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:

$$\begin{aligned}
\Pr(R = 1) &= 0.25 \\
\Pr(S = 1) &= 0.5 \\
\Pr(G = 1 | R = 0, S = 0) &= 0 \\
\Pr(G = 1 | R = 1, S = 0) &= .75 \\
\Pr(G = 1 | R = 0, S = 1) &= .75 \\
\Pr(G = 1 | R = 1, S = 1) &= 1
\end{aligned}$$

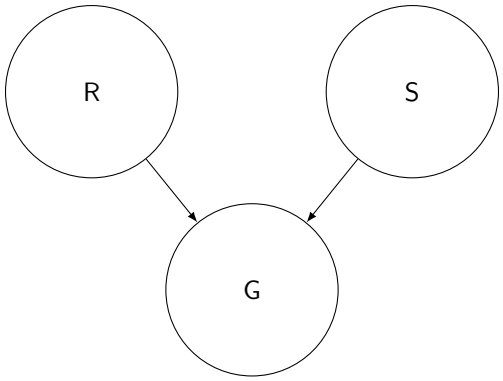
1. Draw the graphical model corresponding to the factorization. Are R and S independent? [Feel free to use facts you have learned about studying independence in graphical models.]
2. You notice it is raining and check on the sprinkler without checking the grass. What is the probability that it is on?
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:
Solution 1:



Solution 2:
S and R are d-separated by G when G is not observed, i.e. S and R are independent conditional upon not observing G. Therefore, the probability that the sprinkler is on given it is raining (without checking the grass) is given by:

$$\begin{aligned}\Pr(S=1|R=1) &= \Pr(S=1) \\ &= 0.5\end{aligned}$$

Solution 3:
Using Bayes' rule, the probability that the sprinkler is on given the grass is wet (without checking that it is raining) is given by:

$$\Pr(S=1|G=1) = \frac{\Pr(G=1|S=1)\Pr(S=1)}{\Pr(G=1)}$$

We can calculate the components of this expression individually.

Using the law of total probability by conditioning on the possible states of R and S and noting (from 2) that R and S are d-separated, when G is not observed:

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

Using the law of total probability by conditioning on the possible states of R:

$$\begin{aligned}\Pr(G=1|S=1) &= \Pr(G=1|S=1,R=1)\Pr(R=1) \\ &\quad + \Pr(G=1|S=1,R=0)\Pr(R=0) \\ &= 1(0.25) + (0.75)(0.75) \\ &= \frac{13}{16}\end{aligned}$$

And, using what is already given:

$$\Pr(S = 1) = 0.5$$

Putting this all together, we get that:

$$\Pr(S = 1|G = 1) = \frac{\frac{13}{16} \cdot (0.5)}{0.5} = \frac{13}{16} = \boxed{0.8125}$$

Solution 4:
Using simple rules of conditional probability, the probability that the sprinkler is on given the grass is wet and given that it is raining is:

$$\begin{aligned}
\Pr(S = 1|R = 1, G = 1) &= \frac{\Pr(G = 1, R = 1, S = 1)}{\Pr(R = 1, G = 1)} \\
&= \frac{\Pr(G = 1|R = 1, S = 1)\Pr(R = 1)\Pr(S = 1)}{\Pr(R = 1, G = 1, S = 1) + \Pr(R = 1, G = 1, S = 0)} \\
&= \frac{\Pr(G = 1|R = 1, S = 1)\Pr(R = 1)\Pr(S = 1)}{\Pr(R = 1)\Pr(S = 1)\Pr(G = 1|R = 1, S = 1) + \Pr(R = 1)\Pr(S = 0)\Pr(G = 1|R = 1, S = 0)}
\end{aligned}$$

which has been simplified using the law of total probability in the denominator by conditioning on the possible states of S and noting (from 2) that R and S are d-separated, when G is not observed.

$$\begin{aligned}
\therefore \Pr(S = 1|R = 1, G = 1) &= \frac{(0.25)(0.5)(1)}{(0.25)(0.5)(1) + (0.25)(0.5)(0.75)} \\
&= \frac{4}{7} \approx \boxed{0.571}
\end{aligned}$$

Solution 5:
Noticing that the probability calculated in 4 is lower than the probability calculated in 3, we can conclude that observing there being rain "explains away" the effect of the sprinkler on the grass. In other words, having observed that the grass is wet and that it is raining, it makes it less likely that the grass was made wet by the sprinkler too, than had we just observed the grass being wet.

Note, that this explaining away effect at G only occurs because we observe G in both cases, thereby unblocking the path between R and S, meaning they are no longer independent, so knowing either of R or S changes the amount we expected the other of R or S to contribute to G - exactly the "explaining away" effect.

Solution 6:

$$\begin{aligned}
\Pr(R) &= \sum_C \sum_G \sum_S \Pr(C, R, G, S) \\
&= \sum_C \sum_G \sum_S \Pr(C) \Pr(R|C) \Pr(S|C) \Pr(G|R, S) \\
&= \sum_C \Pr(C) \Pr(R|C) \sum_G \sum_S \Pr(S|C) \Pr(G|R, S) \\
&= \sum_C \Pr(C) \Pr(R|C) \sum_G \sum_S \psi_1(G, C, R, S) \\
&= \sum_C \Pr(C) \Pr(R|C) \sum_G \psi_2(G, C, R) \\
&= \sum_C \psi_3(C, R) \\
&= \Pr(R)
\end{aligned}$$

Solution 7:

$$\begin{aligned}
\Pr(R) &= \sum_S \sum_G \sum_C \Pr(C, R, G, S) \\
&= \sum_S \sum_G \sum_C \Pr(C) \Pr(R|C) \Pr(S|C) \Pr(G|R, S) \\
&= \sum_S \sum_G \Pr(G|R, S) \sum_C \Pr(C) \Pr(R|C) \Pr(S|C) \\
&= \sum_S \sum_G \Pr(G|R, S) \sum_C \psi_1(C, R, S) \\
&= \sum_S \sum_G \psi_2(G, R, S) \\
&= \sum_S \psi_3(R, S) \\
&= \Pr(R)
\end{aligned}$$

Solution 8:
In question 6, we see that the variable elimination involves summing over a k^4 sized object $\psi_1(G, C, R, S)$, followed by a k^3 sized object $\psi_2(G, C, R)$, followed by a k^2 sized object $\psi_3(C, R)$. Therefore, the variable elimination ordering S, G, C has time complexity:

$$O(k^4) + O(k^3) + O(k^2) = O(k^4)$$

In question 7, we see that the variable elimination involves summing over a k^3 sized object $\psi_1(C, R, S)$, followed by a k^3 sized object $\psi_2(G, R, S)$, followed by a k^2 sized object $\psi_3(R, S)$. Therefore, the variable elimination ordering C, G, S has time complexity:

$$O(k^3) + O(k^3) + O(k^2) = O(k^3)$$

Therefore, the variable elimination ordering C, G, S takes 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.

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.

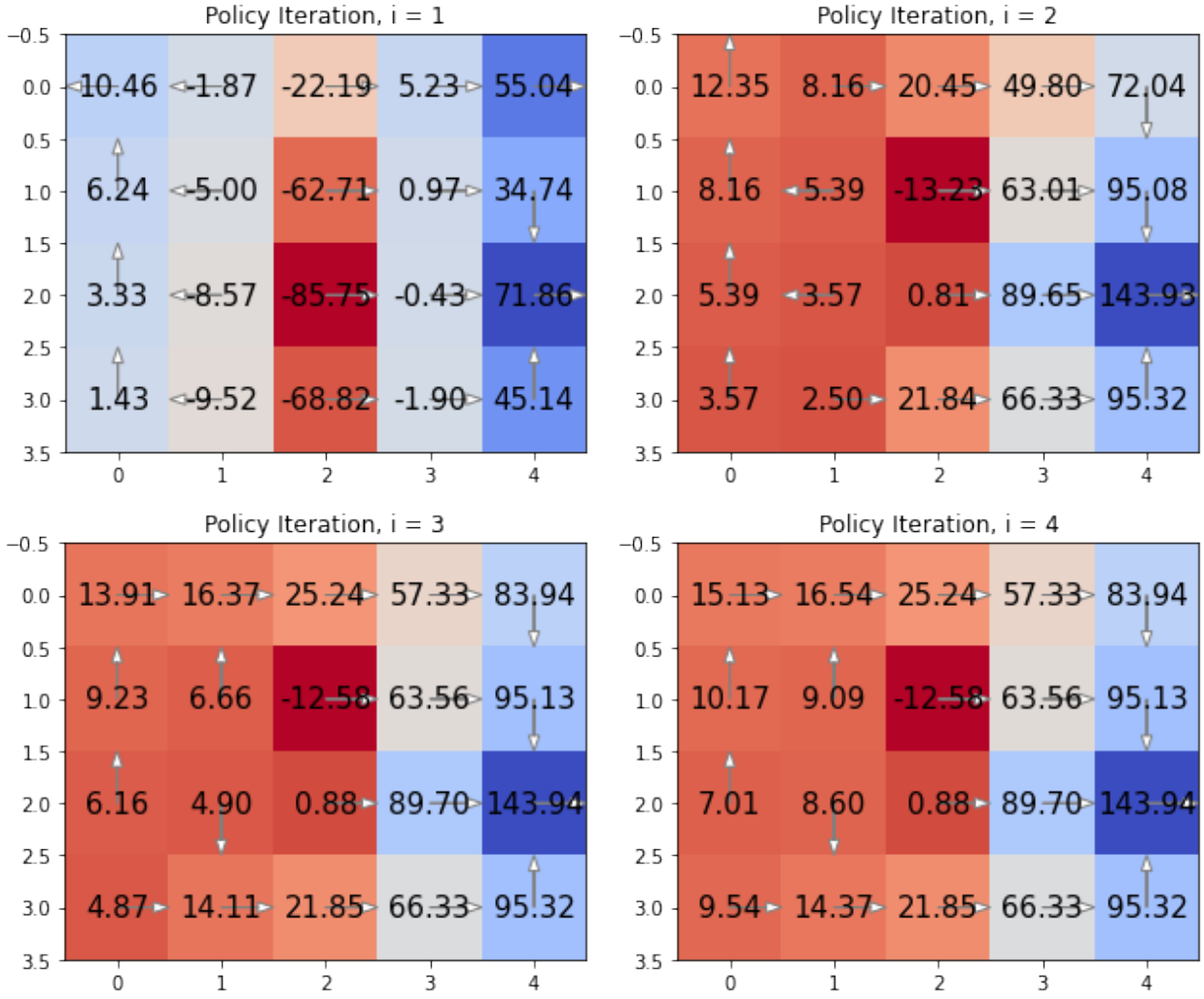
Solution:
Solution 1(a):

```
def policy_evaluation(pi, gamma):
theta = 0.0001
V_0 = np.zeros(num_states)
V = np.zeros(num_states)

while True:
    V_0 = np.array(V)
    for state in range(num_states):
        current_reward = get_reward(state)
        expected_reward = 0
        for next_state in range(num_states):
            expected_reward += gamma * get_transition_prob(state, pi[state], next_state)
                                                                * V_0[next_state]
        V[state] = current_reward + expected_reward

    if max(abs(V - V_0)) < theta:
        break

return V
```

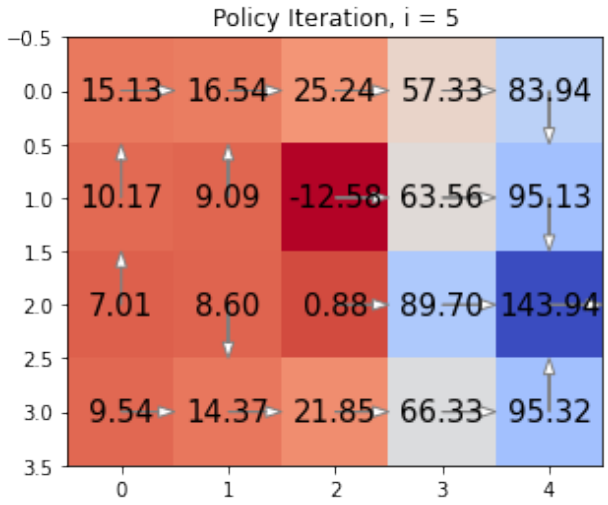
Solution 1(b):

```
def update_policy_iteration(V, gamma):
pi_new = np.zeros(num_states)

for state in range(num_states):
    max_val = get_reward(state)
    for next_state in range(num_states):
        max_val += gamma * get_transition_prob(state, 0, next_state) * V[next_state]

    for action in range(1,num_actions):
        current_reward = get_reward(state)
        expected_reward = 0
        for next_state in range(num_states):
            expected_reward += gamma * get_transition_prob(state, action, next_state) *
                                                            V[next_state]
        new_val = current_reward + expected_reward
        if new_val > max_val and pi_new[state] != action:
            pi_new[state] = action
            max_val = new_val

return pi_new
```

Solution 1(c):


Policy Iteration, i = 1
10.46 -1.87 -22.19 5.23 55.04
6.24 -5.00 -62.71 0.97 34.74
3.33 -8.57 -85.75 -0.43 71.86
1.43 -9.52 -68.82 -1.90 45.14
Policy Iteration, i = 2
12.35 8.16 20.45 49.80 72.04
8.16 5.39 -13.23 63.01 95.08
5.39 3.57 0.81 89.65 143.93
3.57 2.50 21.84 66.33 95.32
Policy Iteration, i = 3
13.91 16.37 25.24 57.33 83.94
9.23 6.66 -12.58 63.56 95.13
6.16 4.90 0.88 89.70 143.94
4.87 14.11 21.85 66.33 95.32
Policy Iteration, i = 4
15.13 16.54 25.24 57.33 83.94
10.17 9.09 -12.58 63.56 95.13
7.01 8.60 0.88 89.70 143.94
9.54 14.37 21.85 66.33 95.32

Solution 1(d):

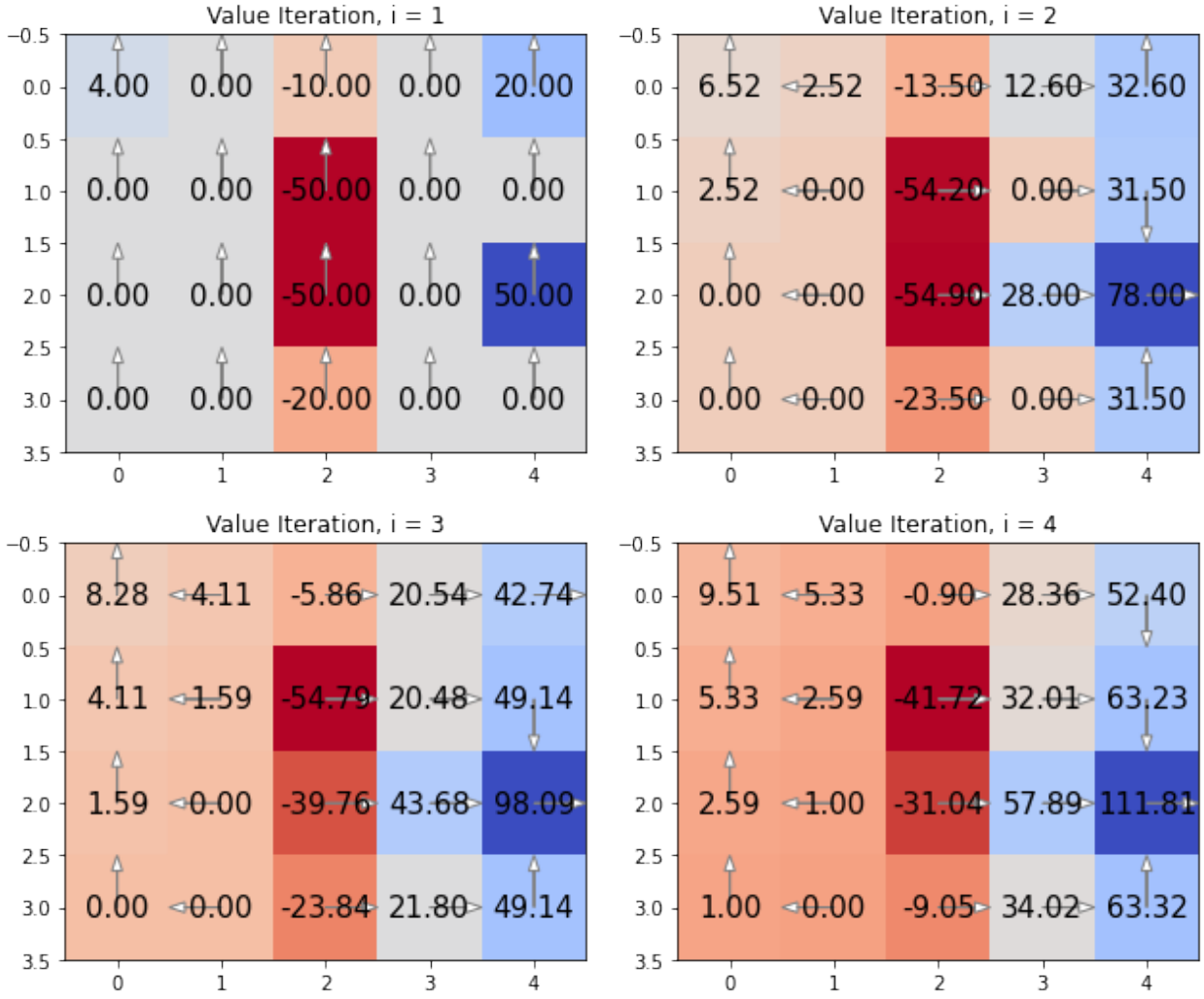


The policy evaluation/iteration algorithm takes 5 iterations to converge. Changing the value of `ct` does not change the number of iterations until convergence, i.e. it remains at 5. This is intuitive because policy evaluation/iteration should converge to the optimal policy exactly when it converges.

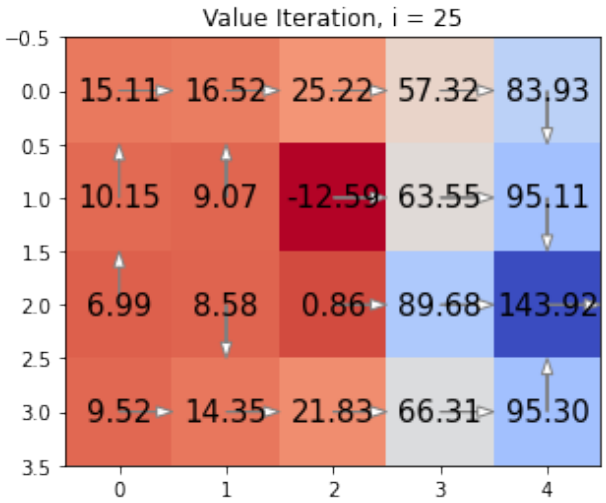
Solution 2(a):

```
def update_value_iteration(V, pi, gamma):
V_new = np.zeros(num_states)
pi_new = np.zeros(num_states)

for state in range(num_states):
    max_val = -99999999999999
    for action in range(0,num_actions):
        current_reward = get_reward(state)
        expected_reward = 0
        for next_state in range(num_states):
            expected_reward += gamma * get_transition_prob(state, action, next_state) *
                                                                    V[next_state]
        new_val = current_reward + expected_reward
        if new_val > max_val:
            max_val = new_val
            V_new[state] = max_val
            pi_new[state] = action

return V_new, pi_new
```

Solution 2(b):


Value Iteration, i = 1
4.00 0.00 -10.00 0.00 20.00
0.00 0.00 -50.00 0.00 0.00
0.00 0.00 -50.00 0.00 50.00
0.00 0.00 -20.00 0.00 0.00
Value Iteration, i = 2
6.52 2.52 -13.50 12.60 32.60
2.52 0.00 -54.20 0.00 31.50
0.00 0.00 -54.90 28.00 78.00
0.00 0.00 -23.50 0.00 31.50
Value Iteration, i = 3
8.28 4.11 -5.86 20.54 42.74
4.11 1.59 -54.79 20.48 49.14
1.59 0.00 -39.76 43.68 98.09
0.00 0.00 -23.84 21.80 49.14
Value Iteration, i = 4
9.51 5.33 -0.90 28.36 52.40
5.33 2.59 -41.72 32.01 63.23
2.59 1.00 -31.04 57.89 111.81
1.00 0.00 -9.05 34.02 63.32

Solution 2(c):



`ct`	Number of iterations until convergence
0.01	25
0.001	31
0.0001	38

Decreasing the value of `ct` increases the number of iterations until convergence. This is intuitive because value iteration does not converge exactly, so decreasing the convergence threshold makes it more difficult to achieve the level of accuracy required to converge, and requires more iterations.

Solution 3:
Using a convergence tolerance of 0.01:

	Policy iteration	Value iteration
Number of iterations	5	25
Time per iteration	0.682	0.260
Overall runtime	3.41	6.49

Policy iteration takes less iterations to converge and takes less time per iteration but takes more time overall to run, when compared to value iteration. This is intuitive, since policy iteration has runtime of $O(|S||A|L + |S|^3)$ per step, and value iteration has runtime of $O(|S||A|L)$ per step, where $|S|$ is defined as the number of states in the world, $|A|$ is defined as the number of possible actions, and L is defined as the maximum number of states reachable from any states through any action. However, policy iteration converges faster overall as it converges exactly and does not need to spend time, unlike value iteration, improving the accuracy of the value function up to the second decimal place, until it reaches the convergence threshold. Thus, value iteration ends up taking many more, albeit shorter iterations, which end up taking longer overall than policy iteration's few, slower iterations.

Solution 4:
Using a convergence tolerance of 0.01:

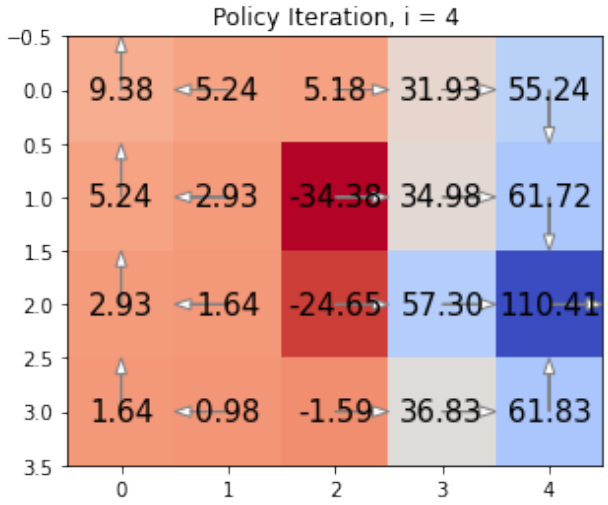


Figure 1: γ = 0.6

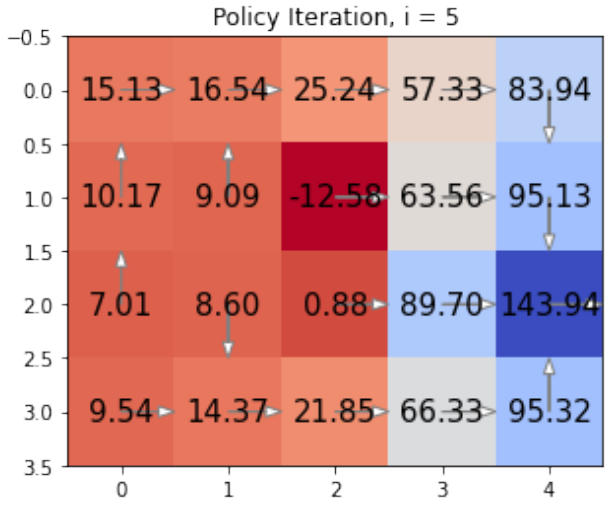


Figure 2: γ = 0.7

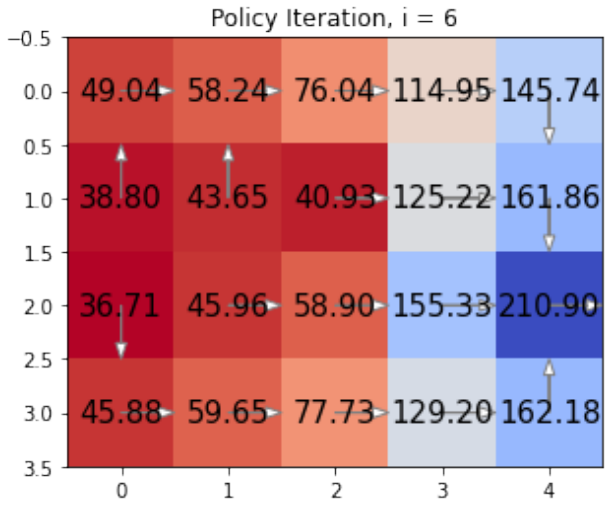


Figure 3: γ = 0.8

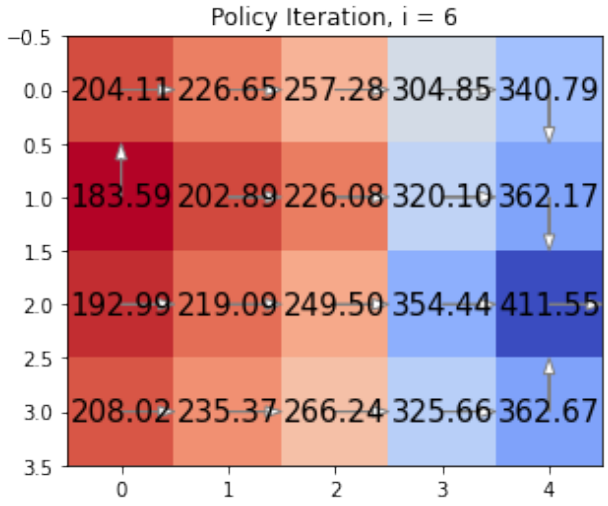


Figure 4: γ = 0.9

γ	Policy iteration runtime	Value iteration runtime
0.6	2.27	4.78
0.7	3.41	6.49
0.8	6.24	10.4
0.9	9.73	21.6

Increasing the magnitude of γ decreases the amount of discounting on rewards from future states. Thus, it makes sense for policies to travel further in order to collect more rewards in the future, and the agent travels for more time. This is the case for both value and policy iteration, so the overall runtime increases for both algorithms as γ increases.

Solution 5:
Larger values of γ increase the value of future rewards as the level of discounting decreases. Thus, as γ tends

towards 1, policies will recommend longer paths, as it becomes worthwhile travelling the long path from the start to the grid-square with the highest reward (R=50). Meanwhile, lower values of γ decrease the value of future rewards as the level of discounting increases. Thus, policies will take shorter paths, given lower values of γ, e.g. to the grid-square with R=4, as it is not worth travelling further to only get the heavily discounted R=50 in the future.

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 5a). 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> }}
```

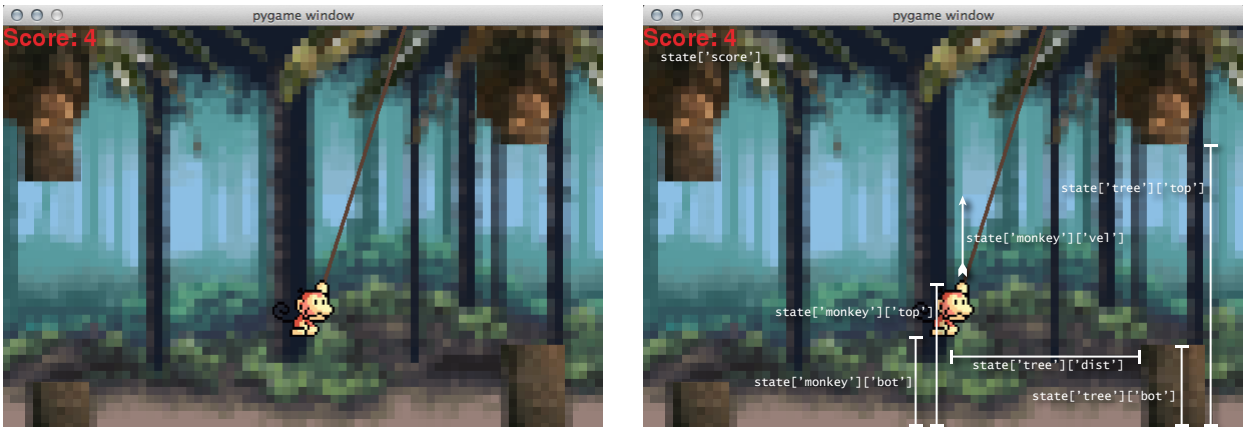
All of the units here (except score) will be in screen pixels. Figure 5b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

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

Solution:
I implemented Q-learning with an ϵ-greedy policy in `stub.py` and then sought to improve performance by inferring gravity at each epoch in `stub_grav.py`. The results were not as straightforward as may have been expected - inferring gravity did not improve results on all of the test metrics I discuss below.

Implementation: In order to begin the Q-learning algorithm, we must have a tuple of the first (state, action, reward). By default, the monkey was initialized to not jump (action 0) from its starting state. Then, beginning with the second state, the monkey followed an ϵ-greedy policy, taking a random next action with probability ϵ (exploring) and taking a greedy next action with probability $1 - \epsilon$ (exploiting, by taking the next action that maximized the Q-function). Regardless of this choice, the Q-function would be updated according to the update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

where $Q(s,a)$ is the current value of the Q-function at state s and taking action a - $Q(s,a)$ starts by being initialized to 0 $\forall\,(s,a)$; α is the learning rate; $R(s,a)$ is the reward at state s and taking action a; γ is the discount rate on the value of future states and actions.

The only key difference between `stub.py` and `stub_grav.py`, is that, in the former, s is a tuple of (x,y) - the discretized coordinates of the monkey - whilst, in the latter, s is a tuple of (x,y,g) - the discretized coordinates of the monkey and the level of gravity g in the game. g was calculated by finding the difference in the actual (not discretized) y-coordinate of the monkey between the first two states. In turned out that there were only two possible levels of gravity in the game, which I will refer to as "low gravity" and "high gravity". Intuitively, the monkey fell further between state 0 and state 1 in "high gravity" games.

The theory behind inferring gravity was that the monkey would have a larger Q-table, from which it could learn about the different value of states and actions in low gravity and high gravity games, and thereby better optimize its policy.

PTO!

Results: Using `stub.py`, I ran the game once (100 epochs) for 3 different values of each of the hyperparameters α, γ, and ϵ. The results were as follows (note, I do not show all 27 combinations of the hyperparameters I tried, just enough to show variation by changing each one):

1 iteration (100 epochs)	Average score across all epochs	Max score across all epochs
$\epsilon = 0.01$ ($\alpha = 0.1$, $\gamma = 0.9$)	13.14	232
$\epsilon = 0.001$ ($\alpha = 0.1$, $\gamma = 0.9$)	37.46	498
$\epsilon = 0.0001$ ($\alpha = 0.1$, $\gamma = 0.9$)	46.38	521

1 iteration (100 epochs)	Average score across all epochs	Max score across all epochs
$\alpha = 0.05$ ($\epsilon = 0.0001$, $\gamma = 0.9$)	38.28	408
$\alpha = 0.1$ ($\epsilon = 0.0001$, $\gamma = 0.9$)	46.38	521
$\alpha = 0.2$ ($\epsilon = 0.0001$, $\gamma = 0.9$)	33.00	512

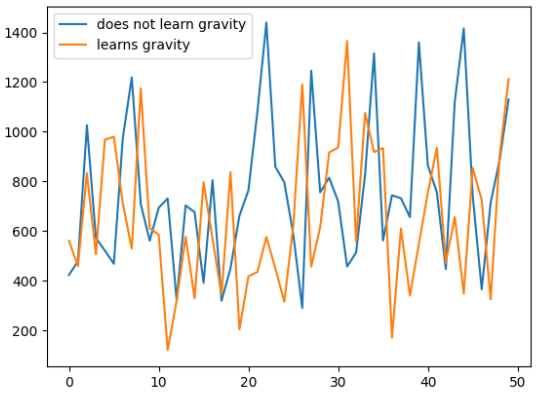
1 iteration (100 epochs)	Average score across all epochs	Max score across all epochs
$\gamma = 0.8$ ($\alpha = 0.1$, $\epsilon = 0.0001$)	39.9	501
$\gamma = 0.9$ ($\alpha = 0.1$, $\epsilon = 0.0001$)	46.38	521
$\gamma = 1.0$ ($\alpha = 0.1$, $\epsilon = 0.0001$)	43.28	493

The values of the hyperparameters $\epsilon = 0.0001$, $\alpha = 0.1$, and $\gamma = 0.9$ produced the best max and average scores. Intuitively, these feel relatively uncontentious - ϵ is low because the monkey cannot get a good average or max score if it does too much exploring in 100 epochs, α is quite low because we do not want to change our Q-table massively at every timestep, and γ is quite high because there is little reason to discount the future heavily (points later are not much less valuable than points now). I chose to use these hyperparameters for the rest of my analysis.

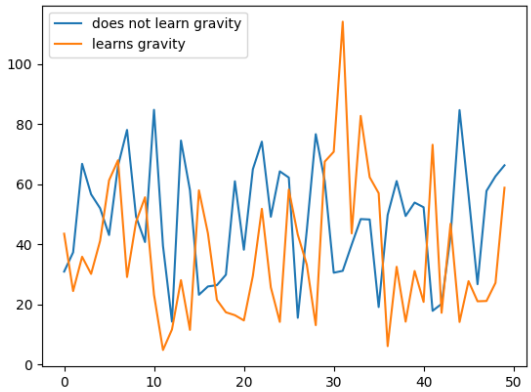
In order to test the difference between `stub.py` and `stub_grav.py` using the optimal parameters, I ran the game 50 times (100 epochs each time), and calculated the following test metrics to demonstrate performance:

50 iterations (100 epochs per iteration)	`stub.py`	`stub_grav.py`
Average score across all epochs in all iterations	48.60	37.37
Max score across all epochs in all iterations	1416	1366
Average of max score in each iteration	753.28	652.98
Variance of max score in each iteration	86258.12	82978.98

PTO!



Max scores in each of 50 iterations



Average scores in each of 50 iterations

A first glance at the graphs above suggests that there is very little difference between the non-gravity-inferring monkey and gravity-inferring monkey. This perhaps suggests that 100 epochs simply is not enough to update the gravity-inferring monkey's Q-table to an extent where it improves the monkey's performance. However, the tabulated data reveals an interesting result: while the non-gravity-inferring monkey got a higher score on average, a higher overall max score, and a higher average max score across all iterations, the gravity-inferring-monkey had less variation (lower variance) in its max score across all iterations.

Another clear finding from the graphs is that both the non-gravity-inferring monkey and gravity-inferring monkey perform very variably, sometimes producing max scores over 1000, and sometimes failing to get a score above 10. After looking into the results from 1 iteration of the game (with 100 epochs), I discovered that the monkey consistently performs significantly worse in low gravity epochs than high gravity epochs:

1 iteration (100 epochs)	`stub.py`	`stub_grav.py`
Average score across all low gravity epochs	0.52	0.27
Average score across all high gravity epochs	65.79	58.58

Consistent with the above findings, the non-gravity-inferring monkey performs a little better on average than the gravity-inferring monkey across epochs.

Conclusion: I would probably choose to use the non-gravity-inferring monkey for the SwingyMonkey game in its current form, since it produces better scores on average. However, in another game format, whereby you were penalized for getting low scores, I would choose to use the gravity-inferring monkey, that more reliably produces scores close to its average max score.

Name

Ben Ray

Collaborators and Resources

Whom did you work with, and did you use any resources beyond cs181-textbook and your notes?
Ty Geri
CS 181 OHs

Calibration

Approximately how long did this homework take you to complete (in hours)? 14