CSCI 699: Robot Learning
Problem Set #2: Due Sun, Oct 8, 11:59PM

Introduction

The coding portion of the homework assignment will be completed using Google Colab. Starter code for this problem set can be downloaded from https://github.com/USC-Lira/CSCI699_RobotLearning_HW2.git. The notebook should contain the code for installing all the necessary Python dependencies.

Submission instructions:

You will submit your homework to Gradescope. Your submission will consist of (1) a single pdf with your answers for the short answer questions (✍) and (2) a zip folder containing the Colab notebooks for the programming questions (💻).

Your answers to the written portion must be typeset with a word processor or LaTeX.

Problem 1 [20 points]

1a) ✍️ [5 points] Prove the following inequality involving two real-valued functions f and h:

$$|\max_z f(z) - \max_z h(z)| \leq \max_z |f(z) - h(z)|$$

1b) ✍️ [5 points] Using the property in 1a, prove the following inequality:

$$||TV - T\bar{V}||_\infty \leq \alpha ||V - \bar{V}||_\infty$$

where T is the Bellman operator used to update the value function, V is the state-value function, and $\alpha \in [0, 1)$.

$$TV(s) = \max_{a \in \mathcal{A}} \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right)$$

Here is a breakdown of each component of the Bellman operator:

- $TV(s)$ is the value of the Bellman operator applied to state s.
- $V(s)$ is the value function at state s.
- $\mathcal{A}$ is the set of all possible actions.
- $\mathcal{S}$ is the discrete state space.
- $R(s,a)$ is the immediate reward received when taking action a in state s.
- γ is the discount factor.
- $P(s'|s,a)$ is the probability of transitioning from state s to state s' upon taking action a.

1c) ✍️ Policy Iteration [10 points]
Consider a simple Markov Decision Process (MDP) with three discrete states and two actions defined below:

$$\mathcal{S} = \{A, B, C\}$$
$$\mathcal{A} = \{X, Y\}$$
$$R = \begin{bmatrix} 5 & -2 \\ 1 & 2 \\ -1 & 0 \end{bmatrix}$$
$$P_X = \begin{bmatrix} 0.3 & 0.6 & 0.1 \\ 0.7 & 0.2 & 0.1 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}$$
$$P_Y = \begin{bmatrix} 0.2 & 0.4 & 0.4 \\ 0.1 & 0.6 & 0.3 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

where $R[i,j]$ represents the reward for taking action j in state i, e.g. $R[1,1] = 5$ represents the reward for taking action X in state A. $P_X[i,j]$ represents the probability of transitioning from the state in row i to the next state in column j when action X is taken. For example, $P_X[1,2] = 0.6$ is the probability of transitioning from state A to state B when action X is taken. P_Y is similarly defined for action Y.

Start with the policy $\pi(s) = X$ for all $s \in \mathcal{S}$. Perform one step of policy evaluation; calculate the state-value function $V(s)$ for the initial policy that prescribes the action X for all states.

Iteratively perform policy improvement to determine the optimal policy π^* and policy evaluation. Clearly show your work for **two iterations** of the algorithm, including the Bellman equations used for the policy evaluation and the Q-values for policy improvement.

Problem 2: Actor Critic Methods[1] [40 points]

In class, we surveyed a few different reinforcement learning algorithms. Many reinforcement learning algorithms approximate the value function using a finite set of parameters θ, i.e., $V(s) \approx V^{\pi}(s)$ or $Q_{\theta}(s,a) \approx Q^{\pi}(s)$. For large state and action spaces, an exciting new approach involves approximating these functions using neural networks.

In this problem, we will first dive deep into so-called policy gradient methods, that directly learn policies without explicitly learning value functions. Then, we will address the high-variance issue of policy gradient methods by reintroducing the value function in the learning process as part of the 'actor-critic' method. By the end of this problem, you will implement the critic component of an actor-critic algorithm, and see how your actor-critic solves a classic cart-pole (aka inverted pendulum) problem.

In case you're not familiar with the cart-pole problem, your objective is to move the cart on the bottom just right to make the pole stand upright without going off screen. By the end of this problem, you will have a controller that successfully solves this problem.

Run the first cell in the Colab notebook (`p2_a2c.ipynb`) to visualize a simple rollout in the cart-pole environment.

Policy Gradient Methods

Let's define a parameterized policy $\pi_{\theta}(a_t|s_t)$ as a probability distribution over actions given a state of the system. Now recall that the objective of reinforcement learning is to find parameters (θ) of that policy such that they maximize an objective function J. More specifically, for this question, we will define our objective function as the expected reward over a finite horizon T given a Markov chain episode τ.

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}}(\tau)[r(\tau)], \tag{1}$$

where

$$r(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t), \tag{2}$$

and

$$\pi_{\theta}(\tau) = p(s_0)\pi_{\theta}(a_0|s_0) \prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1})\pi_{\theta}(a_t|s_t) \tag{3}$$

Policy gradient methods are a popular class of reinforcement learning methods that attempt to learn such policies by directly taking the gradient of this objective.

We can show

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \mathbb{E}_{\tau \sim \pi_{\theta}(\tau)}[\nabla \log \pi_{\theta}(\tau) r(\tau)] && (4)\\
&\approx \frac{1}{N} \sum_{i=1}^{N} \nabla \log \pi_{\theta}(\tau_i) r(\tau_i) && (5)\\
&= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it}|s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). && (6)
\end{aligned}$$

The core of policy gradient is to maximize performance, so we update our parameters using gradient ascent, i.e.,

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla_{\theta} J(\theta_t)}. \tag{7}$$

Note that $\widehat{\nabla_{\theta} J(\theta_t)}$ is a stochastic estimate of our gradient. All methods that follow this general idea are called policy gradient methods. By adding in the additional assumption of 'causality' that roughly says that the action at time t' cannot affect the reward at time t when $t < t'$ (the Markov property from the underlying MDP), we can improve our estimate of the gradient by modifying it to

$$\nabla_{\theta} J(\theta_t) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'}) \right) \right), \tag{8}$$

(note the change in the starting index of the sum of rewards).

[1] This question is based on a question from Stanford CS237B.

Variance Reduction

In practice, the above estimate of the objective's gradient can be quite noisy, i.e. have a high variance. We will therefore make two modifications to the estimate to reduce this variance. These changes have more principled explanations as well, but those explanations are beyond the scope of this class. The first modification we will make is to add a discount factor $\gamma < 1$ to our estimate of the 'reward-to-go'

$$\nabla_\theta J(\theta_t) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_{it}|s_{it})\left(\sum_{t'=t}^{T-1}\gamma^{t'-t}r(s_{it'},a_{it'})\right)\right), \tag{9}$$

Next, we will add a 'baseline' reward estimate

$$\nabla_\theta J(\theta_t) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_{it}|s_{it})\left(\left(\sum_{t'=t}^{T-1}\gamma^{t'-t}r(s_{it'},a_{it'})\right) - V_{\pi_\theta,\phi}(s_{it})\right)\right), \tag{10}$$

The baseline we added is referred to as an approximation of the 'reward-to-go' function, or our best estimate of V^π parameterized by ϕ.

$$V_{\pi_\theta,\phi}(s_{it}) \approx \sum_{t'=t}^{T-1}\mathbb{E}[r(s_{t'},a_{t'})|s_t]. \tag{11}$$

(i) ✍ **[5 points]** We skipped a few steps between Equation (1) and Equation (4). Show:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla \log \pi_\theta(\tau) r(\tau)].$$

(ii) ✍ **[7 points]** Show that the expression on the right-hand side of Equation (8) is equivalent to that of Equation (6).
Hint: Start from Equation (6). Write the outer-most summation (and $1/N$) as an expectation (as in Equation (4)). For the remaining two summations, move the second one inside the first one with the new time index t'. Expand it to two separate summations over t': one will go from 0 to $t-1$, and the other from t to $T-1$. Finally, you will use Markov property for the former (the one that goes from 0 to $t-1$) and prove that term is equal to 0.

(iii) ✍ **[3 points]** Justify why Equation (8) works. In particular, comment on the direction/magnitude of updates in Equation (8). We are only looking for some discussion/justification here – you already did the formal proof in part (ii).

(iv) ✍ **[5 points]** Given the intuition developed in part (iii), explain why policy gradient is susceptible to high variance.

(v) ✍ **[5 points]** When we added the discount factor γ to our gradient estimate in Equation (9), we made sure to specify this one must be smaller than 1 (i.e. $\gamma < 1$). Why is that? For a given time t on episode i, think about how γ scales the different rewards for $t' > t$.

Actor-Critic methods

Actor-critic methods are a sub-class of policy gradient methods that split the policy gradient task into two function approximation tasks. One of them is the learning of a good policy π_θ, referred to as the 'actor', and the other is the learning of the reward-to-go function $V_{\pi_\theta,\phi}(s)$, referred to as the 'critic'. Imagine you're learning to add two numbers a and b with a TA. In the beginning, you don't have any knowledge about this problem, so you solve stochastically. For your first problem ('episode'), say $a = 1, b = 1$, your action could be '−12'. A good TA would give you a negative reward to this action, and you would adjust your policy on how to answer this question accordingly. After a few trials, you perform a good action (in this case '2') to gain a positive reward from your TA. You would adjust your policies to learn how to perform addition, and then receive a new problem (say, $a = 2, b = 3$). Your learning episode continues until your learning is completed. In this example, you're an actor who is consistently adjusting your best guess of the optimal policy (an action a that will maximize your reward r given state s), and the TA is the critic. In real world, however, we don't have a TA, or knowledge of true optimal values, so we train both the actor and critic in parallel.

There are lots of different ways of accomplishing either tasks leading to the multiple variants of actor-critic methods. Here, we will ask you to implement an algorithm that implements the critic part of an actor-critic method using the temporal difference error. In your implementation, you're free to explore, but please keep Actor intact to ensure your Homework 4 code runs without any issue.

Taking our estimate of $\nabla_\theta J$, we can rewrite the estimate of the reward-to-go as

$$\left(\sum_{t'=t}^{T-1}\gamma^{t'-t}r(s_{it'},a_{it'})\right) - V_{\pi_\theta,\phi}(s_t) \approx r(s_t,a_t) + \gamma V_{\pi_\theta,\phi}(s_{t+1}) - V_{\pi_\theta,\phi}(s_t). \tag{12}$$

The quantity on the right is referred to as the temporal difference error or TD error

$$TD_{error} := r(s_t,a_t) + \gamma V_{\pi_\theta,\phi}(s_{t+1}) - V_{\pi_\theta,\phi}(s_t) \tag{13}$$

Roughly speaking, the Bellman equation says that the TD error of a valid 'reward-to-go' function must equal zero. It is therefore possible to train a function approximator by minimizing the TD error. This can be done in various ways, but we choose to generate 'labels'

$$y_t = r(s_t, a_t) + \gamma V_{\pi_\theta,\phi}(s_{t+1}) \tag{14}$$

and minimize the squared error of those predictions

$$\min_{\phi} \sum_{i,t} (V_{\pi_\theta,\phi}(s_{it}) - y_{it})^2. \tag{15}$$

Equipped with our TD error, we can now train our two function approximators quite efficiently. Let's summarize. Actor-critic is a type of policy gradient method. Specifically, we run two neural networks in parallel to learn V^π and π_θ simultaneously. The actor is a neural network that learns to minimize J. The critic is a neural network that learns ϕ to approximate V_{π_θ} by minimizing the squared TD error. Now we're ready to look at the code.

(i) ✍️ **[5 points]** Take a look at the `Actor` class in `p2_a2c.ipynb`. How many layers are there in the Actor neural network? What type of activation functions were used?

In `p2_a2c.ipynb`, have a look at the function `train_actor_critic`. The first thing it does is initialize an environment using OpenAI's gym package13 that you saw in the beginning of this problem. The cart-pole environment has a state space as well as an action space. The cart-pole's state space is a four dimensional vector corresponding to the cart's position (x), the cart's velocity ($\dot{x}$), the pole's angle (θ) and the pole's angular velocity ($\dot{\theta}$), in that order.

$$[x, \dot{x}, \theta, \dot{\theta}]$$

The possible actions that can be applied to the environment are to push the cart towards the left (action 0), or push it towards the right (action 1). After initializing the actor and the critic, the function enters a training loop. The training loop consists of running a number of episodes, where an episode is a sequence of 200 time steps (the gym environment defines this maximum episode length). A transition happens every time env.step is called. As you can see, this function returns the next state but also the reward for that transition. The cart-pole environment is setup to simply return a reward of 1 for every step it takes until termination. Termination happens when one of the following happens:

- Pole angle is $\pm 12°$
- Cart position is more than $\pm 2.4m$
- Episode length is greater than 200

The training loop adds each transition (state, action, reward, next state) to a replay buffer (a simple array containing recently observed transitions).

Next is the training of the function approximators. First, the critic evaluates the temporal difference error, and uses it to improve its current approximation of the value function. This is essentially what we are asking you to implement. Then, the evaluated temporal difference error is passed on to the actor, that uses it to improve the current policy. The reward accumulated over each episode is printed in your terminal as the training progresses.

This whole process repeats until the maximum number of episodes is reached. The actor is then run for a few 'test' episodes that will also render for you.

(i) 💻 **[5 points]** In the Colab notebook, finish implementing the `Critic` class. You will need to create a neural network that takes as input the state and predicts the state-value. You can look at how the `Actor` was implemented for a template. The `forward` pass should take a batch of state, rewards and next states and return a tensor containing the estimated TD error for each sample in the batch.

(ii) ✍️ **[5 points]** Time to run! Test that your `Critic` network is working for the classic cart-pole problem. Run the final cell in the notebook to see your `Actor-Critic` neural network in working. Include the plot of the rewards accumulated over each training episode in your write-up.

If your actor-critic neural network isn't giving you the maximum score of 200, don't stress! We will grade based on how often your actor gets a high score (you should be able to get at least above 100 somewhat con sistently) and the correctness of your implementation. Also keep in mind actor-critic algorithms sometimes require a certain amount of tuning to work well in practice.

Problem 3: Write your own MDP [40 points]

In this problem, you are tasked with designing your own custom Gym environment and solving it using any Reinforcement Learning (RL) algorithm of your choice. We provide two problem scenarios that you can select from to solve using RL.

Scenario A:

Create an $N \times N$ Gridworld environment. There are three entities in this environment: the agent, an apple and a plate. The agent is tasked with cutting the apple and putting the slices of apple onto the plate. You may assume that the agent spawns in the same location at the beginning of each episode, but the object locations will be randomized. The agent is rewarded only if they complete the full task.

Scenario B:

Create an $N \times N$ Gridworld environment. The agent spawns at the bottom-left corner of the grid at every episode. There is a goal cell that is randomly initialized at every episode and fixed throughout the episode. The agent is given a positive reward when it reaches the goal state. The environment is partially observable, so the agent will only have access to its current location. Additionally, the agent will reset to the initial state after a fixed number of steps within the episode. For example, if the total length of the rollout is 60 steps, every 15 steps the agent will reset back to the bottom-left corner.

Design the Gym environment [15 points]

1. ✍️ First, you should create a clear description of your Gym environment considering the following aspects:
   - The state space: What are the states your agent can be in? How do you represent the agent's state?
   - The action space: What actions can the agent take?
   - Reward function: How are rewards calculated in your environment?
   - Termination condition: When does the episode end?
2. 💻 Using Python and any libraries of your choice, create a custom Gym environment following one of the scenarios described above. Ensure that it follows the standard Gym interfaces and implements the functions in the abstraction: `reset`, `step`, and other relevant methods. See the OpenAI gym documentation for more details about the API: https://gymnasium.farama.org/

Training your agent [25 points]

1. ✍️ **[1 point]** Choose any RL algorithm that you are familiar with or interested in exploring. It could be Q-Learning, Deep Q-Networks (DQN), Policy gradient methods, or any other RL algorithm you prefer. Report which algorithm you choose.
2. ✍️ / 💻 **[14 points]** Implement the RL algorithm to train an agent within your custom Gym environment. Make sure training uses a sparse binary reward only given to the agent when the task is completed (full task in Scenario A, reaching a goal in Scenario B). Describe the implementation details (e.g., hyperparameters, any design/architecture choice you made). Feel free to use any deep learning library for your implementation (e.g. Tensorflow, PyTorch, JAX, etc). You should either implement the RL algorithm from scratch or if you choose to use open-source libraries (e.g., stable-baselines) you should include in the writeup a description of how the algorithm works.
3. 💻 **[5 points]** Next, incorporate some reward shaping (subtask rewards, step penalty, etc.) and retrain the agent and show that this helped accelerate the learning or solve the task.
4. ✍️ **[5 points]** Evaluate the trained agent and provide relevant plots for the RL training metrics and task success rate curves.

Here are some additional resources to help you with the implementation:

- PyTorch tutorial on Deep Q-Networks
- OpenAI SpinningUp documentation
- Gym-minigrid: a lightweight gridworld environment

Submit your code implementation as Python files or a Colab notebook. Attach the training / evaluation figures into the writeup.