CS181-S22

Assignment #6

Due: 11:59PM EST, April 29 2022

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?

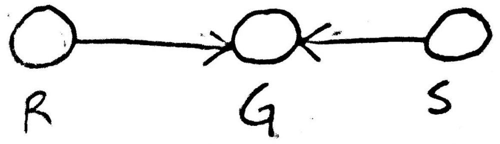
Consider if we introduce a new binary variable, cloudy C, to the the original three binary variables such that the factorization of the joint distribution is now:

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

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:

1. Below is the graphical model for this factorization.



As G is not yet observed, R and S are independent.

2. Given that R and S are independent, the probability that the sprinkler is on given that it is raining is

$$\boxed{P(S=1|R=1)=P(S=1)=0.5}$$

.

3. We want to know the probability that the sprinkler is on given that the grass is wet. By Bayes' Theorem, we have

$$P(S=1|G=1)=\frac{P(G=1|S=1)P(S=1)}{P(G=1)}=\frac{P(G=1|S=1)P(S=1)}{P(G=1|S=1)P(S=1)+P(G=1|S=0)P(S=0)}$$

By the Law of Total Probability, we have

$$P(G=1|S=1)=P(G=1|R=0,S=1)P(R=0)+P(G=1|R=1,S=1)P(R=1)$$
$$=(0.75)(1-0.25)+(1)(0.25)=0.8125$$
$$P(G=1|S=0)=P(G=1|R=0,S=0)P(R=0)+P(G=1|R=1,S=0)P(R=1)$$
$$=(0)(1-0.25)+(0.75)(0.25)=0.1875$$

Plugging these back into our definition of $P(S=1|G=1)$ we get

$$P(S=1|G=1)=\frac{P(G=1|S=1)P(S=1)}{P(G=1|S=1)P(S=1)+P(G=1|S=0)P(S=0)}$$
$$=\frac{(0.8125)(0.5)}{(0.8125)(0.5)+(0.1875)(0.5)}$$
$$\boxed{P(S=1|G=1)=0.8125}$$

4. We want to know the probability that the sprinkler is on given that it is raining and the grass is wet. By Bayes' Theorem we have

$$P(S=1|R=1,G=1)=\frac{P(G=1|R=1,S=1)P(S=1|R=1)}{P(G=1|R=1)}$$

we know that

$$P(G = 1|R = 1, S = 1) = 1$$

and that, since R and S are independent,

$$P(S = 1|R = 1) = P(S = 1) = 0.5$$

Now, in order to calculate $P(G = 1|R = 1)$ we use the Law of Total Probability:

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

$$= (0.75)(0.5) + (1)(0.5) = 0.875$$

Plugging these values into our definition of $P(S = 1|R = 1, G = 1)$ we get

$$P(S = 1|R = 1, G = 1) = \frac{P(G = 1|R = 1, S = 1)P(S = 1|R = 1)}{P(G = 1|R = 1)}$$

$$\boxed{P(S = 1|R = 1, G = 1) = \frac{(1)(0.5)}{(0.875)} \approx 0.571}$$

5. The explaining away effect shown above is that the observance of G makes R a contributor of information to S, despite the fact that, before G was observed, R and S were completely independent.

6. The marginal distribution $P(R)$ for the $C \to G \to S$ ordering is the following

$$P(R) = \sum_{c} P(C)P(R|C) \sum_{G} \sum_{S} P(G|R, S)P(S|C)$$

7. The marginal distribution $P(R)$ for the $S \to G \to C$ ordering is the following

$$P(R) = \sum_{S} \sum_{G} P(G)P(R, S) \sum_{C} P(C)P(R|C)P(S|C)$$

8. The $C \to G \to S$ ordering requires, at its longest step, the calculation of $\sum_S P(G|R, S)P(S|C)$, which requires summing over S for every value in a k by k matrix consisting of all possibilities for G and C. This operation has complexity $O(k^3)$. k here refers to the domain size of the variables which in this case is 2.

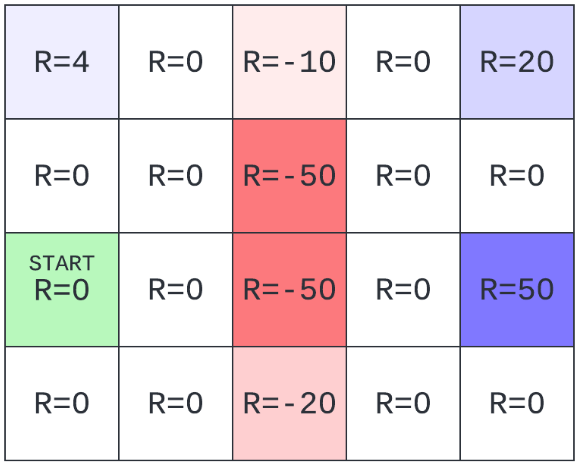
   The longest calculation in the $S \to G \to C$ ordering, on the other hand, is $\sum_C P(C)P(R|C)P(S|C)$. This calculation requires summing over C for every value in a size k vector consisting of all possibilities for S. This operation has complexity $O(k^2)$.

   Based on these complexities, the $S \to G \to C$ ordering requires less computation.

   Note (confusion on this problem): If we are given $P(R|C)$ and $P(C)$ we can find the distribution $P(R)$ by calculating $\sum_C P(R|C)P(C)$, an operation with complexity $O(k)$. In fact, the methods above involve redundant operations such as $\sum_G P(G|R, C) = 1$.

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:



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.

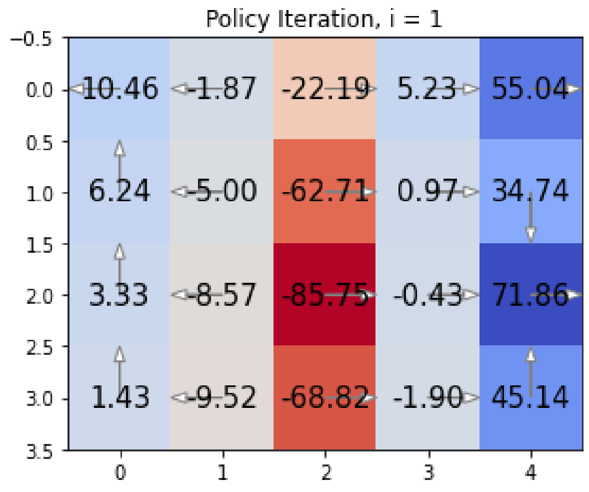
5 Now suppose that the game ends at any state with a positive reward, i.e. it immediately transitions you to a new state with zero reward that you cannot transition away from. What do you expect the optimal policy to look like, as a function of gamma? Numerical answers are not required, intuition is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

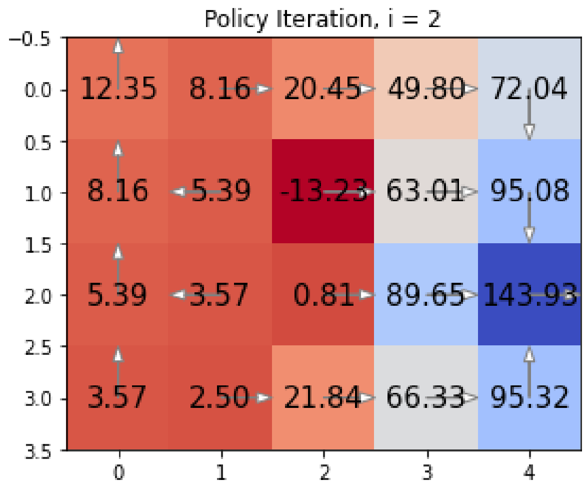
Solution:

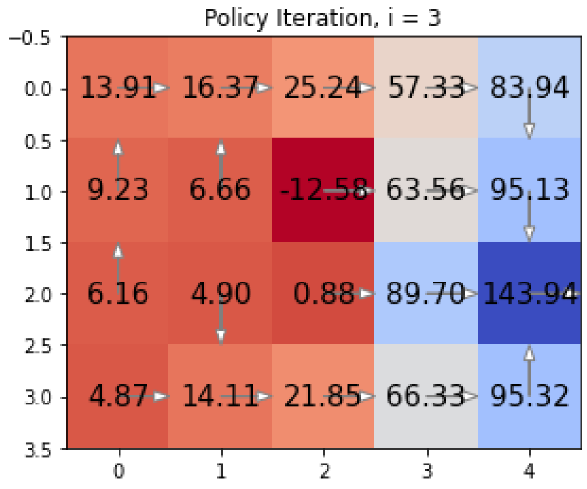
1a. Implemented in `T1.P2.ipynb`.

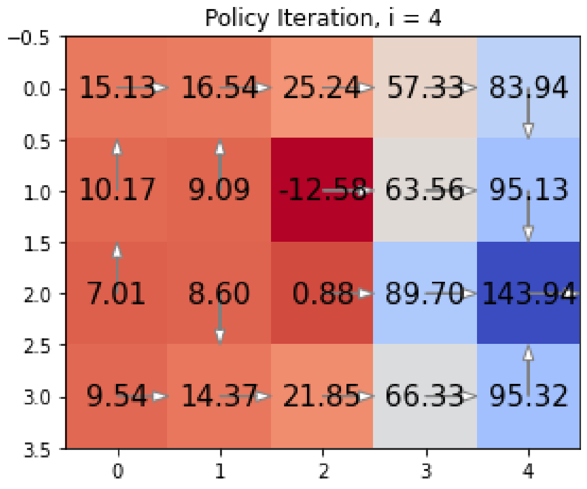
1b. Implemented in `T1.P2.ipynb`.

1c. Below are the plots resulting from setting `max_iter = 4`, `print_every = 1`.

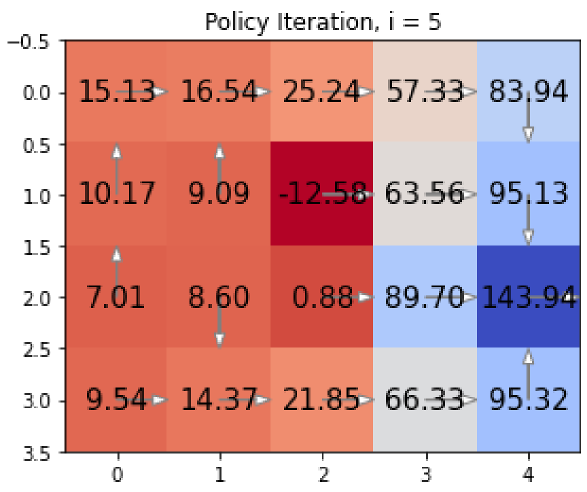








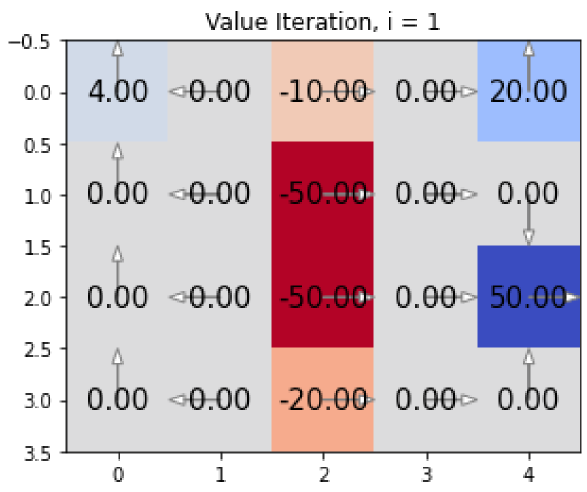
1d. With `ct = 0.01` and `max_iter = 100`, the final learned value function is the following:

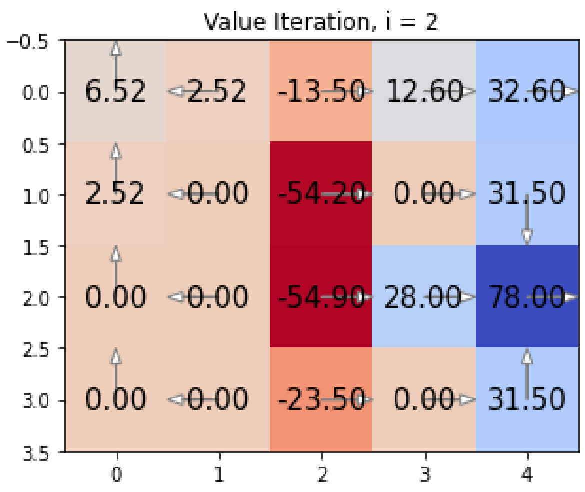


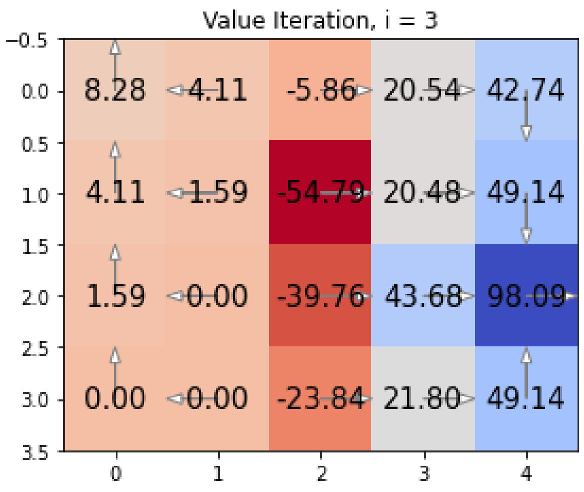
This final learned value function takes 5 iterations to converge. Using `ct = 0.001` does not affect the number of iterations until convergence. Neither does using `ct = 0.0001`.

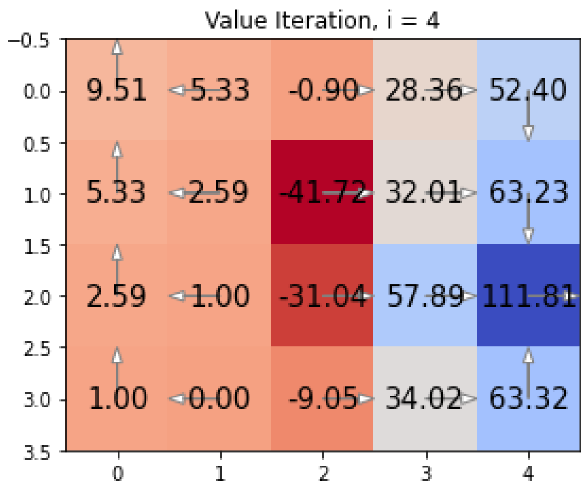
2a. Implemented in `T1.P2.ipynb`.

2b. Below are the plots resulting from setting `max_iter = 4`, `print_every = 1`.

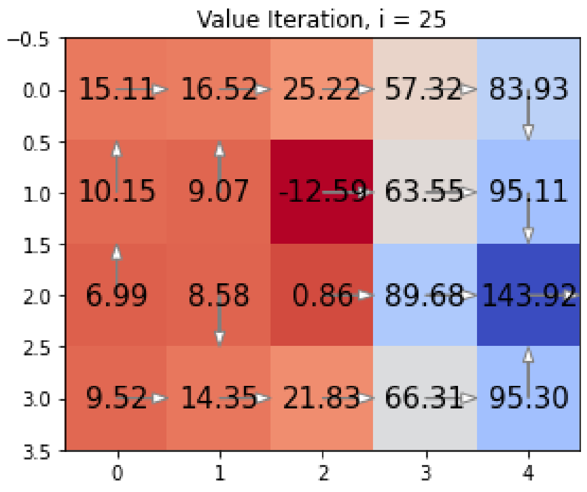








2c. With `ct = 0.01` and `max_iter = 100`, the final learned value function is the following:

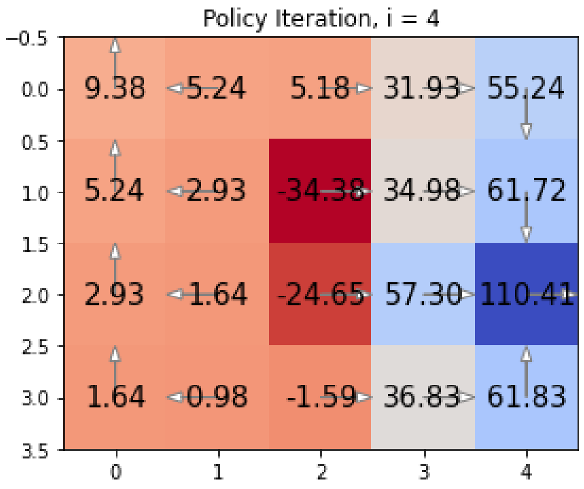


This final learned value function takes 25 iterations to converge. Using `ct = 0.001` changes this to 31 iterations. Using `ct = 0.0001` changes this to 38 iterations.

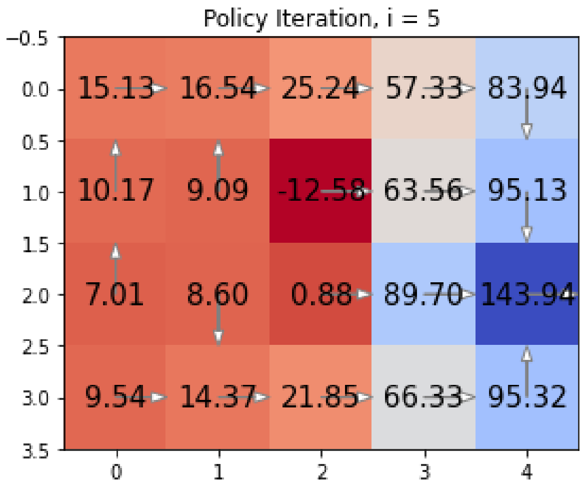
3. Policy iteration requires less iterations and takes considerably more time per iteration. Value iteration requires more iterations and takes considerably less time per iteration. Overall, the tradeoff between number of iterations for each and time per iteration cancels out and both methods have a very similar overall runtime.

   One important thing to note, however, is that Value Iteration is sensitive to the setting for convergence tolerance while Policy Iteration is not. If we want to reach a small convergence tolerance (e.g. `ct = 0.0001`), then Value Iteration takes more iterations and results in a much higher overall runtime as compared with Policy Iteration.

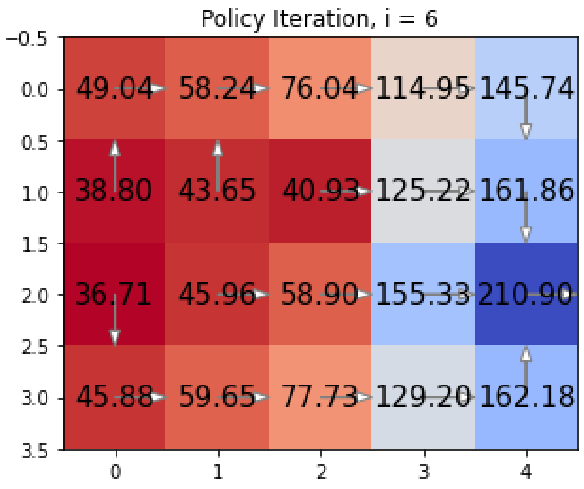
4. We include the plot below. Policy Iteration was used to calculate these.



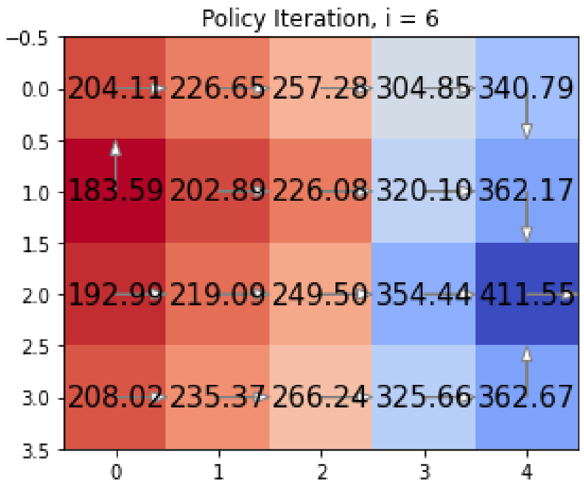
(a) $\gamma = 0.6$



(b) $\gamma = 0.7$



(c) $\gamma = 0.8$



(d) $\gamma = 0.9$

Figure 1: Learned policies with differing discount values (γ)

5. I would expect the optimal policy to, depending on how high gamma is, favor avoiding the squares with reward 4 and 20 in order to prioritize reaching the square with reward 50. With lower values of gamma, the policy will feel fine reaching the squares with the lower positive rewards, and the policy may not look too different from its corresponding policy in the regular game. With high values for gamma, however, it will definitely avoid these lower positive reward value squares.

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 2a). 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> }}
```

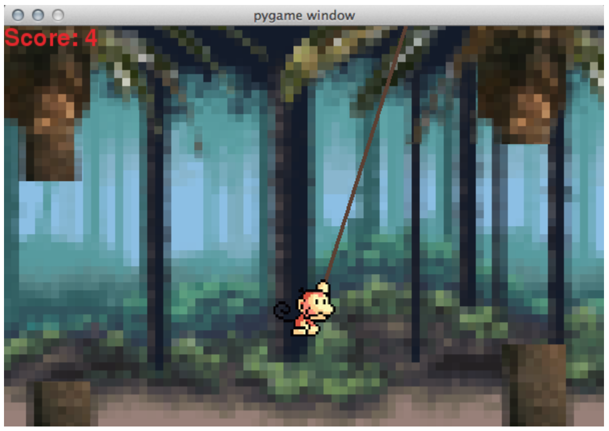
All of the units here (except score) will be in screen pixels. Figure 2b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

Requirements

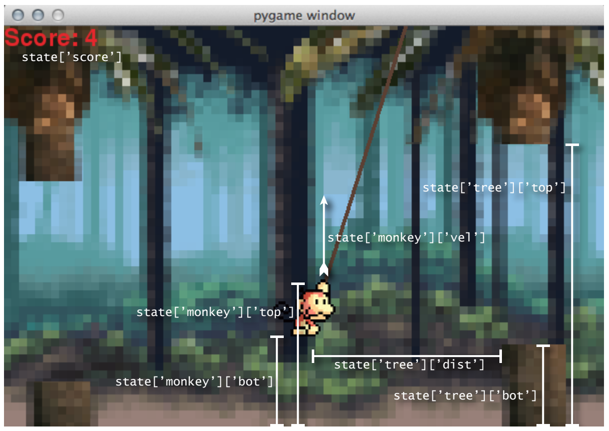
Code: First, you should implement Q-learning with an ϵ-greedy policy yourself. You can increase the performance by trying out different parameters for the learning rate α, discount rate γ, and exploration rate ϵ. *Do not use outside RL code for this assignment.* Second, you should use a method of your 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 2: (a) Screenshot of the Swingy Monkey game. (b) Interpretations of various pieces of the state dictionary.

Solution: Below is the code for my implementation. I added two object variables and so I include the changes to the initialization and reset methods.

```python
def __init__(self):
    self.last_state = None
    self.last_action = None
    self.last_reward = None
    self.is_first_state = True
    self.is_second_state = False
    self.Q = np.zeros((2, X_SCREEN // X_BINSIZE, Y_SCREEN // Y_BINSIZE, 2))

def reset(self):
    self.last_state = None
    self.last_action = None
    self.last_reward = None
    self.is_first_state = True
    self.is_second_state = False
    self.gravity = None

def action_callback(self, state):
    # Do nothing if first state
    if self.is_first_state:
        self.is_first_state = False
        self.is_second_state = True
        self.last_action = 0
        self.last_state = state
        self.gravity = 1
        return self.last_action

    # infer gravity in second state
    if self.is_second_state:
        self.is_second_state = False
        if state["monkey"]["vel"] == -4:
            self.gravity = 1
        else:
            self.gravity = 0

    # Define variables
    alpha_1 = 0.1
    alpha_2 = 0.05
    epsilon = 0.001
    gamma = 0.9

    other_gravity = None
    if self.gravity == 0:
        other_gravity = 1
    else:
        other_gravity = 0

    # 1. Discretize 'state' to get your transformed 'current state' features.
    last_x, last_y = self.discretize_state(self.last_state)
    curr_x, curr_y = self.discretize_state(state)

    # 2. Perform the Q-Learning update using 'current state' and the 'last state'.

    # Updating the inferred gravity's table
    self.Q[self.last_action, last_x, last_y, self.gravity] += alpha_1 * \
        (self.last_reward + gamma * np.amax(self.Q[:, curr_x, curr_y, self.gravity]) \
            - self.Q[self.last_action, last_x, last_y, self.gravity])

    # Updating the table for the gravity that wasn't chosen
    self.Q[self.last_action, last_x, last_y, other_gravity] += alpha_2 * \
        (self.last_reward + gamma * np.amax(self.Q[:, curr_x, curr_y, other_gravity]) \
            - self.Q[self.last_action, last_x, last_y, other_gravity])
    # 3. Choose the next action using an epsilon-greedy policy.
    new_action = None
    if npr.rand() < epsilon:
        if npr.rand() < 0.3:
            new_action = 1
        else:
            new_action = 0
    else:
        new_action = np.argmax(self.Q[:, curr_x, curr_y, self.gravity])

    self.last_action = new_action
    self.last_state = state

    return self.last_action
```

This method was an attempt to implement what I thought was going to be the ideal strategy: to have two different Q-tables for the two different possible gravities (i.e. one Q-table with an extra dimension of length 2). I calculate exactly what the gravity is in the second stage (I realize this is hacky and not what was intended but I figured it would be okay as in the end it did not yield much better results anyways).

When I implemented this strategy I realized that by training the two tables separately I was cutting the training in half for each which resulted in less over all performance than if the model was gravity-blind.

Because of this, I decided to implement two learning rates instead of one. The first learning rate corresponds to the Q-update for the Q-table of the gravity that has been inferred; the gravity that is relevant. The second learning rate corresponds to the other gravity, the one that has not been determined to be used in the current state. I set the first learning rate to 0.9 and played around with different values for the second learning rate, all of which I decided should be less than the first since the model should learn less about a gravity in a game in which that gravity hasn't been chosen than if it had been.

My results were underwhelming, suggesting that my strategy was just not correct. As opposed to setting a second learning rate of 0 (which corresponds to the regular strategy stated in the question), the resulting performance was not much better at all. Below I plot differing results for differing values of the second learning rate.

(Gathered from a recommendation by an Ed post)

$$\alpha_1 = 0.1$$

$$\gamma = 0.9$$

$$\epsilon = 0.001$$

After running 100 iterations 10 times for each differing second learning rate value (α_2) we get:

$$\alpha_2 = 0 : avg.max = 888.0, mean = 55.4, std = 129.97$$

$$\alpha_2 = 0.01 : avg.max = 930.3, mean = 61.37, std = 133.44$$

$$\alpha_2 = 0.05 : avg.max = 872.0, mean = 66.1, std = 138.7$$

$$\alpha_2 = 0.1 : avg.max = 1119.2, mean = 70.8, std = 152.04$$

Name

Rodney Lafuente Mercado

Collaborators and Resources

Whom did you work with, and did you use any resources beyond cs181-textbook and your notes?

Calibration

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