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?

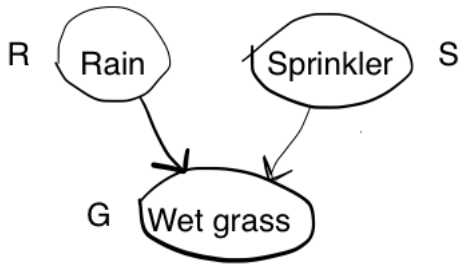
Consider if we introduce a new binary variable, cloudy C, to the the original three binary variables such that the factorization of the joint distribution is now:

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

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. Graphical model:



Yes, R and S are independent by the D-separation rules since G is not observed.

2. Since R and S are independent, the probability is $\Pr(S = 1|R = 1) = \Pr(S = 1) = 0.5$.

3. Using the definition of conditional probability and marginalizing out R,

$$\begin{aligned}
\Pr(S = 1|G = 1) &= \frac{\Pr(S = 1, G = 1)}{\Pr(G = 1)} \\
&= \frac{\Pr(S = 1, G = 1)}{\Pr(S = 0, G = 1) + \Pr(S = 1, G = 1)} \\
&= \frac{\sum_{R=0}^{1} \Pr(R, S = 1, G = 1)}{\sum_{R=0}^{1} \Pr(R, S = 0, G = 1) + \Pr(R, S = 1, G = 1)} \\
&= \frac{0.75 \cdot 0.5 \cdot 0.75 + 0.25 \cdot 0.5 \cdot 1}{0.75 \cdot 0.5 \cdot 0 + 0.25 \cdot 0.5 \cdot 0.75 + 0.75 \cdot 0.5 \cdot 0.75 + 0.25 \cdot 0.5 \cdot 1} \\
&= \frac{0.28125 + 0.125}{0 + 0.09375 + 0.28125 + 0.125} \\
&= \frac{0.40625}{0.5} \\
&= 0.8125.
\end{aligned}$$

4. Again using the definition of conditional probability,

$$\begin{aligned}
\Pr(S = 1|R = 1, G = 1) &= \frac{\Pr(R = 1, S = 1, G = 1)}{\Pr(R = 1, G = 1)} \\
&= \frac{\Pr(R = 1, S = 1, G = 1)}{\Pr(R = 1, S = 0, G = 1) + \Pr(R = 1, S = 1, G = 1)} \\
&= \frac{0.25 \cdot 0.5 \cdot 1}{0.25 \cdot 0.5 \cdot 0.75 + 0.25 \cdot 0.5 \cdot 1} \\
&= \frac{0.125}{0.09375 + 0.125} \\
&= \frac{0.125}{0.21875} \\
&\approx 0.571.
\end{aligned}$$

5. In part 3, the probability of the sprinkler being on increased significantly from its prior probability when we observed that the grass was wet. However, in part 4, when we also observed that it was raining, the probability of the sprinkler being on decreased towards our prior. The rain "explains

away" the likely cause of the grass being wet, which means that the wet grass no longer offers as much information as to whether the sprinkler is on.

6. $$\Pr(R) = \sum_C \Pr(C) \Pr(R|C) \sum_G \sum_S \Pr(S|C) \Pr(G|R,S).$$

7. $$\Pr(R) = \sum_S \sum_G \Pr(G|R,S) \sum_C \Pr(C) \Pr(R|C) \Pr(S|C).$$

8. The complexity of the ordering S, G, C is $O(k^4)$, since to eliminate S, we sum over the k values of S for each possible value of C, G, and R. The complexity of the ordering C, G, S is $O(k^3)$, since to eliminate C, we sum over the k values of C for each possible value of R and S, and similarly do $O(k^3)$ work to eliminate G and $O(k^2)$ work to eliminate S. Therefore the 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:

1. (a) See `T6_P2.ipynb`.

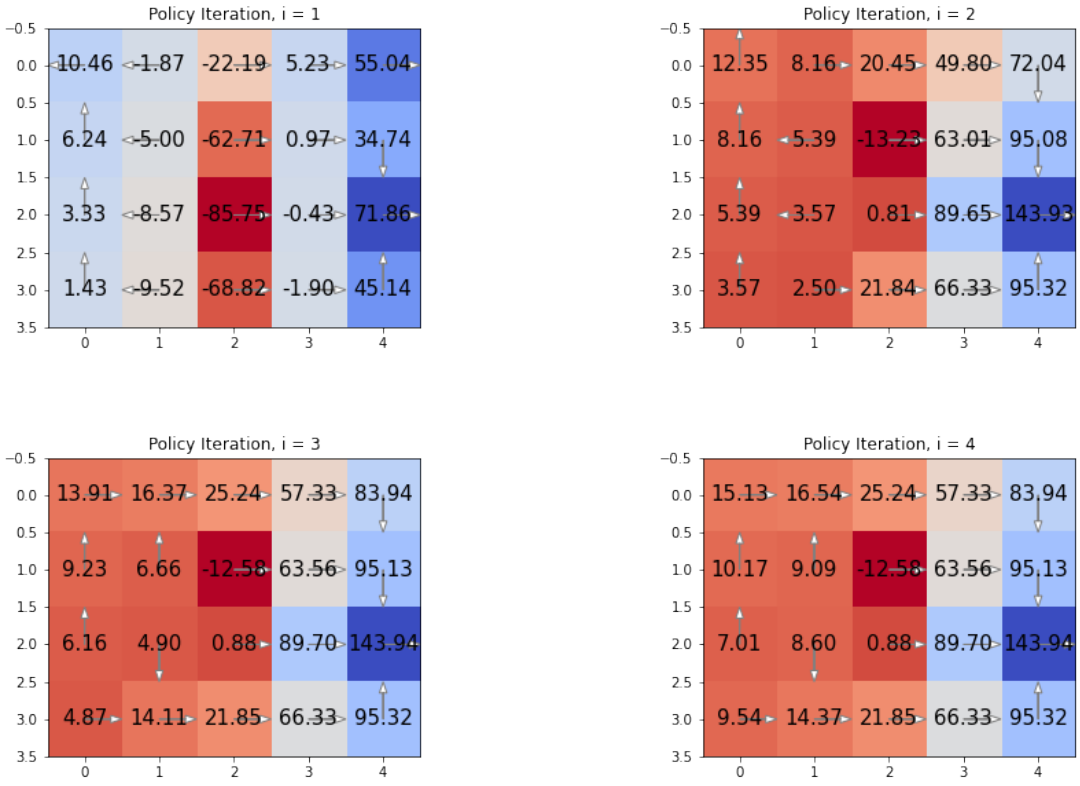


Figure 1: First four policy iterations

(b) See `T6_P2.ipynb`.

(c) See Figure 1.

(d) With `ct = 0.01`, the algorithm converges in 5 iterations. See Figure 2 for the final learned value function and policy. Decreasing `ct` does not affect the number of iterations needed until convergence.

2. (a) See `T6_P2.ipynb`.

(b) See Figure 3.

(c) With `ct = 0.01`, the algorithm converges in 25 iterations. See Figure 4 for the final learned value function and policy. With `ct = 0.001`, the algorithm converges in 31 iterations to the same values and policy. With `ct = 0.0001`, the algorithm converges in 38 iterations to values identical to those in policy iteration and the same policy as in both policy and value iteration.

3. Value iteration takes significantly more iterations to converge than policy iteration does. However, the time per iteration is higher for policy iteration because at each iteration it must solve for the new values in policy evaluation, which involves looping multiple times until convergence. The overall CPU runtimes are given below.

`ct`	Policy Iteration	Value Iteration
`0.01`	`1.95 s`	`1.04 s`
`0.001`	`1.94 s`	`1.26 s`
`0.0001`	`2.19 s`	`1.57 s`

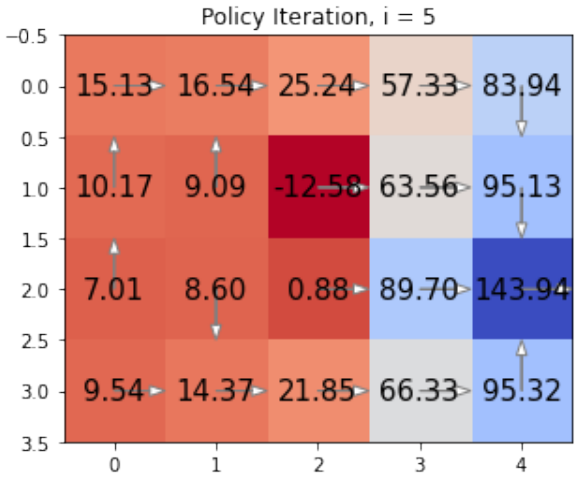


Figure 2: Policy iteration final learned value function and policy

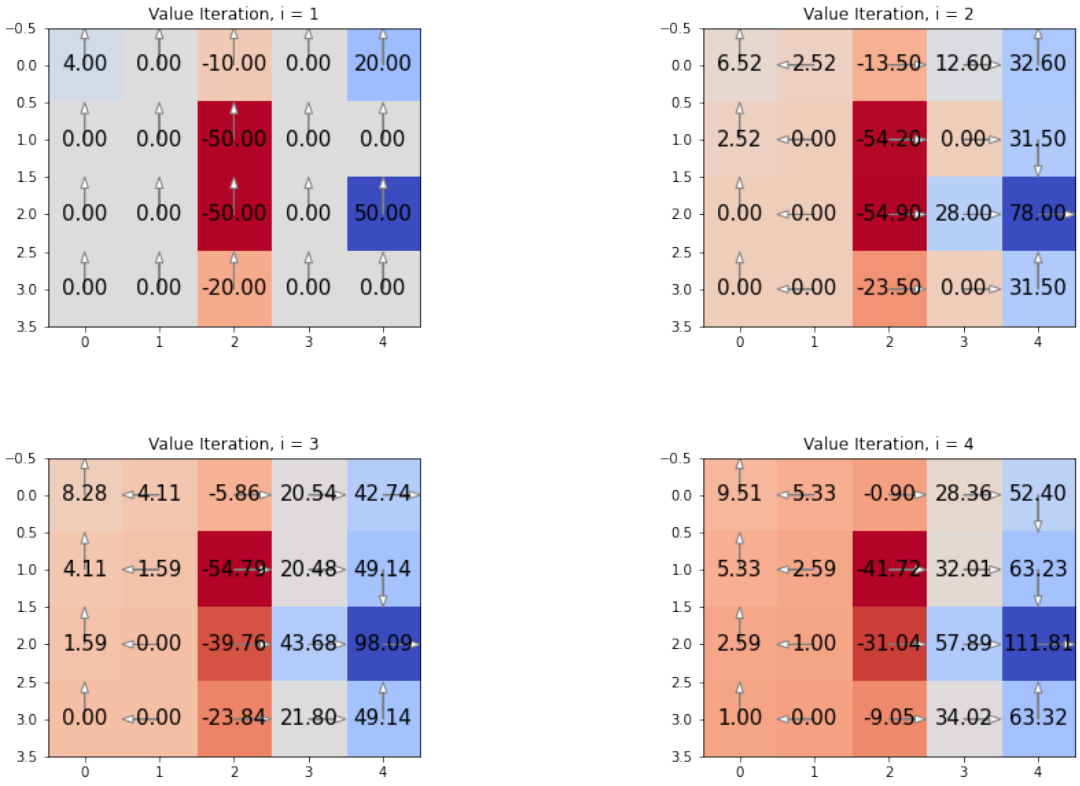


Figure 3: First four value iterations

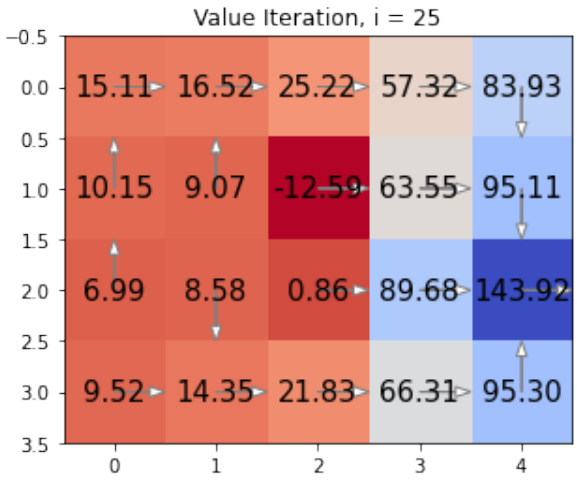


Figure 4: Value iteration final learned value function and policy

This suggests that, at least for this implementation and problem instance, value iteration is faster overall, although its runtime increases faster as the convergence tolerance decreases.

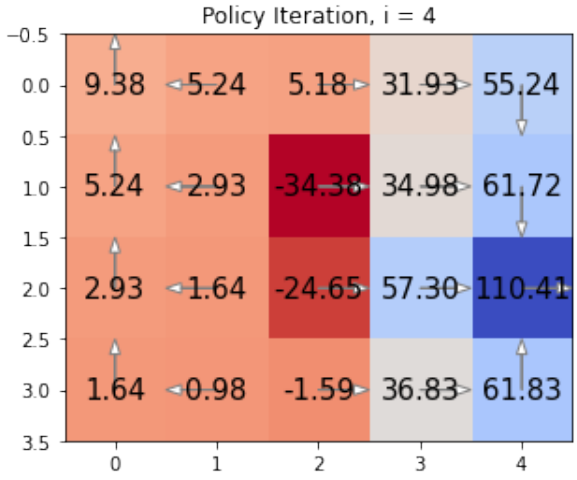
4. See Figure 5. The plots provided are from policy iteration, but the optimal policies determined by both policy and value iteration are identical, and their values only differ very slightly.

   For all values of γ, the optimal policy for states on the right-hand side is to move toward the reward of 50 and stay there. For $\gamma = 0.6$, the optimal policy for states on the left-hand is to move towards the reward of 4 in the upper-left corner and stay there. For $\gamma = 0.7$ and $\gamma = 0.8$, the optimal policy is generally to get the reward of 4 if close to it, and then go around the states with reward -50 in the center to reach the state with reward 50. Finally, for $\gamma = 0.9$, the optimal policy for all states but one is to rush to the right as quickly as possible, going through the states with reward -50 if necessary, to get to the state with a reward of 50.

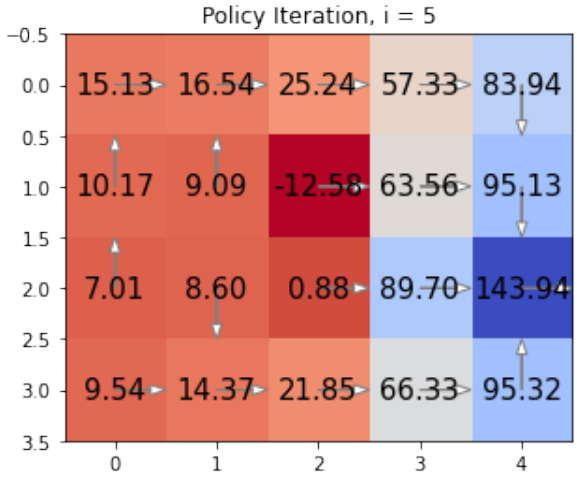
   These varying policies make sense since for smaller values of γ, the algorithms are focused on rewards in the near future, like the 4 in the corner and avoiding the -50s in the center. However, for the larger values of γ, the algorithms care much more about reaching the globally highest reward-state as quickly as possible, since they still have high value for being there even after several moves to reach it.

   Increasing γ increases the number of iterations and overall runtime for both policy and value iteration, because when updating the values of states and actions, the rewards of future states matter more and so must be calculated out farther into the future before converging.

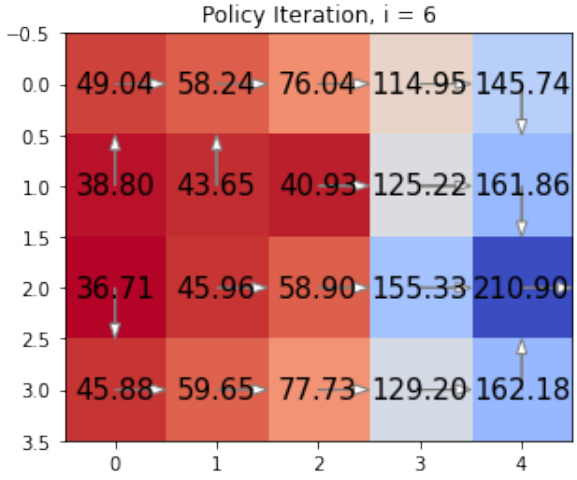
5. The optimal policy for most states on the right side and in the center would still be to go to the reward of 50 for all γ, since that is closest positive reward. For low values of γ, the optimal strategy starting from the left side is to go to the reward of 4, since it is closest and reaching either of the other rewards incurrs a negative reward along the way. For moderately large values of γ, the optimal policy is to traverse around the bottom of Gridworld to get to the reward of 50 fairly quickly, at the expense of a penalty of -20. For large values of γ, the optimal policy is to traverse around the top of Gridworld to reach the reward of 50 while incurring the minimum penalty of -10 along the way, at the expense of taking more time. Note that for no value of γ will the optimal policy pass through the -50 states, since the maximum total reward is now 50, so passing through a -50 would result in a total reward of at most 0.



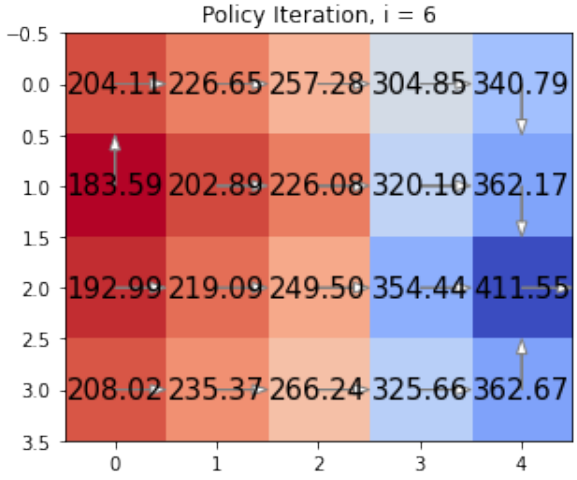
(a) $\gamma = 0.6$



(b) $\gamma = 0.7$



(c) $\gamma = 0.8$



(d) $\gamma = 0.9$

Figure 5: Learned policy for different discount factors γ

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 6a). 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> }}
```

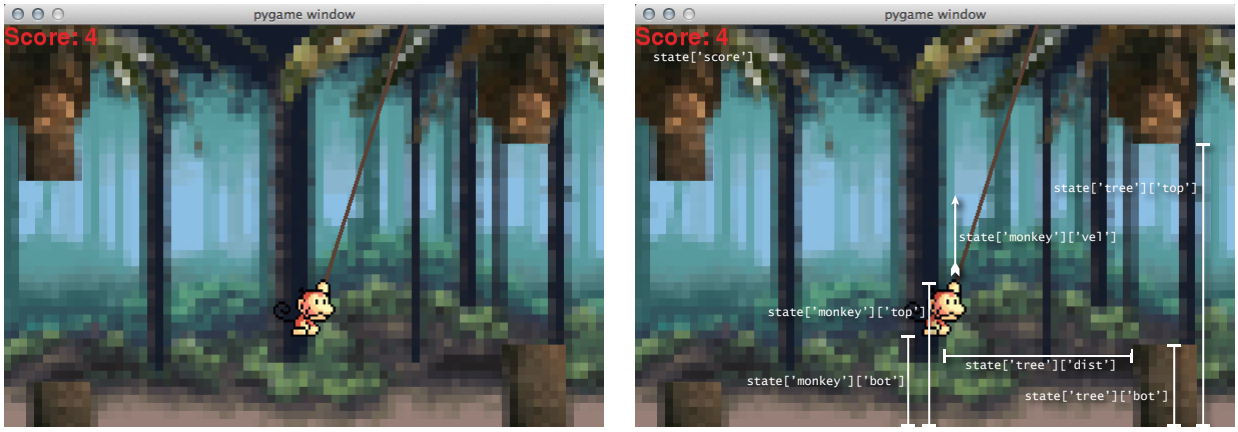
All of the units here (except score) will be in screen pixels. Figure 6b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

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

Solution:

Code:

```
def action_callback(self, state):
  """
  Implement this function to learn things and take actions.
  Return 0 if you don't want to jump and 1 if you do.
  """
  current_state = self.discretize_state(state)

  '''
  # Perform Q-learning update using current state and last state
  if self.last_state is not None:
    td_error = (self.last_reward + self.gamma *
              np.max(self.Q[:, current_state[0], current_state[1]]) -
              self.Q[self.last_action, self.last_state[0], self.last_state[1]])
    self.Q[self.last_action, self.last_state[0], self.last_state[1]] += self.alpha *
                                                td_error
  '''

  # Perform Q-learning update using decaying alpha
  if self.last_state is not None:
    N_tsa = self.N[self.last_action, self.last_state[0], self.last_state[1]]
    alpha_t = 1 / N_tsa
    td_error = (self.last_reward + self.gamma *
              np.max(self.Q[:, current_state[0], current_state[1]]) -
              self.Q[self.last_action, self.last_state[0], self.last_state[1]])
    self.Q[self.last_action, self.last_state[0], self.last_state[1]] += alpha_t * td_error

  '''
  # Choose next action using an epsilon-greedy policy
  if npr.rand() > self.epsilon:
    new_action = np.argmax(self.Q[:, current_state[0], current_state[1]])
  else:
    new_action = int(npr.rand() < 0.5)
  '''

  # Choose next action using decaying epsilon-greedy policy
  N_ts = np.sum(self.N[:, current_state[0], current_state[1]])
  epsilon_t = 1 / N_ts if N_ts > 0 else 1
```

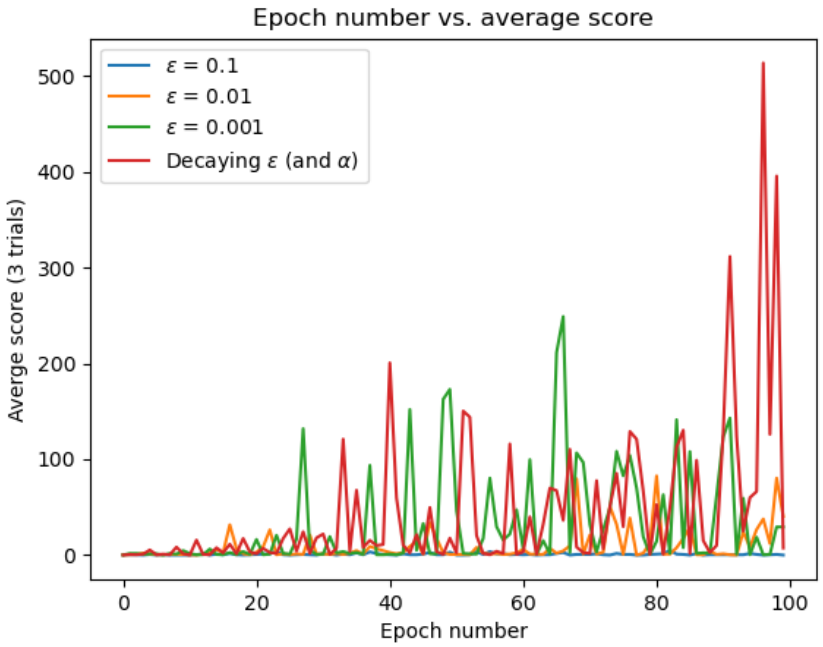
```
if npr.rand() > epsilon_t:
  new_action = np.argmax(self.Q[:, current_state[0], current_state[1]])
else:
  new_action = int(npr.rand() < 0.5)

self.N[new_action, current_state[0], current_state[1]] += 1
self.last_action = new_action
self.last_state = current_state

return new_action
```

Evaluation:

Through trial and error, I found good parameters $\alpha = 0.25, \gamma = 0.9, \epsilon = 0.001$ for the unmodified Q-learning with ϵ-greedy policy. With these parameters, for each trial of 100 epochs, the agent consistently obtained a couple of scores in the 100s, more in the 10s, with the remaining half of the scores in the single digits. On a few trials, the agent scored over 1000 in a game. The most important parameter turned out to be ϵ, which must be quite small or else the scores are all very low. This is because if ϵ is too high, even if an agent finds the optimal Q, it still performs poorly because it often takes random, rather than optimal, actions. This error compounds since the agent needs to consistently take good actions in a game to obtain a high score.



I then implemented decaying ϵ and α to improve performance, by letting

$$\epsilon_t(s) = \frac{1}{N_t(s)}$$

where $N_t(s)$ is the number of times the learner visted state s and

$$\alpha_t(s,a) = \frac{1}{N_t(s,a)}$$

where $N_t(s,a)$ is the number of times the learner took action a in state s. This resulted in more games with scores over 100, usually around 10. The plot above shows average score per epoch for the non-decaying

and decaying parameter agents, demonstrating the importance of a low ϵ and the superiority of decaying parameters. This performance improvement is because with a decaying ϵ and α, the agent explores and learns more early on, then gradually converges to exploiting and learning less, rather than exploring and learning at the same rate throughout. A new issue I encountered is that the agent will sometimes perform very poorly on a trial and not get over 50 on any games. This is likely because an agent will be unlucky early on and explore mostly bad actions, then be stuck with exploiting those actions as the decay occurs.

Name

Alex Encalada-Stuart

Collaborators and Resources

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

No one and no

Calibration

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

15