# C106B Discussion 11: Optimal Control & RL

## 1 Introduction

Today, we'll talk about:

1. Optimal Control
2. Dynamic Programming in Optimal Control
3. 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:

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

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 $J^o_{N-1}$ 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 Reinforcement Learning

Reinforcement learning builds off the idea of optimal control but brings in the idea of experiential updates to iterate up on a given policy or 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:

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

$\gamma$ 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, $\alpha$ 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)
- (2, 1), right, (3, 1)
- (3, 1), exit, +10 reward

**Episode 2:**

- (1, 1), right, (1, 2)
- (1, 2), right, (2, 2)
- (2, 2), right, (2, 1)
- (2, 1), right, (3, 1)

- (3, 1), exit, +10 reward

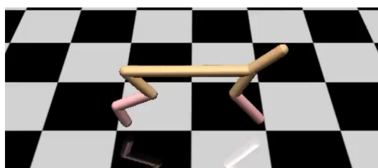
**Episode 3:**

- (1, 1), right, (2, 1)
- (2, 1), right, (2, 2)
- (2, 2), right, (3, 2)
- (3, 2), up, (3, 1)
- (3, 1), exit, +10 reward

Given a discount factor $\gamma$ of 0.1 and a learning rate $\alpha$ of 0.5, perform TD-learning to find the Q-value for the state-action pair of [(1,1), right].

Alternatively, the policy can be learned directly. Instead of discovering Q-values for a particular space, which may not only be computationally difficult but also unnecessary in continuous domains, the current state vector can be fed in to some kind of algorithm that returns the optimal movement to maximize some reward function. This pipeline usually contains some kind of neural network (hence the term "deep reinforcement learning").
During the exploration stage, the network's weights are updated to predict the optimal policy given some state. The system is allowed to transition randomly to build a database. Over time, exploration is reduced in favor of exploitation, where the optimal policy is executed and refined. Ongoing research develops better algorithmic pipelines for training a policy and performing it.

**Problem:** Consider the half-cheetah, a bipedal robot. We want to train this robot to walk in simulation using reinforcement learning. 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 from any given position. 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.

Great! Now we have the tools for a deep RL algorithm. Implementations for this become exceedingly complex, but we have the basic building blocks - an input vector (our features) and a cost function (our rewards), allowing us to bring in neural networks to optimize policies.