EECS/ME/BioE 106B Homework 6: Optimal Control & RL

Spring 2023

Note: This assignment has a programming component in a Colab Notebook.

Problem 1: The Linear Quadratic Regulator

Imagine we have a discrete time system with state x_k and input u_k of the form:

$$x_{k+1} = f(x_k, u_k) \tag{1}$$

Dynamic programming is a technique that helps us find an optimal sequence of inputs:

$$u_0, u_1, ..., u_{N-1} \tag{2}$$

That allow our system to behave in an optimal manner for N timesteps. Let's discuss how the process of dynamic programming works. Consider a cost function of the form:

$$J = L_f(x_N) + \sum_{k=0}^{N-1} L(x_k, u_k) \tag{3}$$

This cost function, where $L_f(x_N)$ is the *terminal cost* and $L(x_k, u_k)$ is the *stage cost*, tells us about the total cost associated with all N timesteps and all $N-1$ inputs. Now, let's imagine that we've already taken a few actions, and are now at step i. The *optimal cost to go* from step i is defined to be the *optimal* cost remaining from step i to the final step N. This is computed:

$$J_i^o = \min_{u_i,...,u_{N-1}} [L_f(x_N) + \sum_{k=i}^{N-1} L(x_k, u_k)] \tag{4}$$

This tells us the minimum remaining cost. The recursive *Bellman equation* allows us to solve for the optimal control input u_i at step i using the optimal cost to go J_i^o. It tells us:

$$J_i^o = \min_{u_i} [L(x_i, u_i) + J_{i+1}^o(x_{i+1})] \tag{5}$$

Where J_{i+1}^o is the optimal cost to go starting from the next state, x_{i+1}. By applying the Bellman equation recursively, starting from $J_N^o = L_f(x_N)$ and working our way backwards, we can identify the entire optimal sequence of inputs $u_0, ..., u_{N-1}$. Let's apply this procedure to solve a famous problem in optimal control. Consider the linear discrete time system:

$$x_{k+1} = Ax_k + Bu_k \tag{6}$$

Suppose we'd like to stabilize the state vector of this system to the origin in some *optimal* way. One way of achieving this is by finding an input u_k that minimizes the cost function:

$$J = x_N^T Q_f x_N + \sum_{k=0}^{N-1} (x_k^T Q x_k + u_k^T R u_k) \tag{7}$$

Where $Q_f, Q \succeq 0, R \succ 0$ are positive definite or semidefinite matrices. By minimizing this cost, we'll find an expression for a *small* control input u_k that brings us close to $x_k = 0$, our desired state! We perform this optimization over a horizon of N steps. Let's apply the principle of dynamic programming to find the optimal input u_i for an arbitrary time step i in the horizon. The solution to this problem gives us an optimal controller known as a *linear quadratic regulator*.

Questions

In this question, we'll consider the system proposed above. Our system has the linear discrete time dynamics $x_{k+1} = Ax_k + Bu_k$ and a cost function:

$$J = x_N^T Q_f x_N + \sum_{k=0}^{N-1} (x_k^T Q x_k + u_k^T R u_k) \tag{8}$$

1. We define the *optimal cost to go* to be the remaining optimal cost from an arbitrary step i in the horizon to our final step. Provide a brief worded explanation as to why the optimal cost to go from step i to step N may be expressed:

   $$J_i^o = \min_{u_i,\ldots,u_{N-1}} [x_N^T Q_f x_N + \sum_{k=i}^{N-1} (x_k^T Q x_k + u_k^T R u_k)] \tag{9}$$

   Then, identify the terminal cost $L_f(x_N)$ and the stage cost $L(x_k, u_k)$ in the cost function.

2. Using the optimal cost to go formula from above, we know that the optimal cost to go at state N, J_N^o, is computed:

   $$J_N^o = x_N^T Q_f x_N \tag{10}$$

   By applying the Bellman equation, we can prove that J_k^o, the optimal cost to go from an arbitrary step k, is computed with an expression of a similar form:[1]

   $$J_k^o = x_k^T P_k x_k \tag{11}$$

   Where $P_k \succeq 0$ is a positive semidefinite matrix that depends on k. Apply this formula for the optimal cost to go to the Bellman equation:

   $$J_i^o = \min_{u_i} [L(x_i, u_i) + J_{i+1}^o(x_{i+1})] \tag{12}$$

   To show that J_i^o may be calculated:

   $$J_i^o = \min_{u_i} [x_i^T Q x_i + (Ax_i + Bu_i)^T P_{i+1} (Ax_i + Bu_i)] \tag{13}$$

3. Using this formula for optimal cost to go, show that the optimal input u_i is:

   $$u_i = -(R + B^T P_{i+1} B)^{-1} B^T P_{i+1} A x_i \tag{14}$$

   This input forms the famous *linear quadratic regulator (LQR)* controller! Notice how the controller is simply a state feedback controller! *Hint: Use the following formulas for partial derivatives:* $\frac{\partial}{\partial x}(x^T M y) = My$, $\frac{\partial}{\partial y}(x^T M y) = M^T x$. *Are* Q, P_{i+1}, R *symmetric?*

4. Let's analyze the P_k matrix in further detail. First, using your input u_i from the previous question, show that the optimal cost to go at step i may be computed:

   $$J_i^o = x_i^T (A^T P_{i+1} A + Q - A^T P_{i+1} B (R + B^T P_{i+1} B)^{-1} B^T P_{i+1} A) x_i \tag{15}$$

5. We now have two expressions for J_i^o. Let's use them to identify how P_i changes with i. Using our assumption that $J_i^o = x_i^T P_i x_i$ and your answer to the previous question, show that P_i is determined by the following equation:

   $$P_i = A^T P_{i+1} A + Q - A^T P_{i+1} B (R + B^T P_{i+1} B)^{-1} B^T P_{i+1} A \tag{16}$$

   This is a special type of discrete time matrix equation called a *Riccati equation*. We may determine P_i for any i by iterating the Riccati equation backwards from $P_N = Q_f$.

6. Go to the provided notebook (linked at the top of the assignment) and implement the LQR module. Attach the plots generated by the code to your solution.

[1] This may be formally proven by induction on k.

Problem 2: Concepts in Deep Q-Learning

Dynamic programming gives us the exact and optimal solution to multi-stage decision processes. In addition to working in the case where the system is *deterministic*, dynamic programming can be extended to the case where the system is *stochastic* (noisy).
When we lack information about our environment and system, however, we must make approximations before applying the principles of dynamic programming! *Reinforcement learning (RL)* helps us find approximate solutions to these optimal control problems in the case where we have limited information about our system. In RL, we'll assume we only have access to the following:

1. *State Estimation:* Information about where our system is currently located.
2. *Reward:* Information about *how good* the current state and input to our system is.

Using just this knowledge, how can we get our system to behave optimally? Before answering this question, let's introduce some basic RL terminology. Firstly, instead of referring to a state at time t as x_t, the field of RL uses s_t by convention. Similarly, instead of using u_t for an input, we use the letter a_t in RL, and refer to inputs as *actions*.
Furthermore, instead of modeling our system with an explicit model of the form $x_{k+1} = f(x_k, u_k, w_k)$, we'll model our system with a *probability distribution* that tells us the probability of landing in a new state s_{t+1} given we started at a state s_t and took an action a_t:

$$p(s_{t+1}|s_t, a_t) \tag{17}$$

Associated with each pair of state and action, (s_t, a_t), we have a reward, $R(s_t, a_t)$. This reward tells us *how good* a particular state-action pair is. To find the best action a_t to take given that we're in a state s_t, we define a control *policy*, which is a closed loop feedback controller. This policy is conventionally denoted by the letter π:

$$a_t = \pi(s_t) \tag{18}$$

Where π represents the policy function returning our input. Using this terminology, we may frame our goal in a general reinforcement learning problem. We'd like to find a policy $\pi^*(s)$ that *maximizes* the expected total reward over N steps:

$$\pi^*(s) = \arg\max_{\pi} \mathbb{E}\Big[\sum_{t=0}^{N} \gamma^t R(s_t, \pi(s_t))\Big] \tag{19}$$

Where $\gamma \in (0, 1]$ is a constant called the *discount factor*, used to prioritize the rewards earlier in time over those that come later. Note that if we choose $\gamma < 1$, we can take the limit of this cost function as $N \to \infty$ to find an optimal policy for all time. How can we solve this optimization problem in practice?
Q-learning is a technique in reinforcement learning commonly used to solve this optimization problem. In Q-learning, we wish to find a special function called a Q-function, which tells us about the *quality* of a particular state-action pair. The Q-function:

$$Q(s, a) \tag{20}$$

Returns the total sum of rewards starting from a state s, taking a first action a, and following an optimal policy π after taking the first action a. We may prove that the optimal Q-function, $Q^*(s, a)$, *must* satisfy the following equation at each state-action pair (s_t, a_t) if the action a_t is provided by an optimal policy:

$$\mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') - Q^*(s_t, a_t)] = 0 \tag{21}$$

Where r_t is the reward at time t and γ is the discount factor. If we can identify this optimal $Q^*(s_t, a_t)$ function, we'll be able to solve for an optimal control policy $a_t = \pi^*(s_t)$ at each value of s_t. Let's discuss a method of determining this optimal Q-function from data!

Questions

These questions provide a conceptual overview of the processes underlying deep Q-learning.

1. Imagine that we have a data set S consisting of n pairs of (x, y) points, and that we'd like to fit the following quadratic function to the data set:

$$f_\theta(x) = \theta_1 x^2 + \theta_2 x + \theta_3 \tag{22}$$

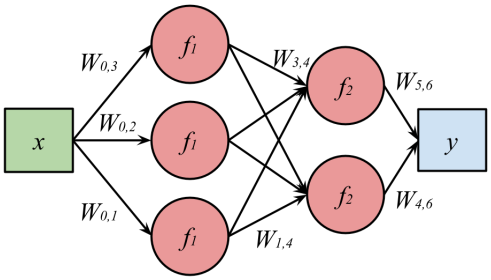
To fit this function to the data, we must find the optimal value of $\theta = [\theta_1, \theta_2, \theta_3]$ such that our "parameterized fit function," $f_\theta(x)$, closely matches the data. Provide a brief worded explanation as to why solving the following optimization problem:

$$\theta^* = \arg\min_\theta \frac{1}{n} \sum_{(x_i, y_i) \in S} (f_\theta(x_i) - y_i)^2 \tag{23}$$

Where n is the number of data points in S, would allow us to identify the value of θ^* that makes $f_\theta(x)$ fit the data. Note that we perform this sum over all pairs of collected (x_i, y_i) data in our set, S. The function we minimize here is called an l_2 loss function!

2. A popular technique for solving minimization problems over large sets of data is called *gradient descent*. Go to the provided notebook and implement the module on gradient descent in Q-learning. Attach the plots generated by the code to your solution.

3. Suppose that we can't find a simple function to fit our data well, and that we'd like a more expressive model. We can use a *neural network* to fit a function $f_\theta(x)$ to complex data! We can visualize the structure of neural networks with diagrams such as the following:



By taking in data points x, scaling them by constant weights $w_{i,j}$, and passing them into compositions of nonlinear functions called *activation functions*, neural networks can fit arbitrarily complex data. Consider the following simple neural network expression:

$$f_\theta(x) = f_2(w_{1,2} f_1(w_{0,1} x)) \tag{24}$$

Where f_1, f_2 are nonlinear activation functions. If $\theta = [w_{0,1}, w_{1,2}]^T$, find an expression for the gradient $\frac{\partial f_\theta(x)}{\partial \theta} = \nabla_\theta f_\theta(x)$ in terms of $\frac{\partial f_1(x)}{\partial x}$, $\frac{\partial f_2(x)}{\partial x}$. *Hint: Apply chain rule.*

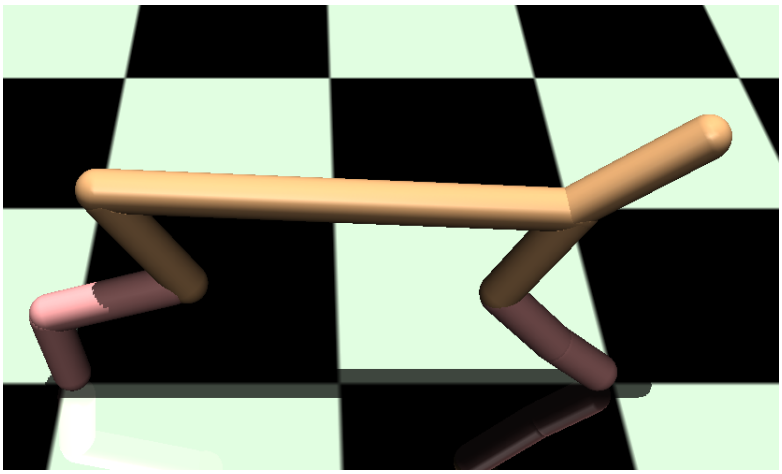
4. In reinforcement learning, we collect data sets S_{exp} called *experience* data sets. These contain (state, action, next state, reward) pairs of the form $S_{exp} = \{(s_t, a_t, s_{t+1}, r_t)\}$. Let's fit a parameterized Q-function $Q_\theta(s, a)$ to this data! Provide a brief worded explanation as to why finding a set of parameters θ^* that solve the optimization problem:

$$\theta^* = \arg\min_\theta \frac{1}{|S_{exp}|} \sum_{(s_t, a_t, s_{t+1}, r_t) \in S_{exp}} (r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') - Q_\theta(s_t, a_t))^2 \tag{25}$$

Where $|S_{exp}|$ is the number of (s_t, a_t, s_{t+1}, r_t) data points, allows us to find a Q_θ that approximates the optimal Q-function. Note that the sum is performed over all pairs of (s_t, a_t, s_{t+1}, r_t) data in S_{exp}. *Hints: Recall at the condition for an optimal Q-function:* $\mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') - Q^*(s_t, a_t)] = 0$. *How is this condition reflected in the loss function above? How does this compare to the l_2 loss function?*

Problem 3: Reinforcement Learning in Legged Robotics

A primary area of application for reinforcement learning is *legged robotics!* For quadrupedal (four-legged) and bipedal (two-legged) robots, for example, neural networks prove to be exceptionally useful tools in modeling and dealing with complex dynamics.
Using the tools of reinforcement learning, we may develop strong control policies for legged robot systems. Let's gain some practice in applying reinforcement learning to a simple legged robot system! In this question, we'll develop a reinforcement learning controller for a simple legged system called a half-cheetah.



Above: The half-cheetah model

This is a common benchmark system in reinforcement learning. Our goal is to make the half-cheetah robot move from left to right in a smooth manner. Let's learn about the tools we can use to accomplish this task!

Questions

1. Go to the provided notebook and implement the reinforcement learning for legged robotics module. Record the video generated by the simulator and attach a link to the video to your solution.

Bonus: Write a Poem!

After a long semester, you've finally reached the end of 106B/206B! Congratulations! To celebrate your accomplishment, write a short poem about robotics, controls, or your experience in this course. We'll share our favorite poems with the class and give them some *extra* extra credit. All types of poems are welcome! *Note: You may not use LLMs such as Chat-GPT for this question. The point of this question is to be creative and have a little bit of fun!*