Optimal Control

- Class of ways to control a system

  ex. MPC, CLF, CBFs, ...

- Build out a framework for solving OC problems

① Some DT system

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

state vec. — input vec.

② Objective variables

For time steps N, find optimal inputs

$$\boxed{u_0 \dots u_{N-1}}$$

③ General cost function

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

terminal cost — stage costs

What we want to solve:

$$\underset{u_0^* \dots u_{N-1}^*}{\arg\min} \; L_f(x_N) + \sum_{k=0}^{N-1} L(x_k, u_k)$$

* Often really hard to solve!

One way to solve ...

Dynamic Programming

- One framework used to solve OC problems
- We break down our problem into smaller steps

- Assume we already know J^o_{N-3}
  - Solve for optimal inputs for $i < N-3$

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

$$= \min_{u_i \dots u_{N-1}} \; L(x_i, u_i) \boxed{+ L_f(x_N) + \sum_{k=i+1}^{N-1} L(x_k, u_k)}$$

Taken out of Σ

$$= \min_{u_i \dots u_{N-1}} \; L(x_i, u_i) + J^0_{i+1}(x_{i+1})$$

$\rightarrow x_{i+1} = f(x_i, u_i)$

$$\boxed{J^0_i = \min_{u_i} \; L(x_i, u_i) + J^0_{i+1}(f(x_i, u_i))}$$

* Bellman Equation

→ Recursive equation for cost

→ Use D.P.

$J^0_N = L_f(x_N)$ → function of our last state

$$J^0_{N-1} = \min_{u_{N-1}} \; L(x_{N-1}, u_{N-1}) + L_f(f(x_{N-1}, u_{N-1}))$$

⋮

Solve for $u_{N-1}(x_{N-1})$

Able to solve for optimal input as a function of our current state

⋮

Work backwards to J^0_0 (solve for optimal inputs along entire trajectory)

Problem:
- System $x_{k+1} = a x_k + b u_k$
- Cost $J = \underbrace{x_N^2}_{\text{terminal cost}} + \sum_{k=0}^{N-1} \underbrace{(x_k^2 + u_k^2)}_{\text{stage cost}}$

* Work backwards to solve for optimal cost @ each time step

$\Rightarrow$ Start w/ terminal cost

$$J_N^o = L_f(x_N) = x_N^2$$

$\rightarrow$ Take one step back

$$J_{N-1}^o = \min_{u_{N-1}} \left\{ L(x_{N-1}, u_{N-1}) + \underline{J_N^o(x_N)} \right\} \text{ Bellman eq.}$$

$$= \min_{u_{N-1}} x_{N-1}^2 + u_{N-1}^2 + \underline{x_N^2}$$

$$= \min_{u_{N-1}} x_{N-1}^2 + u_{N-1}^2 + (a x_{N-1} + b u_{N-1})^2$$

* Expressed optimal cost as a function of our current state!

$$J_{N-1}^o = \min_{u_{N-1}} x_{N-1}^2 + u_{N-1}^2 + a^2 x_{N-1}^2 + 2ab\, x_{N-1} u_{N-1} + b^2 u_{N-1}^2$$

Want to find input minimizing cost!

Quadratic func., set derivative to 0

$$\frac{\partial J_{N-1}^o}{\partial u_{N-1}} = \cancel{2} u_{N-1} + \cancel{2} ab\, x_{N-1} + \cancel{2} b^2 u_{N-1} = 0$$

Solve for u_{N-1}!

$$u_{N-1}(1 + b^2) + ab\, x_{N-1} = 0$$

$$\boxed{u_{N-1} = \frac{-ab\, x_{N-1}}{1 + b^2}}$$

$\rightarrow$ Effectively just state feedback!

Can work all the way back to x_0!

* A basic LQR controller!

Q - Learning

- Might have systems where transition function + costs are unknown
  - Still want to find best input @ any given state

↓

We experiment! Run a bunch of episodes in our system and see what happens

* Reward = inverse of cost (we want reward)

- Q-Function: $Q(s, a)$, function of state & action
  - A model for what we expect our total rewards to be if we perform an action @ some state
  - When running experiments,

$$\boxed{\text{sample} = R(s, a, s') + \gamma \cdot \max_{a'} Q(s', a')}$$

  - we run some action from some state
  - Get a reward $= R(s, a, s')$
  - Add that reward to the total rewards from next state
  - γ: discount factor prefer <u>early</u> rewards

* Temporal Difference Learning
- way to update the Q-value w/ samples

$$Q(s, a) = (1 - a)\, Q(s, a) + a \cdot \text{sample}$$

- <u>Policy</u>: what action you should take @ a given time step
  ex. Taking the action that maximizes Q-value

$$a = \max_{a} Q(s, a)$$

C106B/206B Discussion 11: Optimal Control & RL

1 Introduction

Today, we'll talk about:

1. Optimal Control
2. Dynamic Programming in Optimal Control
3. Q-Learning
4. Reinforcement Learning

2 Optimal Control

At the center of optimal control, we have the question: how can we find a control input u that moves our system in some *optimal* manner? We encode what's "optimal" and what's not with a cost function, which we represent with the letter J.

By convention, in optimal control, we seek to find the solution to the optimization problem:

$$u^* = \arg\min_{u \in \mathcal{U}} J(x, u) \tag{1}$$

This gives us an optimal feedback control law as a function of our state, x. How can we solve this optimization problem subject to the constraint of our system dynamics?

3 Dynamic Programming in Optimal Control

Dynamic programming is a famous technique that we can use to solve a variety of optimal control problems. Here, we'll discuss the discrete time version of dynamic programming. Let's imagine we have a linear discrete time system of the form:

$$x_{k+1} = f(x_k, u_k) \tag{2}$$

Imagine we specify some final time of interest, N, and that we'd like to find an optimal sequence of inputs $u_0, ..., u_N$ to the system. A common cost function for this type of system is the following:

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

As it allows us to account for all N steps! L_f is called the *terminal* cost, while L is called the *stage* cost. Dynamic programming finds a sequence of inputs $\{u_0, u_1, ..., u_{N-1}\}$ that minimizes this cost function by breaking the problem up into smaller, easier to solve pieces.

We can break the problem up into these pieces by considering the *optimal cost to go* - the optimal cost *remaining* after we've already executed j steps and still have $N - j$ steps left over. We express this as:

$$J_j^o = \min_{\{u_k, \ldots, u_{N-1}\}} [L_f(x_N) + \sum_{k=j}^{N-1} L(x_k, u_k)] \tag{4}$$

The famous *Bellman equation* allows us to write the optimal cost to go *recursively*! This equation states:

$$J_j^o = \min_{u_j \in \mathcal{U}} [L(x_j, u_j) + J_{j+1}^o(x_{j+1})] \tag{5}$$

Where $x_{j+1} = f(x_j, u_j)$. Thus, the Bellman equation turns our problem from an optimization over a *sequence* of inputs to a set of optimizations over single inputs. By writing problems in this manner, we can determine the optimal control sequence $\{u_0, u_1, ..., u_{N-1}\}$ to the system.

Problem: Consider the discrete time system $x_{k+1} = ax_k + bu_k$, where $x_k, u_k \in \mathbb{R}$. Using dynamic programming, find an expression for the optimal input u_{N-1} and the optimal cost to go J_{N-1}^o in the optimal control problem:

$$U^* = \arg\min_{U} x_N^2 + \sum_{k=0}^{N-1} (x_k^2 + u_k^2) \tag{6}$$

Where $U = \{u_0, ..., u_{N-1}\}$ is the sequence of optimal inputs. This is a simple formulation of the famous LQR control problem!

4 Q-Learning

A Q-function, defined as $Q(s, a)$, is a function to calculate the total rewards of a trajectory if a certain action is taken from a certain state. Q-learning builds off the idea of optimal control but brings in the idea of experiential updates to iteratively improve a Q-function. Transition probabilities and costs of any movements are unknown. Instead, we define some reward function dependent on the overall goal we want to accomplish. The system is trained to either learn the Q-values or find the optimal policy, defined as the action resulting in the highest reward at any given state.

Temporal-difference learning with Q-values incorporates samples taken into an exponential moving average that updates the rewards for a particular state-action pair. Some action is taken from a state, which is recorded as a sample:

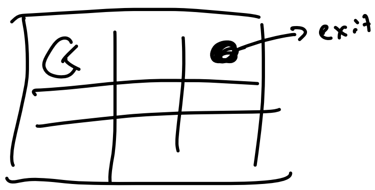
$$\text{sample} = R(s, a, s') + \gamma \cdot \max_{a'} Q(s', a')$$

γ is the discount factor, which prefers more recent rewards. As the saying goes, "A dollar today is better than a dollar tomorrow!" Then, the Q-value is updated, as per the following equation:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot \text{sample}$$

Above, α is defined as the *learning rate*, or the weight we want to give our new sample. Typically, this learning rate reduces over time.

Problem: Let's look at the following gridworld. The top-right square (3,1) has an exit, with a reward of +10. A movement can be performed to any adjacent square, and it succeeds with some unknown probability. Let's say we see the following episodes:



Episode 1:

- (1, 1), right, (2, 1) $\rightarrow$ $0 + 0.1(0.1) = 0.1$
- (2, 1), right, (3, 1) $\rightarrow$ $0 + (0.1) \cdot 10 = 1$
- (3, 1), exit, +10 reward $\rightarrow$ $+10$

Episode 2:

- (1, 1), right, (1, 2) $\rightarrow$ $0 + 0.1(0.01) = 0.001$

- (1, 2), right, (2, 2) $\rightarrow 0 + 0.1(0.1) = 0.01$
- (2, 2), right, (2, 1) $\rightarrow 0 + 0.1(1) = 0.1$
- (2, 1), right, (3, 1) $\rightarrow 0 + 0.1(10) = 1$
- (3, 1), exit, +10 reward $\rightarrow +10$

Episode 3:

- (1, 1), right, (2, 1) $\rightarrow 0.001$
- (2, 1), right, (2, 2)
- (2, 2), right, (3, 2)
- (3, 2), up, (3, 1)
- (3, 1), exit, +10 reward

Given a discount factor γ of 0.1 and a learning rate α of 0.5, perform TD-learning to find the Q-value for the state-action pair of [(1,1), right]. $s = (1, 1)$ $a = \text{right}$

After Ep. 1 : $Q(s,a) = 0.1$

After Ep. 2 : $Q(s,a) = (1 - 0.5) \cdot 0.1 + 0.5(0.001)$

After Ep. 3: . . .

5 Reinforcement Learning

The gridworld example is a small and contained. Rewards are easy to store for every state and maximize by searching through a table. However, for more complicated systems and tasks, performing calculations in this space becomes intractable.

There are many, many ideas within reinforcement learning, some of which you will have read papers about in journal club. There are proofs about techniques, applications in certain domains, implementations in novel ways, etc. Below are some high-level ideas in the ways that deep reinforcement learning is performed (deep because of neural networks), which should give you a good start when you start reading papers in this space.

1. **Imitation Learning:** Imitation learning works exactly the way it sounds. A bunch of expert actions are provided, and a network learns to perform the expert action from any given state. The expert actions are then effectively imitated.

2. **Policy Gradients:** With policy gradients, a reward function is provided. Then, the neural network is trained to predict actions that maximize the reward function. Because the reward is a function, it has a gradient that we can use; we perform what is effectively gradient ascent along the reward function (or gradient descent along its inverse, the loss).

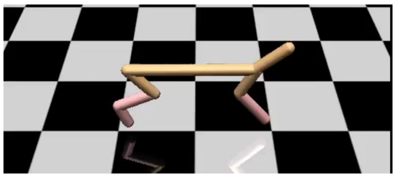
3. **Soft Actor-Critic:** This very commonly-used method is the continuous version of Q-learning for higher-dimensional action spaces. Two networks are trained: a critic network predicts Q-values and an actor network predicts the actions that maximize those Q-values.

4. **Model-Based Reinforcement Learning:** This method is often used in systems like humanoid robots, where predicting the next state from a particular action is difficult. A model is trained to predict this next state (essentially $f(x, u)$), and a standard trajectory planning technique like MPC is used to optimize the system's path.

5. **Offline Reinforcement Learning:** One issue with systems is that we may not be able to get samples of the particular task that we are dealing with. We do, however, have a corpus of samples from a separate task performed by the same system. Offline RL techniques allow us to bridge this gap.

There are other important things you would have to know if you are going into reinforcement learning, such as balancing exploration and exploitation of a policy (finding new states or taking advantage of the best-trained policy), inverse reinforcement learning (learning a reward function), and algorithms (CQL, IQL, AWAC, etc.). However, the ideas above should give you some solid grounding while reading new papers on this topic!

Problem: Consider the half-cheetah, a bipedal robot. We want to train this robot to walk in simulation using policy gradients. To feed this into our network, we first need to represent the current state of the robot in some kind of feature vector so that we may predict the optimal action. Construct an observation vector.



Problem: Now that we have a state, we want to create a reward function that the learning algorithm optimizes. At the same time, we want to punish movements that are too large to preemptively avoid losing balance. Construct a reward function.