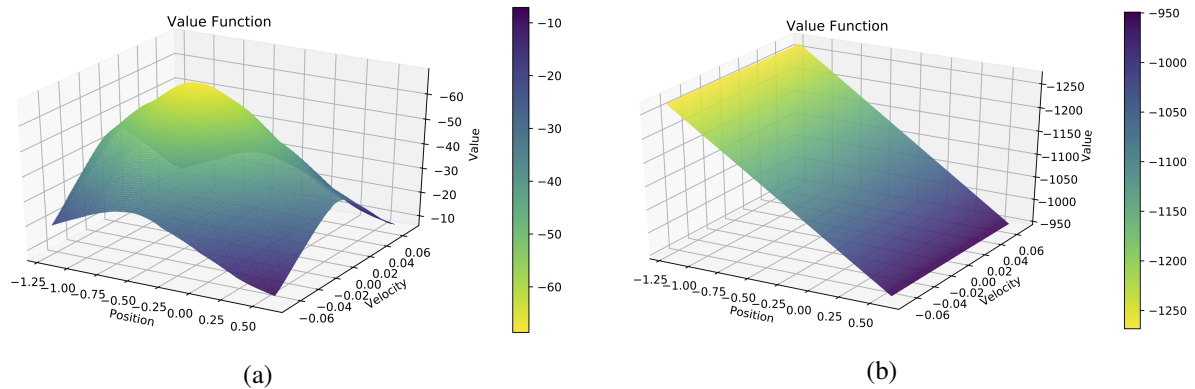


Figure 4: Estimated optimal value function visualizations for both types of features

2. (2 points) For both raw and tile features, we have run Q-learning with some good⁶ parameters and created visualizations of the value functions after many episodes. For each plot in Figure 4, 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

Based on the graphs of the two value function visualizations, I will guess that (a) is for the Tile Feature, and (b) is for the Raw Feature. This is because since for the Raw feature, the feature only has dimension of two, and thus the dimension for the parameters that we learnt is also two, thus we must have a flat surface as in (b). However, since for the Tile Feature, we have 192 features out of 2048 potential features, thus we have much more parameters to learn, and thus the value function can be a non-linear function such as in (a).

3. (2 points) We see that Figure 4b 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

Yes. It can be a nonlinear value function. Since the value function is the maximum one among all the possible actions, depending on the parameter values of position and velocity, we may have maximum function for several flat plains, which means several pairs of 2-dimensional parameters. Thus the value function can potentially have the shape such that it is a combination of several flat plains, and thus a pyramid like non-linear function.

⁶For some sense of good.

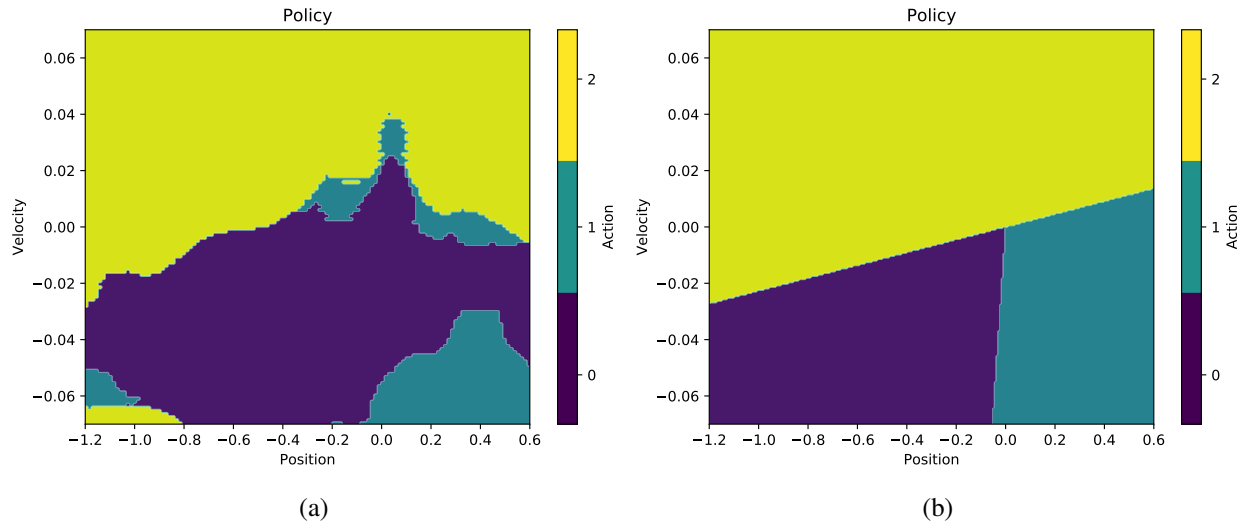


Figure 5: Estimated optimal policy visualizations for both types of features

4. (2 points) In a similar fashion to the previous question we have created visualizations of the potential policies learned. For each plot in Figure 5 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

Similar to question 2, my guess is that (a) is for the Raw Feature, and (b) is for the Tile Feature. This is because since the raw feature has only two dimensions, it cannot provide accurate result for the policy function with high probability. Since Raw Features are inaccurate, frequently the data points will be incorrectly classified. Thus we can see that the boundary is not flat, and has some overlapping. For example the blue range around position of -1 shouldn't happen. The true optimal policy function should look like something in (b), and since Tile Features can achieve higher accuracy, (b) should be the Tile Feature. Right Bottom is action 1 since it is near the goal, and with the position and velocity, even though we don't push it will reach the goal.

1.6 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these parts 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. No.
2. No.
3. No.

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 6. 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.

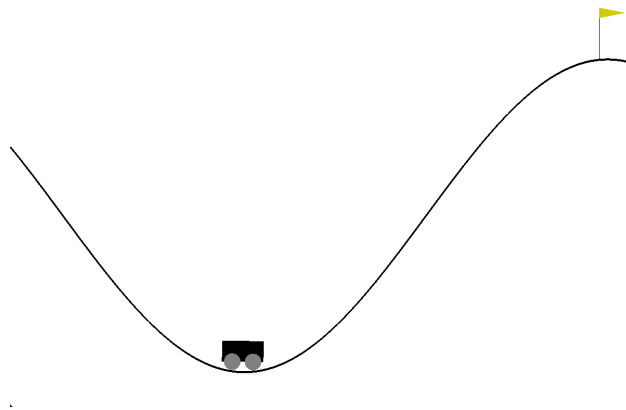


Figure 6: What the Mountain Car environment looks like. The car starts at some point in the valley. The goal is to get to the top right flag.

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 Section⁷:

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

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.

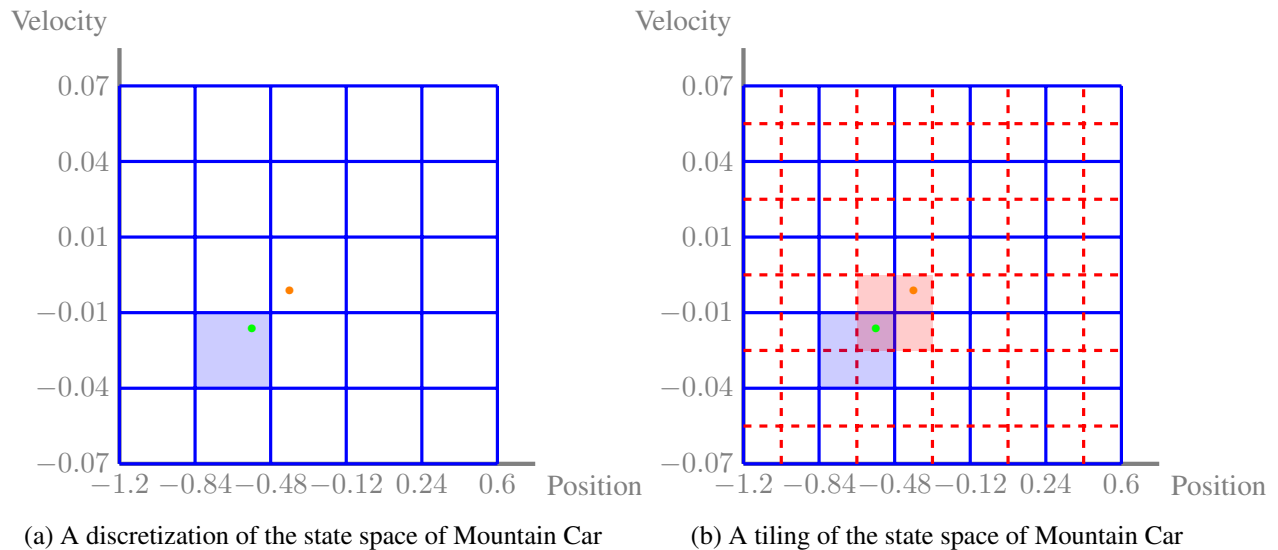


Figure 7: State representations for the states of Mountain Car

⁷Note 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 7a. 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 7b. 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 analogous 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: $\{0 \rightarrow \text{position}, 1 \rightarrow \text{velocity}\}$.

In “tile” mode you are given indices of the tiles which are active in a sparse format: $\{T_1 \rightarrow 1, T_2 \rightarrow 1, \dots, T_n \rightarrow 1\}$ where T_i is the tile index for the i th tiling. All other tile indices are assumed to map to 0. For example the state representation of the example in Figure 7b would become $\{6 \rightarrow 1, 39 \rightarrow 1\}$.

The dimension of the state space of the “raw” mode is 2. The dimension of the state space of the “tile” mode is 2048. These values can be accessed from the environment through the `state_space` property, and similarly for other languages.

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

- `reset()`: Reset the environment to starting conditions.
- `step(action)`: Take a step in the environment with the given action. `action` must be either 0, 1 or 2. This will return a tuple of `(state, reward, done)` which is the next state, the reward observed, and a boolean indicating if you reached the goal or not, ending the episode. The `state` will be either a raw or tile representation, as defined above, depending on how you initialized Mountain Car. If you observe `done = True` then you should `reset` the environment and end the episode. Failure to do so will result in undefined behavior.
- **[Python Only]** `render(self)`: Optionally render the environment. It is computationally intensive to render graphics, so only render a full episode once every 100 or 1000 episodes. Requires the installation of `pyglet`. This will be a no-op in Gradescope.

You should now implement your Q-learning algorithm with linear approximations as

`q_learning.{py|java|cpp}`. The program will assume access to a given environment file(s) which contains the Mountain Car environment which we have given you. **Initialize the parameters of the linear**

model with all 0 (and don't forget to include a bias!) and use the epsilon-greedy strategy for action selection.

Your program should write a output file containing the total rewards (the returns) for every episode after running Q-learning algorithm. There should be one return per line.

Your program should also write an output file containing the weights of the linear model. The first line should be the value of the bias. Then the following $|\mathcal{S}| \times |\mathcal{A}|$ lines should be the values of weights, outputted in row major order⁸, 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
```

⁸https://en.wikipedia.org/wiki/Row-_and_column-major_order

```
-0.0007770480987990149
0.0011306483117300896
0.0017559989206646666
```

<returns_out>

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

2.5 Debugging Tips

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

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

Example command for python users:

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

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

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

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

Example command for python users:

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

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

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

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

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

2.6 Gradescope Submission

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

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