Homework 5: Markov Decision Processes and Reinforcement Learning

Introduction

In this assignment, you will gain intuition for how MDPs and RL work. For readings, we recommend Chapters 11 and 12 of the CS181 textbook and the pre-lecture materials from April 11th and April 13th.

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 'HW5'**. Remember to assign pages for each question.

Please submit your **LaTeX file and code files to the Gradescope assignment 'HW5 - 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 (Markov Decision Processes, 20 pts)

In this problem, you will be working on building your conceptual understanding of MDPs.

1. For the past few weeks in this course we have seen how *latent variable models* can be fit using the EM algorithm. Recall that in the M-step, we maximize the ELBO with respect to the model parameters given our best guess for the probability distribution over the latent variables. One of the terms in the ELBO is the *complete data likelihood*.

   (a) The complete data likelihood for one latent variable model called pPCA is as follows:

   $$p(Z_{1\cdots N}, Y_{1\cdots N}|\theta) = \prod_{n=1}^{N} p(Y_n, Z_n|\theta) = \prod_{n=1}^{N} p(Y_n|Z_n, \theta)p(Z_n)$$

   Question: Use what you know from the factorization of the complete data likelihood for pPCA to either draw, or describe in words, its associated graphical model.

   (b) The complete data likelihood for a Hidden Markov Model (HMM) is as follows:

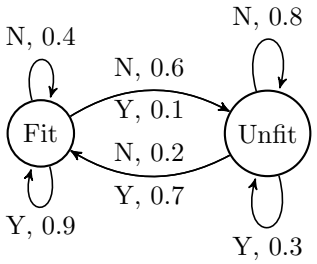
   $$\begin{aligned} p(Z_{1\cdots N}, Y_{1\cdots N}|\theta, \mathcal{T}) &= \prod_{n=2}^{N} p(Y_1, Z_1|\theta) \prod_{n=1}^{N} p(Y_n, Z_n|Z_{n-1}; \theta, \mathcal{T}) \\ &= p(Y_1|Z_1; \theta)p(Z_1) \prod_{n=2}^{N} p(Y_n|Z_n; \theta)p(Z_n|Z_{n-1}, \mathcal{T}) \end{aligned}$$

   Question: How does the graphical model for the HMM differ from that for pPCA? Where do you see the Markov assumption coming into play?

2. In the following example, we translate a medical study of personal fitness into an MDP. We observe the participants' performance on a number of physical activities and divide them into two categories: *fit* and *unfit*. Every week we also ask participants whether they worked out or not. We define our MDP as follows:

   $$\mathcal{S} = \{\text{unfit}, \text{fit}\}$$
   $$\mathcal{A} = \{\text{workout (Y), not workout (N)}\}$$

s_n	a_n	s_{n+1}	$\mathcal{R}(s_n, a_n, s_{n+1})$
fit	Y	fit	0
fit	Y	unfit	-2
fit	N	fit	2
fit	N	unfit	-1
unfit	Y	fit	1
unfit	Y	unfit	-1
unfit	N	fit	10
unfit	N	unfit	-1



(continued on next page...)

Problem 1 (cont.)

2. **Question:**

   (a) What is one design choice we made in this model? What is one pro and one con of this design choice?

   (b) Suggest one modification we could make to this model and describe what additional data (if any) we would need to collect to make this modification. How would this modification affect the rest of the model?

3. Recall that *planning* in an MDP is the process of finding a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the agent's expected reward.

   Question:

   (a) In the MDP setting, why can't $\pi(s_n)$ depend on the *history* – the values of previous states $s_{n-1}, s_{n-2}, ...$?

   (b) Is this assumption reasonable for our personal fitness MDP?

4. Recall that the *return* for a trajectory (a sequence of states and actions) of length N is given by:

$$G = \sum_{n=1}^{N} \gamma^n R_n$$

   where R_n is the reward collected at time n.

   Question: Why do we discount? Often we argue that discounting is needed if we allow infinite trajectories. Show that if $\gamma = 1$ and the trajectory is infinite then G can be undefined.

5. Recall that the value function of an MDP for state s under policy π is defined as follows:

$$V^\pi(s) = \mathbb{E}_\pi[G|Z_0 = s]$$

   where the expectation above is taken over all randomly varying quantities in G.

   Question:

   (a) In our personal fitness MDP what quantities that G depends on are randomly varying? *Hint:* Think about sources of randomness in your reward, transition and policy.

   (b) Suppose the agent's policy π is to exercise if in the `unfit` state and not exercise if in the `fit` state. Calculate $V^\pi(\text{fit})$ and $V^\pi(\text{unfit})$.

6. Recall that the Q-function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ quantifies the value of a policy π starting at state s, taking action a, and *then* following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G|Z_0 = s, A_0 = a]$$

   Question: Assuming that the agent follows the same policy as in the previous question, calculate $Q^\pi(\text{fit}, \text{Y})$, $Q^\pi(\text{fit}, \text{N})$, $Q^\pi(\text{unfit}, \text{Y})$, $Q^\pi(\text{unfit}, \text{N})$.

Solution 1:

1. (a) Below is a drawing of the associated graphical model:

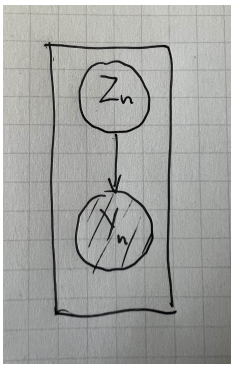


Figure 1: pPCA Graphical Model

(b) Below is the HMM's graphical model:

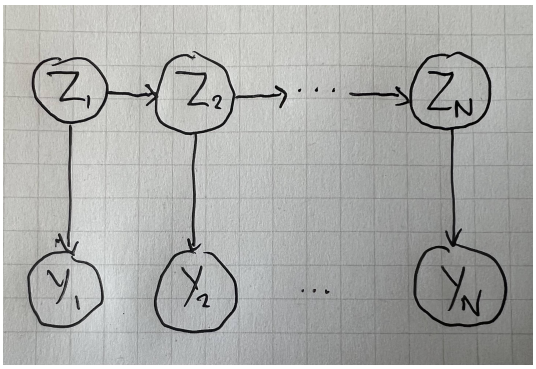


Figure 2: HMM Graphical Model

Complete data likelihood of HMM vs pPCA is very similar, except for HMM, Z_n is conditioned on Z_{n-1} and $\mathcal{T}$. This is shown by the directed edges in the diagram from Z_{n-1} to Z_n.

2. (a) One design choice is the state space, which is a binary of fit, unfit. This lends itself to a very simple model that is easy to visually represent. However, it also doesn't accurately capture the complexity of physical health, which is not typically viewed as a binary.

(b) We could strike a balance between simplicity and complexity by perhaps adding more actions (such as "Eating Healthy") and keeping the state binary. We would have to modify our model to specify which state our actions lead to (perhaps in combination.) In this case, we might need to measure diets and 'health' levels.

3. (a) According to the Markov assumption, $\pi(s_n)$ only depends on the current state s_n, and so it is independent of the history.

(b) This assumption is not reasonable in the context of personal fitness, since a person's health history is certainly relevant, and people with different health histories might need to take different actions. That being said, our model has a binary state, and so it would be reasonable if we are trying to maintain simplicity.

4. If $\gamma = 1$, then with infinite trajectory, as $N \to \infty$, $\sum_{n=1}^{N} \gamma^n R_n \to \infty$. Since G does not converge in this case, it is undefined.
   Being able to accumulate infinite rewards regardless of policy makes it impossible to distinguish between any policy. Adding a discount allows our model to prioritize sooner reward rather than later reward, which intuitively makes sense, since it is usually preferable to get a reward earlier rather than later.

5. (a) In our personal fitness MDP, G depends on γ and R. However, γ is fixed, and so we look at R: it depends on s_n, a_n, s_{n+1}, i.e. the transition matrix and the policy. Our transition function is random, and it might be possible for our policy to have some randomness.

   (b) Let us define fit to be S_0 and unfit to be S_1. Thus, we have the following equation:

Problem 2 (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 3a). 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> }}
```

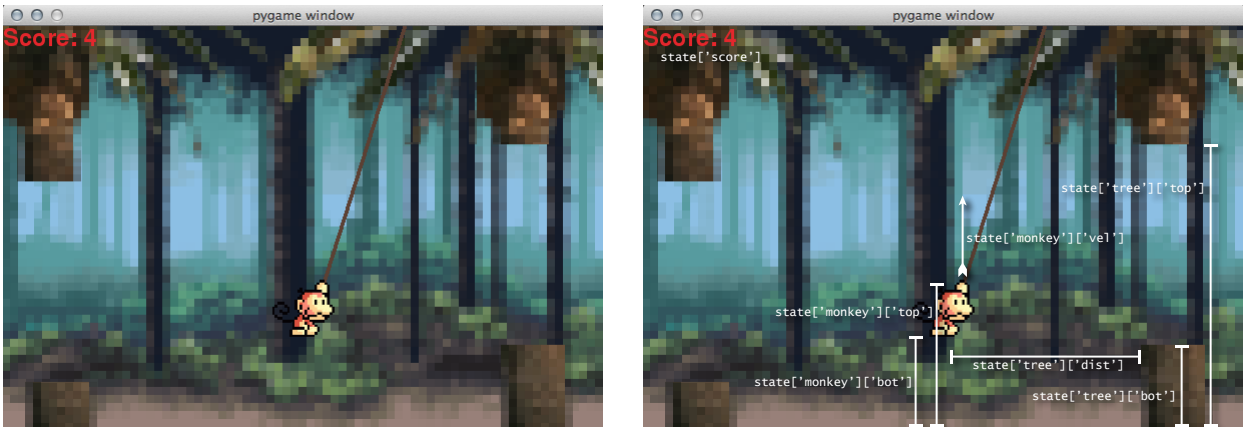
All of the units here (except score) will be in screen pixels. Figure 3b shows these graphically. Note that since the state space is very large (effectively continuous), the monkey's relative position needs to be discretized into bins. The pre-defined function `discretize_state` does this for you.

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

Solution 2:

Approach:

I implemented Q-learning with an ϵ-greedy policy, using the following parameters:

- $\epsilon = 0.001$
- $\alpha = 0.1$
- $\gamma = 0.1$

The algorithm uses a Q-table that stores all of the expected rewards for all the actions, at all states. It is updated iteratively with each epoch, and is used to estimate the highest expected reward. At each iteration, the monkey has an probability ϵ of randomly picking an action (exploring), and probability $1 - \epsilon$ of taking the action with the highest expected reward. Also implemented is ϵ decay function that decays the Learner's ϵ at every iteration using a minimum epsilon value, the starting value of epsilon and the total number of moves at a given iteration.

Evaluation: On several test runs, the monkey consistently scored above the benchmark of 50 within the first 100 epochs. See below:

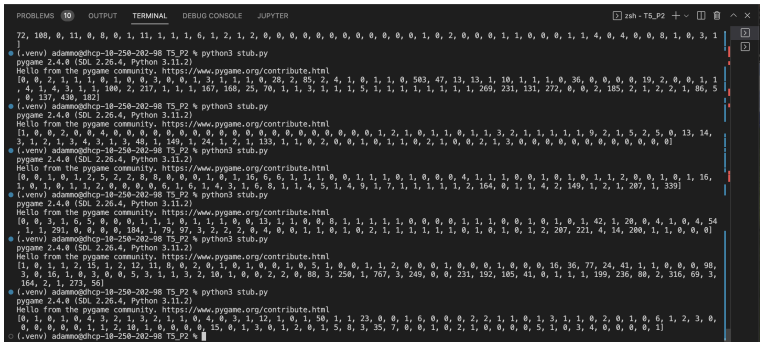


Figure 4: Performance of Q-learning algorithm

My initial parameter choices were informed in part by intuition, and in part by a grid search that I performed on a parameter space that I picked arbitrarily. For γ, a higher value was picked because it feels natural to have a higher discount value–a greedy algorithm would prioritize surviving at every tree, which seems like a good strategy to survive as long as possible. For ϵ, we want the monkey to explore less and trust its learned knowledge more, and so it has a low value, and in addition it decays over time (the more the monkey learns, the more it relies on its learned knowledge.)

See figure 5 for a plot of the score of several runs of the game, each with a different parameter configuration.

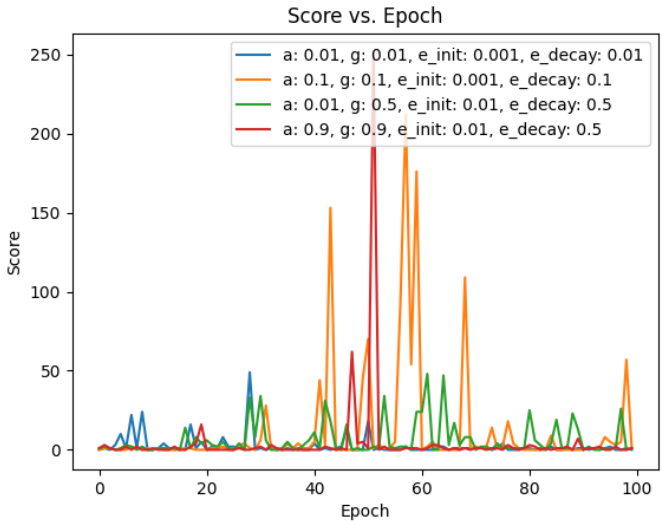


Figure 5: Results of monkey's performance on various hyperparam configurations

Code:

```
import numpy as np
import numpy.random as npr
import pygame as pg

X_BINSIZE = 200
Y_BINSIZE = 100
X_SCREEN = 1400
Y_SCREEN = 900


class Learner(object):
    """
    This agent jumps randomly.
    """

    def __init__(self, alpha=0.1, gamma=0.1, initial_epsilon=0.01, eps_decay_rate=0.1):
        self.last_state = None
        self.last_action = None
        self.last_reward = None

        # Initialize our hyperparameters
        self.alpha = alpha
        self.gamma = gamma

        self.min_epsilon = 0.001
        self.initial_epsilon = initial_epsilon
```

```
        self.epsilon = self.initial_epsilon
        self.eps_decay_rate = eps_decay_rate

        self.total_moves = 0


        # We initialize our Q-value grid that has an entry for each action and state.
        # (action, rel_x, rel_y)
        self.Q = np.zeros((2, X_SCREEN // X_BINSIZE, Y_SCREEN // Y_BINSIZE))

    def reset(self):
        self.last_state = None
        self.last_action = None
        self.last_reward = None

    def discretize_state(self, state):
        """
        Discretize the position space to produce binned features.
        rel_x = the binned relative horizontal distance between the monkey and the tree
        rel_y = the binned relative vertical distance between the monkey and the tree
        """

        rel_x = int((state["tree"]["dist"]) // X_BINSIZE)
        rel_y = int((state["tree"]["top"] - state["monkey"]["top"]) // Y_BINSIZE)
        return (rel_x, rel_y)

    def decay_epsilon(self):
        """
        Decay epsilon based on total number of moves, initial epsilon and current epsilon
        """
        self.epsilon = self.min_epsilon + (self.initial_epsilon - self.min_epsilon) * np.exp
    (-self.eps_decay_rate * self.total_moves)

    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.
        """

        # Discretize the state
        current_state = self.discretize_state(state) # (rel_x, rel_y)

        # Update Q
        if self.last_state is not None:
            # Get the Q values for the last state
            last_state = self.last_state
            last_action = self.last_action
            last_reward = self.last_reward

            # Get the Q value for the last state and action
            last_Q = self.Q[last_action][last_state]

            # Get the Q value for the best action in the current state
            current_Q = np.max(self.Q[:, current_state[0], current_state[1]])

            # Update the Q value for the last state and action
            self.Q[last_action][last_state] = last_Q + self.alpha * (last_reward + self.
    gamma * current_Q - last_Q)

        # Decay epsilon
        self.total_moves += 1
        self.decay_epsilon()

        # Choose the next action using an epsilon-greedy policy
        if npr.rand() < self.epsilon:
```

```
            # Choose a random action
            new_action = npr.randint(2)

        else:
            # Choose the best action
            new_action = np.argmax(self.Q[:, current_state[0], current_state[1]])

        new_state = current_state

        self.last_action = new_action
        self.last_state = new_state

        return self.last_action

    def reward_callback(self, reward):
        """This gets called so you can see what reward you get."""

        self.last_reward = reward
```

Problem 3 (Impact Question: Assessing the energy consumption of data centers and understanding a RL approach to optimize it, 3.5 pts)

Every computation consumes energy. Over the past decades large scale data centers have been built which are significant energy consumers. Consequently, every computation is costly.

1. **Energy consumption**: How much energy do data centers consume globally per year? And what's their share in global energy consumption? (1 point)
2. **Consumption optimization with RL**: How would you formulate a RL model which enables the optimization of the load of a data center? Describe the main elements of your model (state space, action space, reward). (1.5 points)
   Hint: This RL Paper provides background information for this.
3. **Alternatives**: List one alternative idea on how to reduce the climate impact of data centers. (1 point)

Name

Adam Mohamed

Collaborators and Resources

I used the following resources: Q-Learning: A Complete Example in Python Multiple line graphs matplotlib

and had some help from Evan Jiang

Calibration

This homework took about 6 hours