Homework 6: Inference in Graphical Models, MDPs

Introduction

In this assignment, you will practice inference in graphical models as well as MDPs/RL. For readings, we recommend Sutton and Barto 2018, Reinforcement Learning: An Introduction, CS181 2017 Lecture Notes, and Section 10 and 11 Notes.

Please type your solutions after the corresponding problems using this LaTeX template, and start each problem on a new page.

Please submit the **writeup PDF to the Gradescope assignment 'HW6'**. Remember to assign pages for each question.

Please submit your **LaTeX file and code files to the Gradescope assignment 'HW6 - Supplemental'**.

You can use a **maximum of 2 late days** on this assignment. Late days will be counted based on the latest of your submissions.

Problem 1 (Explaining Away + Variable Elimination 15 pts)

In this problem, you will carefully work out a basic example with the "explaining away" effect. There are many derivations of this problem available in textbooks. We emphasize that while you may refer to textbooks and other online resources for understanding how to do the computation, you should do the computation below from scratch, by hand.

We have three binary variables: rain R, wet grass G, and sprinkler S. We assume the following factorization of the joint distribution:

$$\Pr(R, S, G) = \Pr(R)\Pr(S)\Pr(G \mid R, S).$$

The conditional probability tables look like the following:

$$\begin{aligned}
\Pr(R = 1) &= 0.25 \\
\Pr(S = 1) &= 0.5 \\
\Pr(G = 1 | R = 0, S = 0) &= 0 \\
\Pr(G = 1 | R = 1, S = 0) &= 0.75 \\
\Pr(G = 1 | R = 0, S = 1) &= 0.75 \\
\Pr(G = 1 | R = 1, S = 1) &= 1
\end{aligned}$$

1. Draw the graphical model corresponding to the factorization. Are R and S independent? [Feel free to use facts you have learned about studying independence in graphical models.]
2. You notice it is raining and check on the sprinkler without checking the grass. What is the probability that it is on?
3. You notice that the grass is wet and go to check on the sprinkler (without checking if it is raining). What is the probability that it is on?
4. You notice that it is raining and the grass is wet. You go check on the sprinkler. What is the probability that it is on?
5. What is the "explaining away" effect that is shown above?

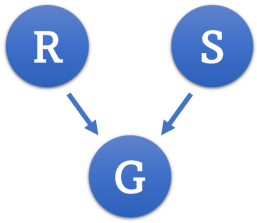
Consider if we introduce a new binary variable, cloudy C, to the the original three binary variables such that the factorization of the joint distribution is now:

$$\Pr(C, R, S, G) = \Pr(C)\Pr(R|C)\Pr(S|C)\Pr(G \mid R, S).$$

6. For the marginal distribution $\Pr(R)$, write down the variable elimination expression with the elimination ordering S, G, C (where S is eliminated first, then G, then C).
7. For the marginal distribution $\Pr(R)$, write down the variable elimination expression with the elimination ordering C, G, S.
8. Give the complexities for each ordering. Which elimination ordering takes less computation?

Solution:

1. The graphical model is:



R and S are independent if G is unknown since the path from R to S is blocked.

2. G is unknown, so S and R are independent, and $P(S = 1|R = 1) = P(S = 1) = 0.5$.
3. By the law of total probability,

$$P(G = 1) = \sum_{i \in \{0,1\}} \sum_{j \in \{0,1\}} P(G = 1|R = i, S = j)P(R = i, S = j).$$

Plugging in and simplifying yields

$$P(G = 1) = 0 \cdot 0.75 \cdot 0.5 + 0.75 \cdot 0.25 \cdot 0.5 + 0.75 \cdot 0.75 \cdot 0.5 + 1 \cdot 0.25 \cdot 0.5 = 0.5.$$

Similarly,

$$P(S = 1, G = 1) = P(G = 1|R = 0, S = 1)P(R = 0, S = 1) + P(G = 1|R = 1, S = 1)P(R = 1, S = 1).$$

Plugging in and simplifying yields

$$P(S = 1, G = 1) = (0.75 \cdot 0.75 + 1 \cdot 0.25) \cdot 0.5 = 0.40625.$$

Thus $P(S = 1|G = 1) = \frac{P(S=1,G=1)}{P(G=1)} = \frac{0.40625}{0.5} = 0.8125$.

4. By the law of total probability,

$$P(R = 1, G = 1) = P(G = 1|R = 1, S = 0)P(R = 1, S = 0) + P(G = 1|R = 1, S = 1)P(R = 1, S = 1).$$

Plugging in and simplifying yields

$$P(R = 1, G = 1) = 0.75 \cdot 0.25 \cdot 0.5 + 1 \cdot 0.25 \cdot 0.5 = 0.21875.$$

Thus

$$P(S = 1|R = 1, G = 1) = \frac{P(S = 1, R = 1, G = 1)}{P(R = 1, G = 1)} = \frac{0.125}{0.21875} = \frac{4}{7} \approx 0.571.$$

5. The probability that the sprinkler is on given that the grass is wet and that it rained is lower that the probability that the sprinkler is on given only that the grass is wet. Knowing that it also rained decreases the chance of the sprinkler being on because the rain explains the grass being wet.
6. Summing out and applying the elimination ordering yields

$$P(R) = \sum_{C=0}^{1} P(C)P(R|C) \sum_{G=0}^{1} \sum_{S=0}^{1} P(S|C)P(G|R, S).$$

Note that $\sum_{G=0}^{1} \sum_{S=0}^{1} P(S|C)P(G|R, S) = 1$ since we're just summing $P(G|R, C)$ over all values of G. Thus we just need to compute $\sum_{C=0}^{1} P(C)P(R|C)$. $P(C)$ has $O(k)$ terms while $P(R|C)$ has $O(k^2)$ terms.

7. Summing out and applying the elimination ordering yields

$$P(R) = \sum_{S=0}^{1} \sum_{G=0}^{1} P(G|R,S) \sum_{C=0}^{1} P(C)P(R|C)P(S|C).$$

Note that $\sum_{C=0}^{1} P(C)P(R|C)P(S|C)$ evaluates to $P(R,S)$, and we can pull it out out the sum over G. Additionally, $\sum_{G=0}^{1} P(G|R,S) = 1$ since we're summing over all values of G. Thus we just need to evaluate

$$P(R) = \sum_{S=0}^{1} \sum_{C=0}^{1} P(C)P(R|C)P(S|C).$$

$P(C)$ has $O(k)$ terms, and $P(R|C)$ and $P(S|C)$ each have $O(k^2)$ terms.

8. The complexity of part 6 is $O(k^2)$ since we need to handle all values of R and C (one summation per value of R) while the complexity of part 7 is $O(k^3)$ since we need to handle all values of R, C, and S (two summations per value of R). Thus the elimination ordering S, G, C requires less computation.

Problem 2 (Policy and Value Iteration, 15 pts)

This question asks you to implement policy and value iteration in a simple environment called Gridworld. The "states" in Gridworld are represented by locations in a two-dimensional space. Here we show each state and its reward:

R=4	R=0	R=-10	R=0	R=20
R=0	R=0	R=-50	R=0	R=0
START R=0	R=0	R=-50	R=0	R=50
R=0	R=0	R=-20	R=0	R=0

The set of actions is {N, S, E, W}, which corresponds to moving north (up), south (down), east (right), and west (left) on the grid. Taking an action in Gridworld does not always succeed with probability 1; instead the agent has probability 0.1 of "slipping" into a state on either side, but not backwards. For example, if the agent tries to move right from START, it succeeds with probability 0.8, but the agent may end up moving up or down with probability 0.1 each. Also, the agent cannot move off the edge of the grid, so moving left from START will keep the agent in the same state with probability 0.8, but also may slip up or down with probability 0.1 each. Lastly, the agent has no chance of slipping off the grid - so moving up from START results in a 0.9 chance of success with a 0.1 chance of moving right.

Also, the agent does not receive the reward of a state immediately upon entry, but instead only after it takes an action at that state. For example, if the agent moves right four times (deterministically, with no chance of slipping) the rewards would be +0, +0, -50, +0, and the agent would reside in the +50 state. Regardless of what action the agent takes here, the next reward would be +50.

Your job is to implement the following three methods in file `T6_P2.ipynb`. Please use the provided helper functions `get_reward` and `get_transition_prob` to implement your solution.

Do not use any outside code. (You may still collaborate with others according to the standard collaboration policy in the syllabus.)

Embed all plots in your writeup.

Problem 2 (cont.)

Important: The state space is represented using integers, which range from 0 (the top left) to 19 (the bottom right). Therefore both the policy `pi` and the value function `V` are 1-dimensional arrays of length `num_states = 20`. Your policy and value iteration methods should only implement one update step of the iteration - they will be repeatedly called by the provided `learn_strategy` method to learn and display the optimal policy. You can change the number of iterations that your code is run and displayed by changing the `max_iter` and `print_every` parameters of the `learn_strategy` function calls at the end of the code.

Note that we are doing infinite-horizon planning to maximize the expected reward of the traveling agent. For parts 1-3, set discount factor $\gamma = 0.7$.

1a. Implement function `policy_evaluation`. Your solution should learn value function V, either using a closed-form expression or iteratively using convergence tolerance `theta = 0.0001` (i.e. if $V^{(t)}$ represents V on the t-th iteration of your policy evaluation procedure, then if $|V^{(t+1)}[s] - V^{(t)}[s]| \leq \theta$ for all s, then terminate and return $V^{(t+1)}$.)

1b. Implement function `update_policy_iteration` to update the policy `pi` given a value function `V` using **one step** of policy iteration.

1c. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 policy iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

1d. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

2a. Implement function `update_value_iteration`, which performs **one step** of value iteration to update `V`, `pi`.

2b. Set `max_iter = 4`, `print_every = 1` to show the learned value function and the associated policy for the first 4 value iterations. Do not modify the plotting code. Please fit all 4 plots onto one page of your writeup.

2c. Set `ct = 0.01` and increase `max_iter` such that the algorithm converges. Include a plot of the final learned value function and policy. How many iterations does it take to converge? Now try `ct = 0.001` and `ct = 0.0001`. How does this affect the number of iterations until convergence?

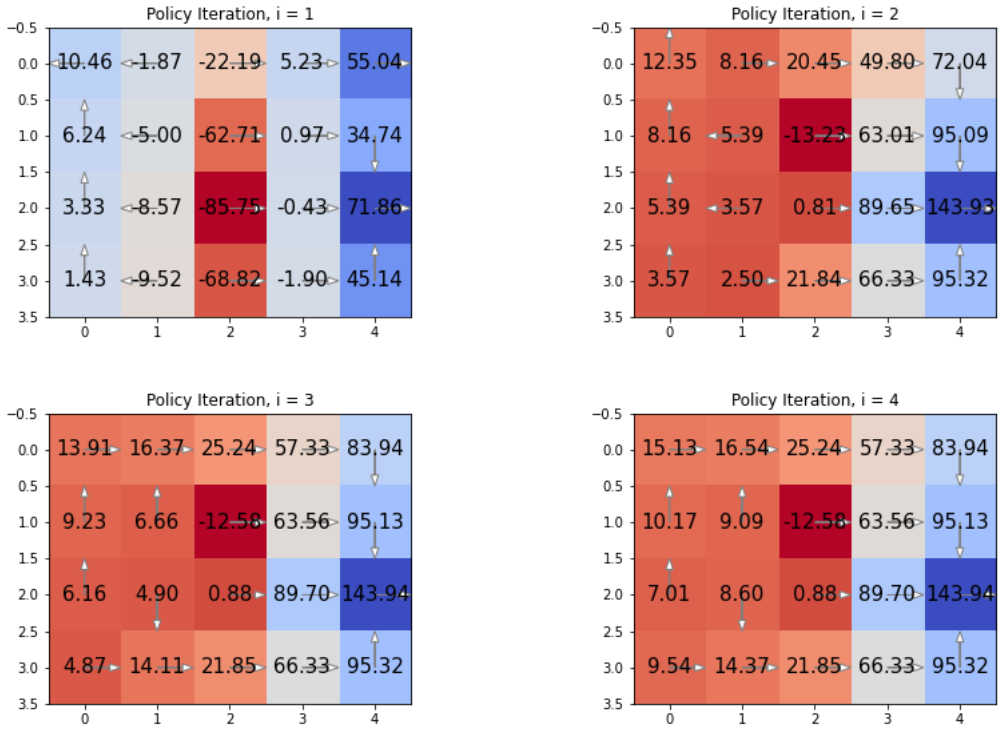
3 Compare and contrast the number of iterations, time per iteration, and overall runtime between policy iteration and value iteration. What do you notice?

4 Plot the learned policy with each of $\gamma \in (0.6, 0.7, 0.8, 0.9)$. Include all 4 plots in your writeup. Describe what you see and provide explanations for the differences in the observed policies. Also discuss the effect of gamma on the runtime for both policy and value iteration.

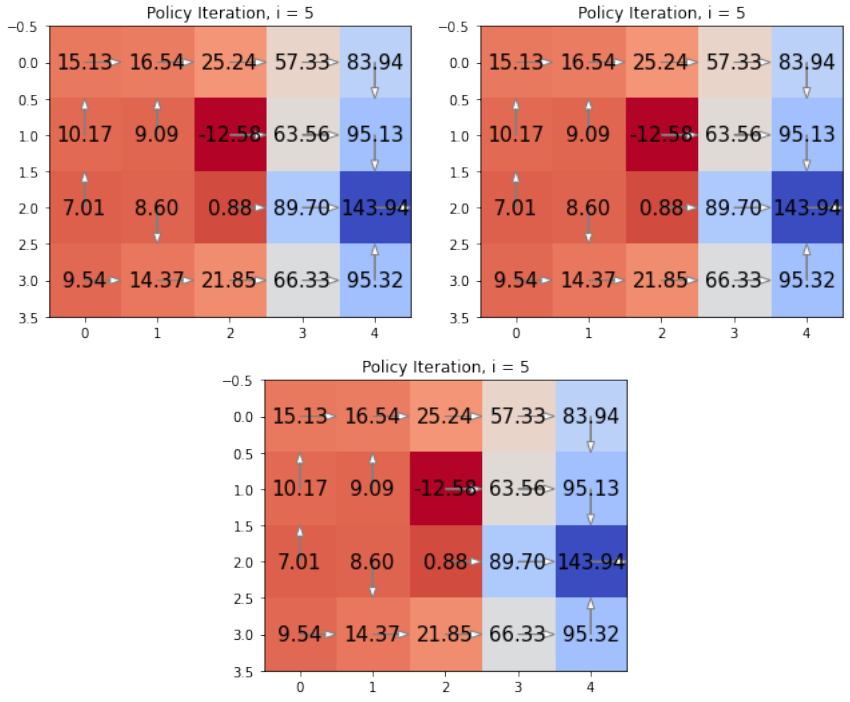
5 Now suppose that the game ends at any state with a positive reward, i.e. it immediately transitions you to a new state with zero reward that you cannot transition away from. What do you expect the optimal policy to look like, as a function of gamma? Numerical answers are not required, intuition is sufficient.

Solution:

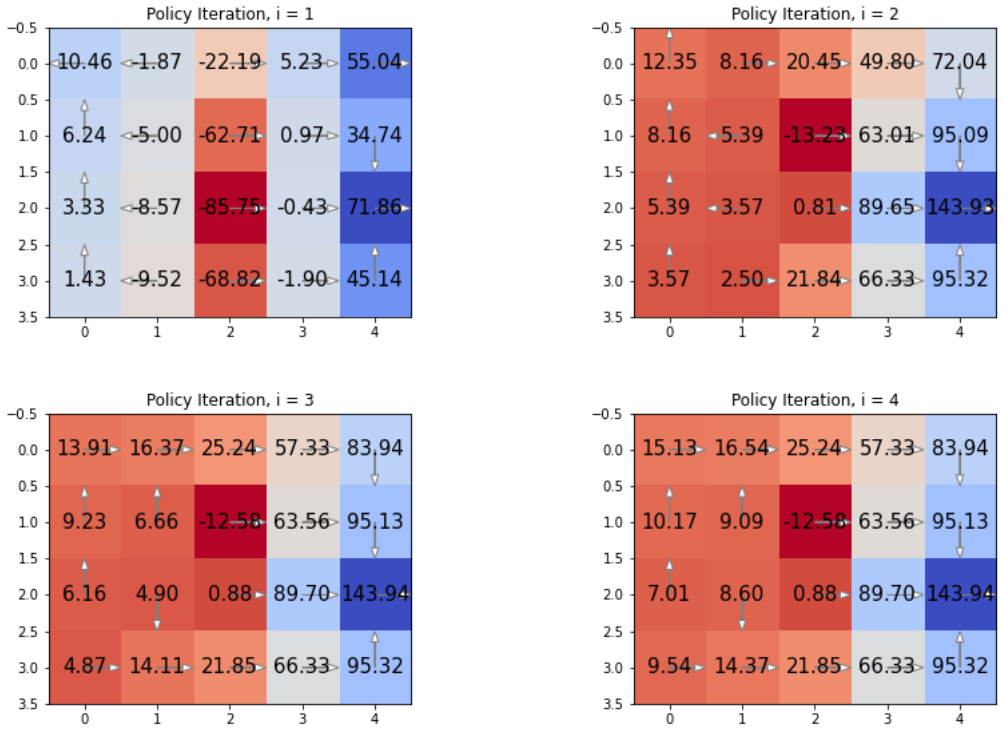
1c. The plots are:



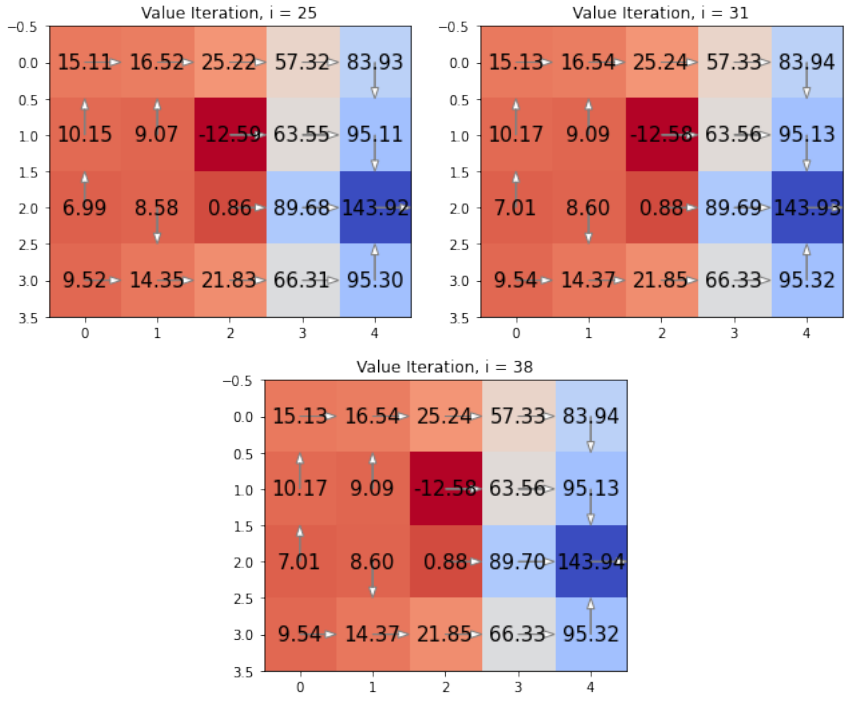
1d. The number of iterations needed is 5 for all 3 convergence thresholds. They all produce the same graph:



2b. The plots are:



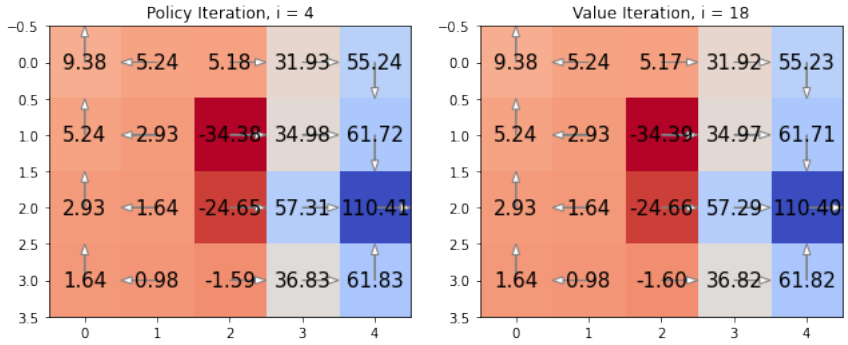
2c. The number of iterations needed for 0.01, 0.001, and 0.0001 are 25, 31, and 38, respectively. The graphs are shown in order:



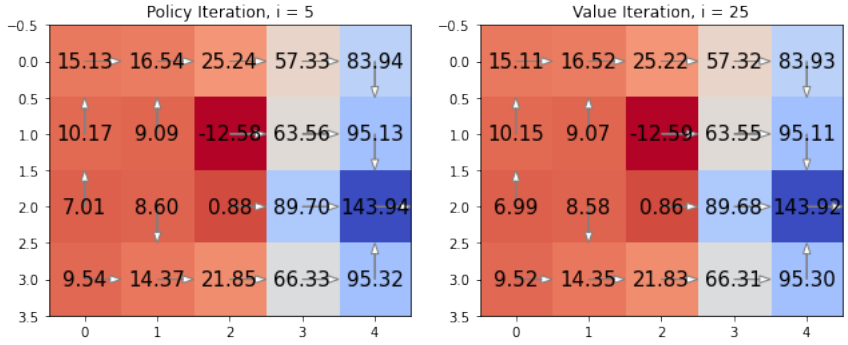
3. Note that I used the closed form solution for policy iteration, and I ran both with ct equal to 0.01

and printevery equal to 10. Reaching the convergence threshold of 0.01 takes 5 iterations for policy iteration and 25 iterations for value iteration. The overall runtime for policy is 0.474 seconds, and the time per iteration is 0.095 seconds. The overall runtime for value is 1.394 seconds, and the time per iteration is 0.056 seconds. While policy iteration takes about 70% longer per iteration, it takes far fewer iterations to reach the same convergence thresholds, so its overall runtime is about a third of value iteration's runtime. This makes sense since the computational complexity of policy iteration is higher than that of value iteration.

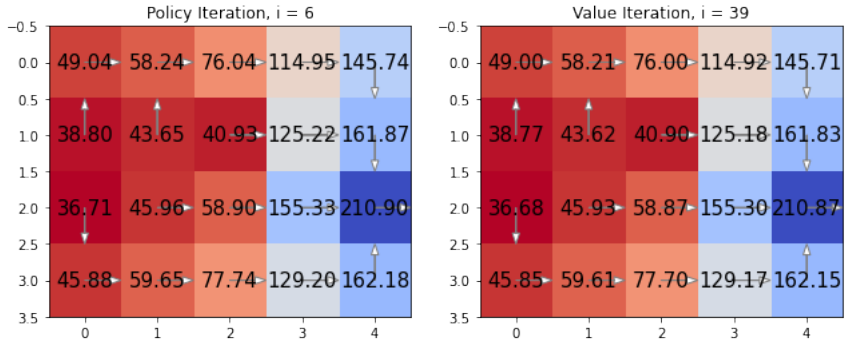
4. The policy and value results for gamma equal to 0.6 are:



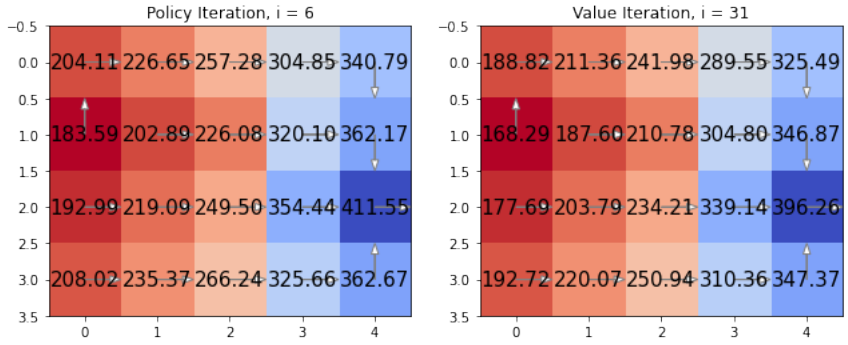
The policy and value results for gamma equal to 0.7 are:



The policy and value results for gamma equal to 0.8 are:



The policy and value results for gamma equal to 0.9 are:



As γ increases, we discount future rewards less or equivalently we value the future more. Mathematically this leads to higher values for each state as the future component of each value increases. In addition, we see a policy change on the left half of the grid as γ increases. Since we care more about the future, we're willing to put off the smaller short term reward in the top left for the higher rewards on the right side of the grid. Thus the policy arrows in the second quadrant switch from pointing towards the top left corner to pointing towards the right side as γ increases.

I ran all experiments with ct equal to 0.01, and printevery equal to 10. The runtime has decreased for both policy and value iteration when gamma is 0.6 since both methods need fewer iterations to converge (4 and 18 iterations, respectively). The runtime has increased for both policy and value iteration when gamma is 0.8 since both methods need more iterations to converge (6 and 39 iterations, respectively). The runtime has increased for both policy and value iteration when gamma is 0.9 since both methods need more iterations to converge (6 and 31 iterations, respectively).

5. Since positive rewards send us to a state with zero reward, the value of the positive reward states will simply be the reward amount. Thus with sufficiently high γ, the policy will be to avoid the short term gain (since this would be the only possible gain) and wait to get to the bigger reward. For lower values of γ, the optimal policy might still choose to go for the lower short term gain even though that ends the game.

Problem 3 (Reinforcement Learning, 20 pts)

In 2013, the mobile game *Flappy Bird* took the world by storm. You'll be developing a Q-learning agent to play a similar game, *Swingy Monkey* (See Figure 1a). In this game, you control a monkey that is trying to swing on vines and avoid tree trunks. You can either make him jump to a new vine, or have him swing down on the vine he's currently holding. You get points for successfully passing tree trunks without hitting them, falling off the bottom of the screen, or jumping off the top. There are some sources of randomness: the monkey's jumps are sometimes higher than others, the gaps in the trees vary vertically, the gravity varies from game to game, and the distances between the trees are different. You can play the game directly by pushing a key on the keyboard to make the monkey jump. However, your objective is to build an agent that *learns* to play on its own.

You will need to install the `pygame` module (http://www.pygame.org/wiki/GettingStarted).

Task: Your task is to use Q-learning to find a policy for the monkey that can navigate the trees. The implementation of the game itself is in file `SwingyMonkey.py`, along with a few files in the `res/` directory. A file called `stub.py` is the starter code for setting up your learner that interacts with the game. This is the only file you need to modify (but to speed up testing, you can comment out the animation rendering code in `SwingyMonkey.py`). You can watch a YouTube video of the staff Q-Learner playing the game at http://youtu.be/l4QjPr1uCac. It figures out a reasonable policy in a few dozen iterations. You'll be responsible for implementing the Python function `action_callback`. The action callback will take in a dictionary that describes the current state of the game and return an action for the next time step. This will be a binary action, where 0 means to swing downward and 1 means to jump up. The dictionary you get for the state looks like this:

```
{ 'score': <current score>,
  'tree': { 'dist': <pixels to next tree trunk>,
            'top': <height of top of tree trunk gap>,
            'bot': <height of bottom of tree trunk gap> },
  'monkey': { 'vel': <current monkey y-axis speed>,
              'top': <height of top of monkey>,
              'bot': <height of bottom of monkey> }}
```

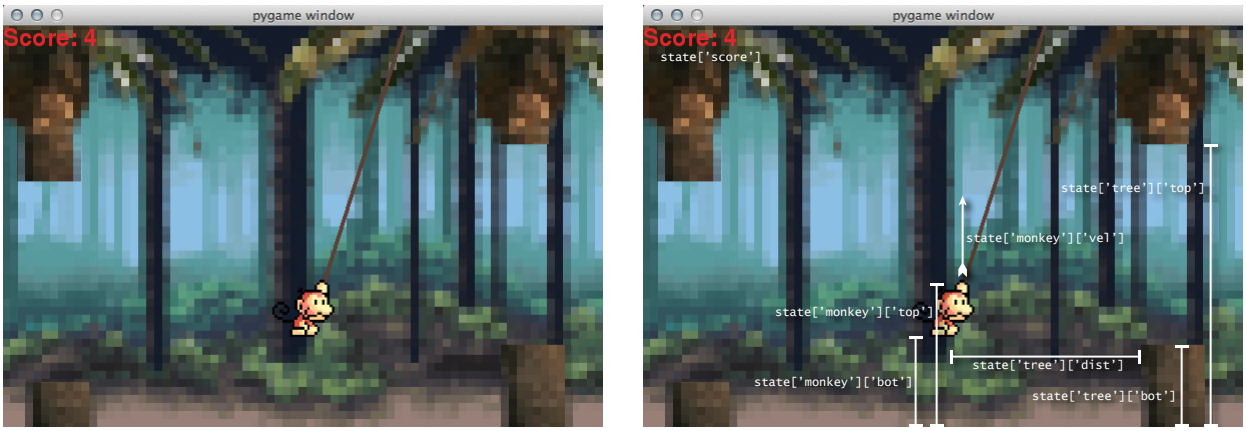
All of the units here (except score) will be in screen pixels. Figure 1b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

Requirements

Code: First, you should implement Q-learning with an ϵ-greedy policy yourself. You can increase the performance by trying out different parameters for the learning rate α, discount rate γ, and exploration rate ϵ. *Do not use outside RL code for this assignment.* Second, you should use a method of your choice to further improve the performance. This could be inferring gravity at each epoch (the gravity varies from game to game), updating the reward function, trying decaying epsilon greedy functions, changing the features in the state space, and more. One of our staff solutions got scores over 800 before the 100th epoch, but you are only expected to reach scores over 50 before the 100th epoch. **Make sure to turn in your code!**

Evaluation: In 1-2 paragraphs, explain how your agent performed and what decisions you made and why. Make sure to provide evidence where necessary to explain your decisions. You must include in your write up at least one plot or table that details the performances of parameters tried (i.e. plots of score vs. epoch number for different parameters).

Note: Note that you can simply discretize the state and action spaces and run the Q-learning algorithm. There is no need to use complex models such as neural networks to solve this problem, but you may do so as a fun exercise.



(a) SwingyMonkey Screenshot

(b) SwingyMonkey State

Figure 1: (a) Screenshot of the Swingy Monkey game. (b) Interpretations of various pieces of the state dictionary.

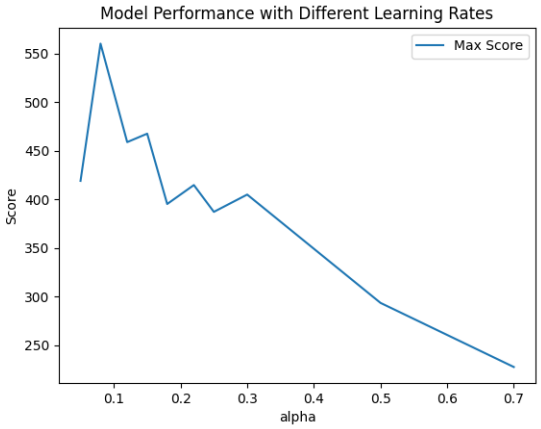
Solution:

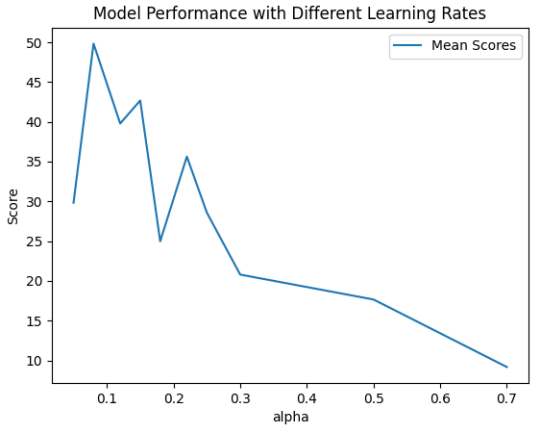
My code is:

```
def action_callback(self, s):
    if self.last_state == None:
        self.last_state = s
        self.last_action = int(npr.rand() < 0.5)
        return self.last_action

    a = self.last_action
    r = self.last_reward
    old_x, old_y = self.discretize_state(self.last_state)
    curr_x, curr_y = self.discretize_state(s)
    q_val = self.Q[a, old_x, old_y]
    best_q = np.max(self.Q[:, curr_x, curr_y])
    self.Q[a, old_x, old_y] = (1 - self.alpha) * q_val + self.alpha * (r + self.gamma *
                                                        best_q)

    rand_move = int(npr.rand() < 0.5)
    new_action = np.argmax(self.Q[:, curr_x, curr_y]) if npr.rand() > self.epsilon else
                                                        rand_move
    self.last_action = new_action
    self.last_state = s
    return self.last_action
```

I plotted the model's performance with multiple values for each parameter: the learning rate α, the discount factor γ, and the exploration rate ϵ. For every tested value of the parameter, I ran 5 trials with 100 epochs each. Then I took the scores from the last 50 epochs and found the mean and max scores. The logic here is that many of the earlier scores are going to be low while the model's learning, and the last 50 scores are more reflective of how much the model has actually learned given some time to train. Then I averaged the mean and max scores over the 5 trials. My default values for the parameters where 0.001 for ϵ, 0.1 for α, and 0.9 for γ (based on Matt Shabet's suggestion on Ed).

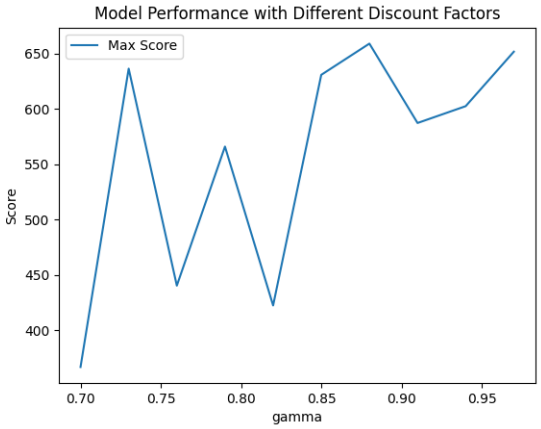
The results for α are:

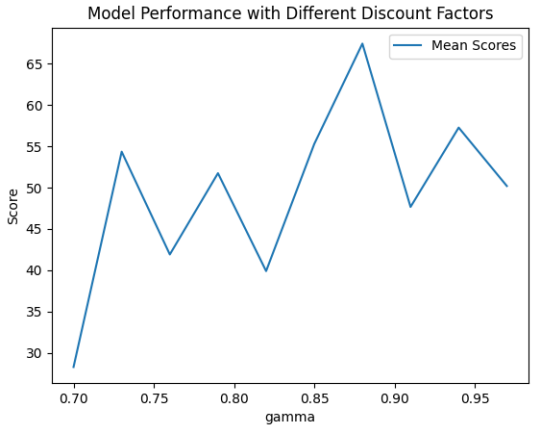




The optimal value of α is about 0.08, and this is apparent in both metrics.

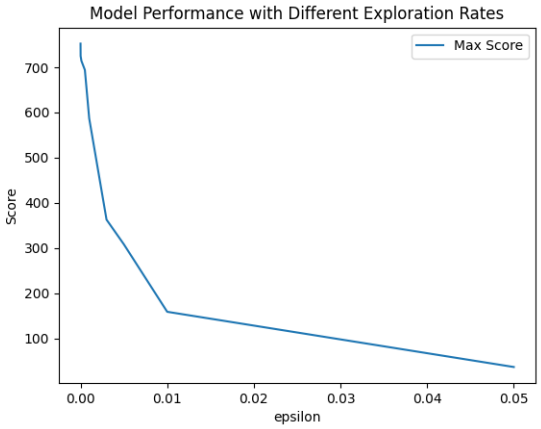
The results for γ are:

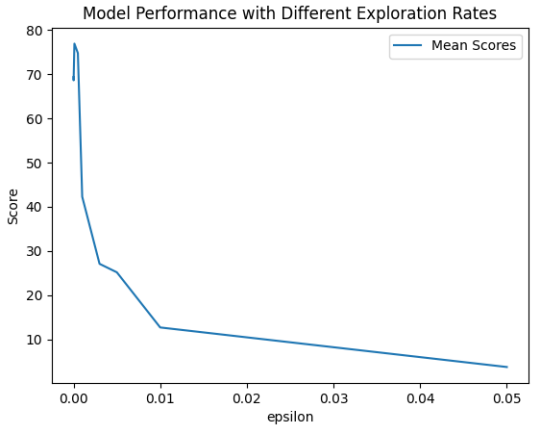




The optimal value of γ is about 0.88. While lower values of γ did pretty well on the max score metric, 0.88 did better on both metrics — particularly the mean score. Note that I did try much lower values of γ, but I plotted just these higher values since it was clear that the best values were in this range and the restriction made the graph easier to read.

The results for ϵ are:





For ϵ, it kind of seemed like the model did better the farther you pushed down ϵ. In some cases, going all the way down to 0 would make the performance worse, but I couldn't get any consistent results. I think a good value of ϵ is 0.00001.

The model was doing pretty well with these parameters. The first third of epochs usually had very low scores as the model was learning, but the model would start getting much higher scores past this point and get more and more consistent as the epochs went on.

I decided to try epsilon decay since I was curious about how the model got better with lower epsilons but still had lots of very low scores at the beginning. It all seems like a very natural change since we intuitively want to explore more when we know less.

I experimented with different combinations of starting initial epsilon value and exponential decay rate. I was able to get decent performance with 0.01 as the decay rate and 0.8 as the starting value. This means that the epsilon value gets multiplied by 0.01 at each epoch.

Name

Christy Jestin

Collaborators and Resources

I worked with David Qian.

Calibration

I spent about 16 hours.