Lecture 1

Introduction

Model-based methods provide a powerful framework for controlling challenging robotic systems; however, imperfect models often lead to poor performance during real-world deployment. Machine learning methods provide one means of addressing deficient models, either through explicitly learning a model of the system dynamics or computing a control policy directly from data. In this course, we will explore the intersection between optimal control and machine learning, covering both model-free and model-based methods for learning-based control. We will start with a review of dynamic programming and its popular algorithmic instantiations: value iteration and policy iteration. We will then explore other methods for model-based controller synthesis, followed by an exploration of three primary means of incorporating learning into control synthesis: learning value functions, control policies, and dynamics models The course will culminate in a discussion of model-based reinforcement learning and adaptive optimal control. We will also discuss advanced topics such as learning Lyapunov functions and contraction metrics from data, iterative learning control, and techniques for adaptive nonlinear model predictive control.

1.1. Notation

The optimal control and reinforcement learning communities use a variety of overlapping (and conflicting) definitions and notations. In these notes, we will attempt to unify and clarify these notations and definitions where possible. We will also tend to use notations and definitions from the optimal control community.

- s will be used to represent discrete states. We define $s \in \mathbb{S}$, where $\mathbb{S} = \{s0, s1, s2, ..., sN\}$.
- a will be used to represent discrete actions. We define $a \in \mathbb{A}$, where $\mathbb{A} = \{a0, a1, a2, ..., aN\}$.
- x will be used to represent continuous state values. We define $x \in \mathbb{X}$ or $x \in \mathbb{R}^{N_x}$.
- u will be used to represent continuous control input values. We define $u \in \mathbb{U}$ or $u \in \mathbb{R}^{N_u}$..
- g will be used to represent a "running cost". Occasionally, r will be used to represent a reward.
- $J(x)$ represents a value function or cost-to-go.
- $\dot{x} = f(x, u)$ represents a transition model for a deterministic continuous dynamical system.
- $x_{k+1} = f(x_k, u_k)$ represents a transition model for discrete time continuous dynamical system.
- $s_{k+1} \sim p(s_{k+1}|s_k, a_k)$ represents a transition model for a stochastic, discrete time, discrete state system.
- The subscript k will represent a discrete time index (e.g. x_k).
- The superscript k will denote an iteration to an algorithmic update. (e.g. $\pi^k(s)$)

Traditionally, the optimal control and reinforcement learning communities use different notation to refer to similar concepts:

	Optimal Control	Reinforcement Learning
state: in the sense of Markov	$x \in \mathbb{X}$	$s \in \mathbb{S}$
control/action: inputs to the system	$u \in \mathbb{U}$	$a \in \mathbb{A}$
dynamics/transition model	$x_{k+1} = f(x_k, u_k, w_k)$	$s_{k+1} \sim p(s_{k+1}\|s_k, a_k)$
cost/reward	$g(x, u)$	$r(s, a)$
cost-to-go/value function	$J(x)$	$V(s)$

1.2. The Learning-Based Control Problem

If you work in robotics, at some point you will encounter a problem where you do not have a perfect model of your robotic system. If you have a controls background, you may try to write down your robot dynamics in a general form, such as

$$\dot{x} = f(x, u, w), \tag{1.1}$$

where x is the state, u is the control, and w is a random variable. In many cases, you don't know f, or you know f only partially. What do you do?

If you have a computer science background, you might try to discretize the state and action spaces to a transition model that looks like

$$s_{k+1} \sim p(s_{k+1}|s_k, a_k), \tag{1.2}$$

where s_k is your discrete state and a_k is your discrete action. However, you still don't know $p(s_{k+1}|s_k, a_k)$.

You know you need to learn *something* about the system to proceed. What what should you learn?

If your a controls engineer, you may try to learn a representation for f using data. Or, you may try to learn u, perhaps through some means of iterative learning control. The computer scientist may try to learn a value function $J(x)$ via reinforcement learning.

1.3. What is Learning?

Before proceeding further, perhaps we should define learning. In this course, when we discuss learning, we will be primarily be referring to the use of *supervised learning* for regression, or more broadly, the use of *function approximation* to improve controller performance. We choose this definition so as not to conflate our notion of learning with *reinforcement learning*, of which function approximation is often a key component. Instead, we will try to refer to reinforcement learning as *approximate dynamic programming*.

However defining learning as function approximation still doesn't help us decide what exactly we should learn.

1.4. Learning Paradigms

When designing a learning-based control system, a fundamental design choice is deciding what to learn. Typically, there are three major ways to employ learning in your control design: one could learn some sort of parameterization of the performance metric (e.g., the cost-to-go, $J(x)$), the dynamics model itself (e.g., $f(x, u)$), the policy (e.g., $u = \pi(x)$), or some combination of the three. Approximate dynamic programming approaches tend to either learn the value function or the policy. Control approaches tend to learn the dynamics model.

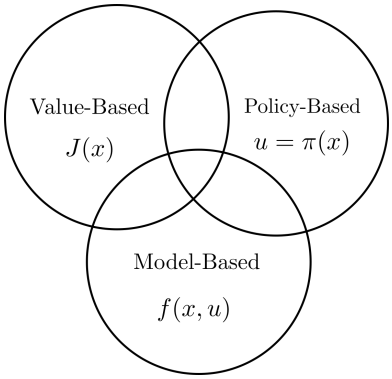

Figure 1.1: A diagram of the learning paradigms for control design.

In this course, we will refer to these learning paradigms as value-based, model-based, policy-based, or mixed. Note, when referring to these paradigms, we are referring to *what is being learned*, or to which component of the control algorithm learning is being applied.

1.5. Course Outline

This course is separated into two major parts. In the first part, we consider methods where the dynamics model is fully known and optimization can be employed for controller synthesis. In the second part of the course, we consider approaches where major components of the controller are learned using function approximation. In particular, we consider situations where the major learning paradigms are applied (value-based, policy-based, model-based, and mixtures), and how these learning paradigms can employed in cases where the the dynamics model is partially or entirely unknown.

Lecture 2

Dynamic Programming and Optimal Control

Tools from optimal control have provided a fairly useful means for addressing many challenging problems in robotics. Even when optimality is not the ultimate goal, these methods can provide a means of providing a feasible solution to otherwise intractable problems. Optimal control (and optimization) will provide the lense through which we view almost all of our approaches for learning-based control.

2.1. Objective Function

To understand optimal control, we must understand the notion of an objective function. Many times these objective functions are assumed to be additive, that is, the costs are summed over time.

For instance, in a discrete state, discrete action scenario, we often have

$$J = \sum_{k=0}^{N} g_k(s_k, a_k) \tag{2.1}$$

For continuous states, continuous actions, we have

$$J = \int_0^T g(x(t), u(t))dt. \tag{2.2}$$

These additive cost structures are very important, and as we'll see, allows us to exploit recursion via dynamic programming.

2.2. Principle of Optimality

To understand dynamic programming, we must first understand the *principle of optimality*.

The principle of optimality can be stated succinctly as: *the tail of an optimal sequence is optimal for the tail subproblem*.

Suppose the optimal solution for a problem passes through some intermediate point (x_1, t_1), then the optimal solution to the same problem starting at (x_1, t_1) must be the continuation of the same path.

Consider the following proof-by-contradiction:

Let J_1 be the cost-to-go from x_1 to x_f, and let J_0 be the cost-to-go from x_0 to x_1. Let J^* be the optimal cost-to-go from x_0 to x_f. Let J_1' be the cost-to-go from x_1 through some other point x_2 to x_f.

Assume that

$$J_1' < J_1 \tag{2.3}$$

But then:

$$J_0 + J_1' < J_0 + J_1 = J^* \tag{2.4}$$

is a contradiction.

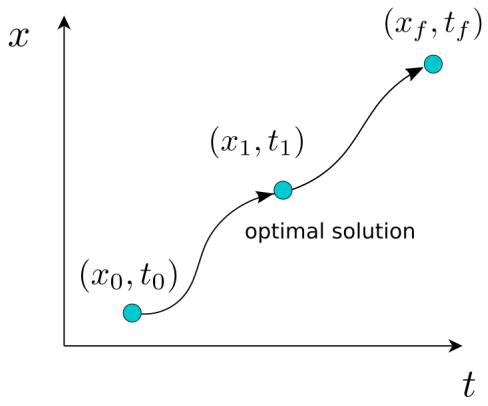

Figure 2.1: Principle of Optimality

2.2.1 Dynamic Programming

An optimal control problem for discrete state space, discrete action space can be written as:

$$\min_{s,a} \quad J = \sum_{k=0}^{N-1} g_k(s_k, a_k) + g_N(s_N) \tag{2.5}$$

$$s.t. \quad s_{k+1} = f(s_k, a_k) \quad k = 0, 1, ..., N-1 \tag{2.6}$$

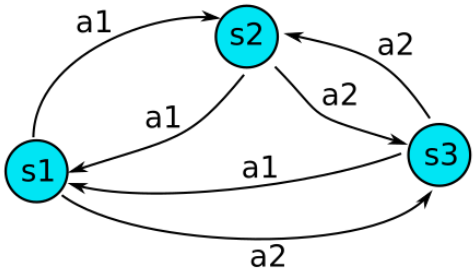

Figure 2.2: A simple directed graph

Richard Bellman used the principle of optimality to define a recursive relationship which is often referred to as the Dynamic Programming algorithm.

Let's expand out the above minimization and then apply recursion:

$$J^*(s_0) = \min_{a_k \in \mathbb{A}\ \forall k} \left[\sum_{k=0}^{N-1} g_k(s_k, a_k) + g_N(s_N) \right] \tag{2.7}$$

$$= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + g(s_1, a_1) + g(s_2, a_2) \dots \right] \tag{2.8}$$

$$= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + \big(g(s_1, a_1) + g(s_2, a_2) + g(s_3, a_3) \dots \big) \right] \tag{2.9}$$

$$= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + J^*(s_1) \right]. \tag{2.10}$$

For a general timestep k, we have the Dynamic Programming recursion:

$$J_k^*(s_k) = \min_{a \in \mathbb{A}} \left[g(s_k, a) + J_{k+1}^*(f(s_k, a)) \right] \tag{2.11}$$

$$= \min_{a \in \mathbb{A}} \left[g(s_k, a) + J_{k+1}^*(s_{k+1}) \right] \quad \forall k, s \in \mathbb{S} \tag{2.12}$$

Given an optimal cost-to-go, the optimal policy can then be extracted as follows:

$$\pi_k^*(s_k) = \arg\min_{a_k \in \mathbb{A}} \left[g(s_k, a_k) + J_{k+1}^*(f(s_k, a_k)) \right] \tag{2.13}$$

Example 1

Consider the shortest path problem, where it is possible to travel along the edges from A to B. Rather than computing the cost of every path between A and B, we can start at B and work backwards to each node in the graph to compute the optimal cost-to-go and the associated action (see Figure 2.3). This process is called dynamic programming. The solution arrived at using dynamic programming only requires on 15 calculations rather than 20. It scales $(n+1)^2 - 1$ v.s. $(2n)!/(n!)^2$.

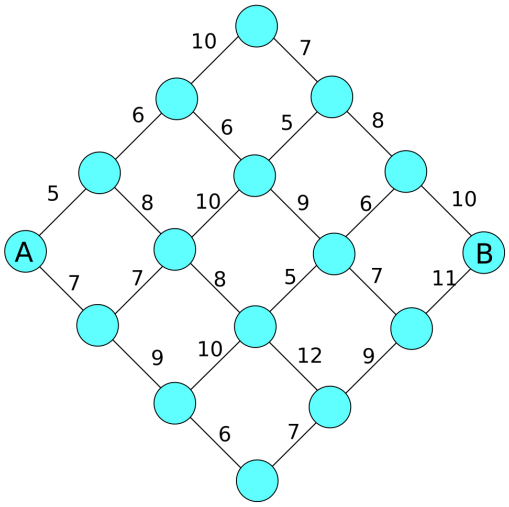

Figure 2.3: Shortest Path Problem

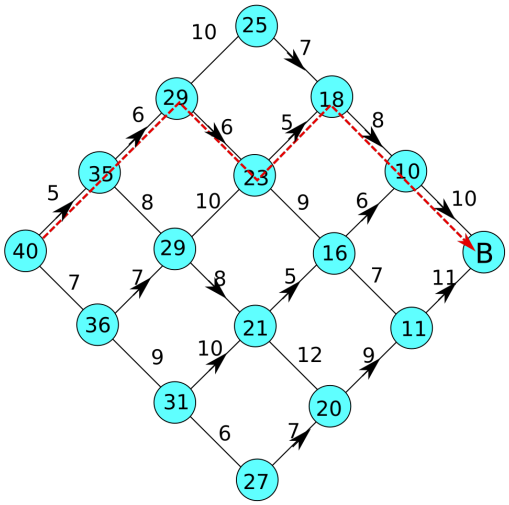

Figure 2.4: Shortest Path Problem Solution

Example 2

Consider a grid world, with obstacles (gray). The robot can take one of four actions (up,down,left,right). I has a cost of "1" for existing in the world, and incurs no cost at the goal.

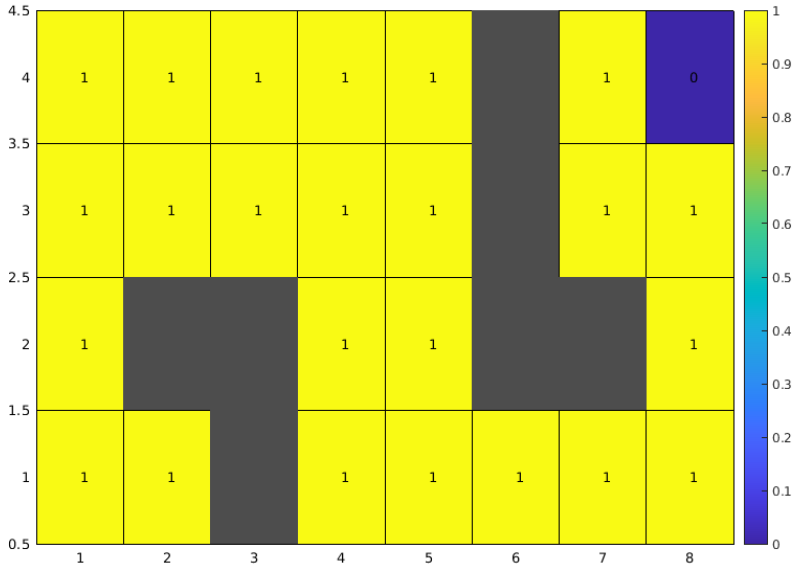

Figure 2.5: Value Iteration (at T-1)

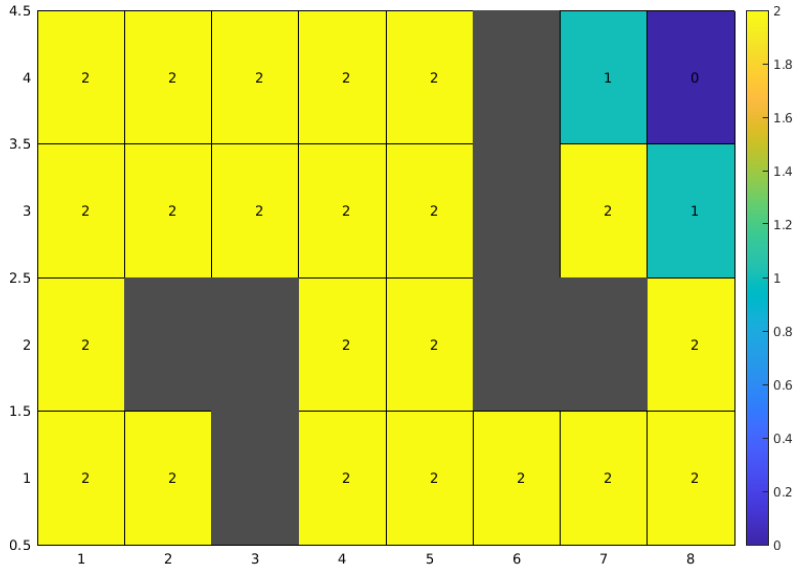

Figure 2.6: Value Iteration (at T-2)

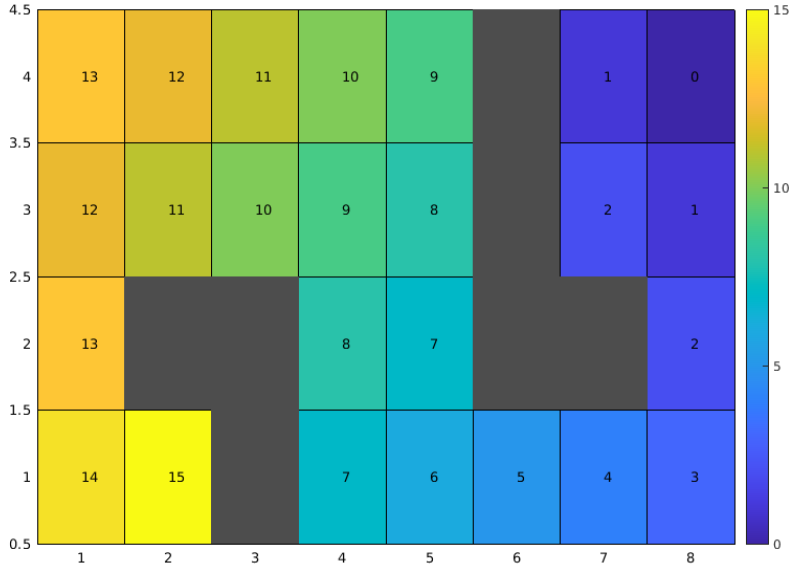

Figure 2.7: Value Iteration (after T steps)

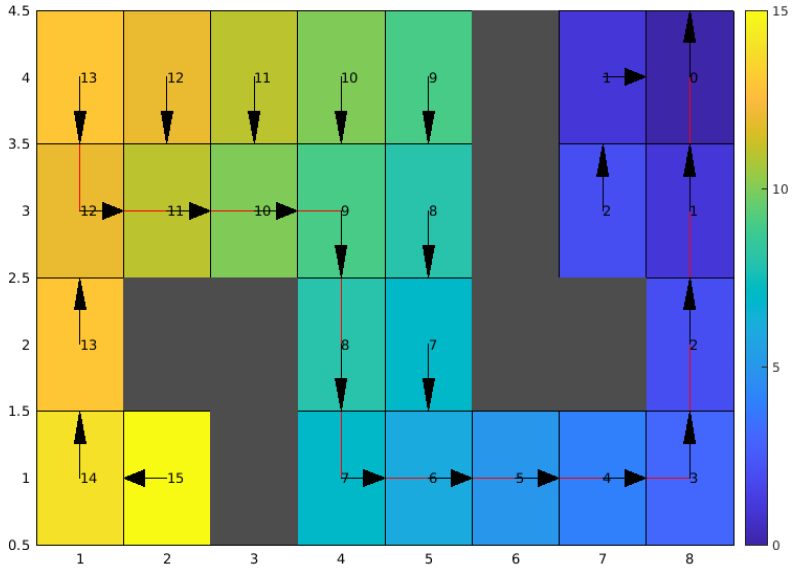

Figure 2.8: Optimal Policy after T-steps

2.2.2 Value Iteration and Infinite Horizon Problems

What happens when your cost has an infinite sum?
Apply a discount factor, $\gamma \in [0, 1)$, to make sure the sum converges to a finite value.

$$J^*(s_0) = \min_{a_k \in \mathbb{A}\ \forall k} \sum_{k=0}^{\infty} \gamma^k g(s_k, a_k). \tag{2.14}$$

Let's expand this out and then apply recursion:

$$\begin{aligned} J^*(s_0) &= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + \gamma g(s_1, a_1) + \gamma^2 g(s_2, a_2) \dots \right] && (2.15) \\ &= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + \gamma \big(g(s_1, a_1) + \gamma g(s_2, a_2) + \gamma^2 g(s_3, a_3) \dots \big) \right] && (2.16) \\ &= \min_{a_k \in \mathbb{A}\ \forall k} \left[g(s_0, a_0) + \gamma J^*(s_1) \right]. && (2.17) \end{aligned}$$

Given a timestep k, this leads to the modified Bellman equation:

$$J_k^*(s_k) = \min_{a_k \in \mathbb{A}} \left[g(s_k, a_k) + \gamma J_{k+1}^*(s_{k+1}) \right]. \tag{2.18}$$

Often, we will let $s_k = s$, $a_k = a$, and $s_{k+1} = s' = f(s, a)$ for simplicity of notation.

$$J_k^*(s) = \min_{a \in \mathbb{A}} \left[g(s, a) + \gamma J_{k+1}^*(s') \right]. \tag{2.19}$$

If the cost-to-go converges, then as $T \to \infty$, $J_k^* \to J^*$. We can then write:

$$J^*(s) = \min_{a \in \mathbb{A}} \left[g(s, a) + \gamma J^*(s') \right] \tag{2.20}$$

and

$$\pi^*(s) = \arg\min_{a \in \mathbb{A}} \left[g(s, a) + \gamma J^*(s') \right]. \tag{2.21}$$

To solve for J^* an algorithm known as Value Iteration is used:

$$\hat{J}^{*,i+1}(s) = \min_{a \in \mathbb{A}} \big[g(s,a) + \gamma \hat{J}^{*,i}(s') \big]. \tag{2.22}$$

For problems represented by discrete states and discrete actions, Value Iteration is an algorithm that leverages dynamic programming and the principle of optimality to iteratively compute the cost-to-go.

The algorithm starts by initializing an estimate of the optimal cost-to-go, $\hat{J}^*$, for all s. It then iteratively updates $\hat{J}^*$ using the Bellman equation until convergence.

Why does adding a discount factor ensure finite cost over an infinite horizon? Effectively, it creates a geometric series whose sum converges to a finite limit as $k \to \infty$ so long as $g(s,a)$ is bounded. If $-G \leq g(s,a) \leq G$, where G is a positive scalar, we can write

$$-\sum_{k=0}^{N} \gamma^k G \leq J(s_0) \leq \sum_{k=0}^{N} \gamma^k G \tag{2.23}$$

$$-\frac{G}{1-\gamma} \leq J(s_0) \leq \frac{G}{1-\gamma}. \tag{2.24}$$

2.2.3 Stochastic Problems

Can we optimize stochastic systems?

$$J^*(s_0) = \min_{a_k \in \mathbb{A} \ \forall k} E\bigg[\sum_{k=0}^{\infty} \gamma^k g(s_k, a_k) \bigg] \tag{2.25}$$

Expected cost preserves the dynamic programming recursion.

$$J^*(s) = \min_a E\big[g(s,a) + J^*(s') \big] \tag{2.26}$$

$$\tag{2.27}$$

Assume $g(s,a)$ is deterministic. The law of the unconscious statistician (LOTUS) gives

$$E\big[J^*(s') \big] = \sum_{s'} p(s'|s,a) J^*(s'). \tag{2.28}$$

Then

$$J^*(s) = \min_{a \in \mathbb{A}} \big[g(s,a) + \sum_{s'} p(s'|s,a) J^*(s') \big]. \tag{2.29}$$

Bibliography

[1] Russ Tedrake. *Underactuated Robotics*. 2023.

[2] Dimitri Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

[3] Drew Bagnell, Byron Boots, and Sanjiban Choudhury. *Modern Adaptive Control and Reinforcement Learning*. 2022.