A

EECS C106B / 206B Robotic Manipulation and Interaction **Spring 2022**

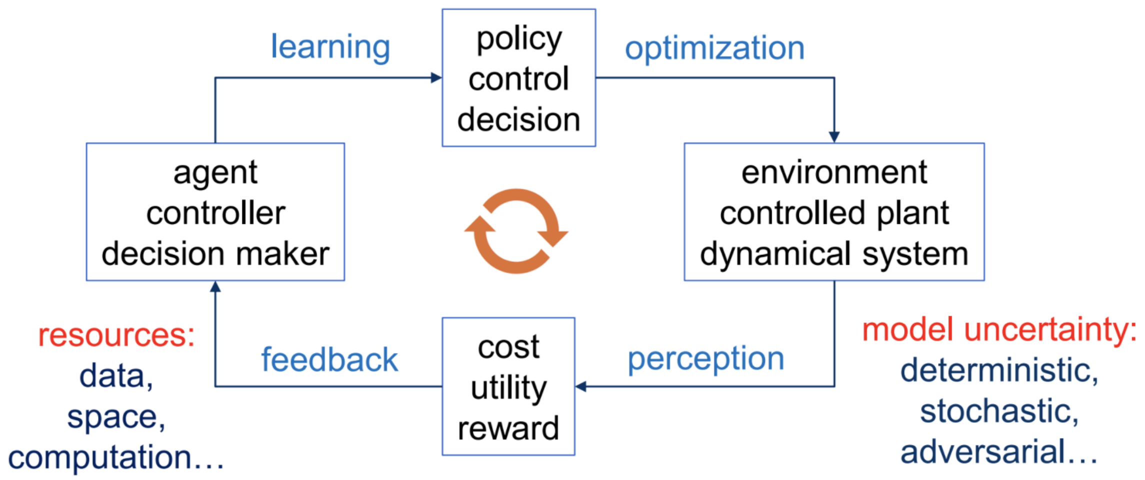
Lecture 5A: (Optimal Control, Reinforcement Learning, Vision)

Scribes: Bryan Chen, Aaron Rovinsky

5.1 Overview of Autonomous Systems and Control

5.1.1 A Common Closed-Loop Autonomous System Setup

Closed-loop autonomous systems, such as vacuuming robots, autonomous cars, video game players, stock traders, and other examples, follows the same decision loop:



5.1.2 Recent History of Control and RL

AI/RL involves learning through data/experience, simulation, model-free methods, and feature-based representations. Decision/Control/Dynamic Programming involves optimality, Markov Decision Programs, and policy/value iterations. Following is a history of their applications:

- 1950s: Exact dynamic programming and optimal control is formed (Bellman, Shannon)
- Late 80s-early 90s: AI/RL and Decision/Control/DP ideas meet
- 1992, 1996: The first autonomous control success in backgammon programs (Tesauro)

- Mid 90s: Algorithmic progress, analysis, and applications are made. The first books are published
- Mid 2000s: ML, Big Data, Robotics, Deep Neural Nets
- 2016: Alpha Go, AlphaZero, DeepMind

5.1.3 New Challenges

In modern problems, RL can handle the following issues: unknown or changing environment, delayed rewards or feedback, enormous state/action space, and nonconvexity.

5.2 Terminology

5.2.1 State Space Model

The state space models are based on the environment controlled plant dynamical system. They are similar, except OC/DP has a control space and dynamical system whereas AI/RL has an action space and MDP transition or simulation.

Table 5.1: State Space Model Comparison

OC/DP	**AI/RL**
State and Control space: S, U	State and Action space: S, A
State: $x_k \in S, k = 0, 1...$	State: $s_t \in S, t = 0, 1...$
Control: $u_k \in U, k = 0, 1...$	Action: $a_t \in A, t = 0, 1...$
Dynamical System:	MDP Transition (or simulation):
$\mathrm{x}_{k+1} = f(x_k, u_k)$,$x_{k+1} = f(x_k, u_k, w_k)(stochastic)$	$\mathrm{T}_{ijk} = p(s_{t+1} = i \| s_t = j, a_t = k)$
Output/observation (feature): $\mathrm{y}_k = h(x_k, u_k) + n_k$	Observation (feature): $p(o_t \| s_t)$

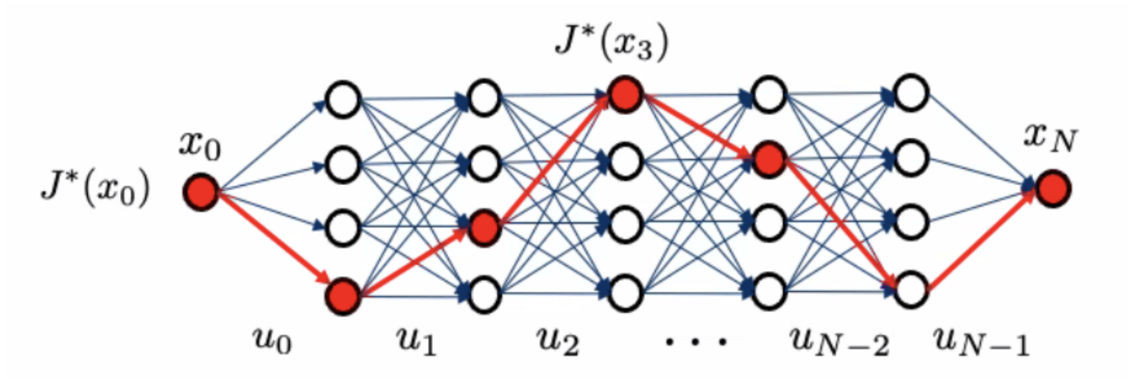
5.2.2 Optimization Objective

The optimization objective is based on the policy control decision. For OC/DP, a cost function and control law is used, while AI/RL has reward and policy.

Table 5.2: Optimization Objective Comparison

OC/DP	**AI/RL**
Cost function: $J(x_0; u_0, ..., u_N) = \sum_{k=0}^{N} g(x_k, u_k)$	Reward return: $J(s_1; a_1, ..., a_T) = \frac{1}{T} \sum_{t=1}^{T} E[r(s_t, a_t)]$
Control law: $\mathrm{u}(\mathrm{x}_k); u^*(x_k)$,	Policy: $\pi(a_t \| s_t); \pi^*(a_t \| s_t)$,
$\mathrm{x}_{k+1} = f(x_k, u(x_k)), u_{k+1} = u(x_{k+1})$	$\mathrm{p}((\mathrm{s}_{t+1}, a_{t+1}) \| (s_t, a_t)) = p(s_{t+1} \| s_t, a_t)\pi(a_{t+1}, s_{t+1})$
Value function (minimal cost to go):	Value function (maximal return):
$J^*(x_0) = \min_{u(\cdot)} \sum_{k=0}^{N} g(x_k, k)$	$V^*(s_1) = \max_{\pi(\cdot)} \frac{1}{T} \sum_{t=1}^{T} E_\pi[r(s_t, a_t)]$

5.3 Principle of (Path) Optimality



Principle of Optimality (Richard Bellman '54): An optimal path has the property that any subsequent portion is optimal.

So optimality naturally lends itself to dynamic programming. We can express an optimal controller as:

$$J^*(x_k) = \min_{u_k}[g(x_k, u_k) + J^*(\underbrace{f(x_k, u_k)}_{x_{k+1}})], \forall x_k$$

Consider the Bellman operator

$$T(J)(x) = \min_{u}[g(x, u) + J(f(x, u))]$$

Repeatedly applying the Bellman operator is a form of dynamic programming which converges to the optimal controller, i.e. $T(J^*) = J^*$.

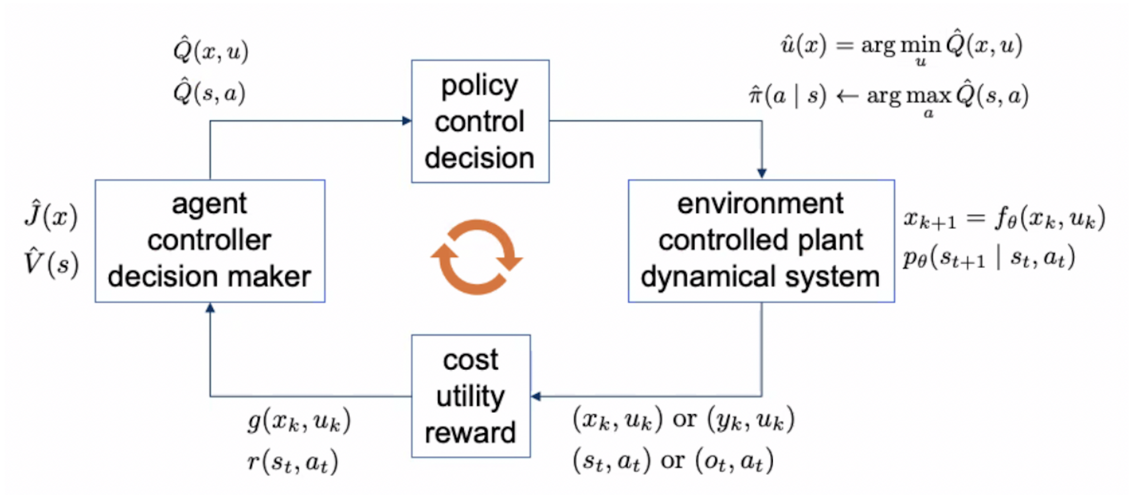
5.4 Value Function vs. Q-Function

OC/DP	**AI/RL**
$J^*(x_k) = \min_{u_k} \underbrace{[g(x_k, u_k) + J^*(f(x_k, u_k))]}_{Q(x_k,u_k)}, \forall x_k$	$V^*(s_t) = \max_\pi E_\pi[p(s_{t+1}\|s_t, a_t) \underbrace{[r(s_t, a_t) + V^*(s_{t+1})]}_{Q(s_t,a_t}$
$u_k^* = \arg\min_{u_k} Q(x_k^*, u_k)$	$\pi^*(s_t) = \arg\max_{a_t} Q(s_t, a_t)$
$x_{k+1}^* = f(x_k^*, u_k^*)$	$p(s_{t+1}\|s_t, \pi^*(s_t))$

In optimal control, the value function $J^*(x_k)$ describes the minimum total present and future cost of being in the current state x_k and taking an action u_k to end up in the state x_{k+1}, i.e. the total cost-to-go assuming the optimal controller. Similarly, in reinforcement learning, the value function $V^*(s_t)$ describes the total reward-to-go assuming the optimal policy. The Q-function is a very similar concept but considers the input/action as well. So the Q-function describes the optimal cost-to-go or reward-to-go assuming you are in state x_k or s_t and take the action u_k or $\pi(s_t)$.

5.5 The Closed-Loop Autonomous System Formalized

This diagram describes the learning/control process using the terminology established earlier in this lecture. An agent uses its policy to produce a control input, the agent interacts with its environment using using this



$\hat{Q}(x,u)$
$\hat{Q}(s,a)$
policy control decision
$\hat{u}(x) = \arg\min_u \hat{Q}(x,u)$
$\hat{\pi}(a \mid s) \leftarrow \arg\max_a \hat{Q}(s,a)$
$\hat{J}(x)$
$\hat{V}(s)$
agent controller decision maker
environment controlled plant dynamical system
$x_{k+1} = f_\theta(x_k, u_k)$
$p_\theta(s_{t+1} \mid s_t, a_t)$
cost utility reward
$g(x_k, u_k)$
$r(s_t, a_t)$
(x_k, u_k) or (y_k, u_k)
(s_t, a_t) or (o_t, a_t)

input and makes some observation, the observation translates to some kind of cost or reward, the agent is updated based on the observed cost or reward, and the process iterates.

5.6 From Principle to Computation

5.6.1 What to Compute and How

For both OC/DP and AI/RL, certain functions, control/policy, and identification must be computed.

	OC/DP	**AI/RL**
Optimal value function:	$J^*(x)$	$V^*(s)$
Optimal Q-function:	$Q^*(x,u)$	$Q^*(s,a)$
Optimal control/policy	$u^*(x)$	$\pi^*(a\|s)$ or $u^*(y), \pi^*(a\|o)$
System/model ID	$f^*(x,u)$	$p^*(s_{t+1}\|s_t, a_t$

One possible method is the closed-form solution from the Linear Quadratic Regulator setup:

$$J^*(x_k) = \min_{u_k} [x_k^T Q x_k + [u_k^T R u_k + J^*(Ax_{k+1} + Bu_k)]$$

This can be solved by using the Riccati equation with the form:

$$K_k = -(\bar{R} + \bar{B}^T V_{k+1} \bar{B})^{-1} \bar{B}^T V_{k+1} \bar{A}$$

$$V_k = \bar{Q} + \bar{A}^T V_{k+1} \bar{A} - \bar{A}^T V_{k+1} \bar{B} (\bar{R} + \bar{B}^T V_{k+1} \bar{B})^{-1} \bar{B}^T V_{k+1} \bar{A}$$

Another example is in parallel parking a nonholonomic car. Here, the optimal trajectories are zigzagging sinusoids.

$$\min \int_0^1 ||u(t)||^2 dt$$

5.6.2 Control vs Learning

Overall, control and learning have different applications, conditions, and assumptions. This is important when selecting a solution for a broad class of problems vs a few important instances.

OC/DP	**AI/RL**
LQR	Backgammon (Tesauro 1992)
Parallel parking	Chess (Deep Blue 1997)
Chained form systems	AlphaGo (2017)
Mechanical systems	Video games, robots

OC/DP conditions and assumptions: clear model class/uncertainty, clear cost function, low/moderate dimension, continuous state/time.

AI/RL conditions and assumptions: unknown models (but can sample from them), uncertain/long-horizon return, large-scale, high-dimensional, discrete state/time

5.6.3 Computation with Approximation

If no analytical or closed-form solution exists, there are several options to compute the approximate cost function $\tilde{J}$:

- Problem approximation: Use the optimal cost function of a related problem, then compute with exact DP
- Rollout and model predictive control: Use the cost function of some policy computed with a simplified optimization process
- Neural networks and other feature-based architecture: These can approximate the function with offline training
- Simulation to generate training data for architecture: Approximation architecture involve parameters ”optimized” with data
- Policy iteration, self-learning, repeated policy changes: Multiple policies are sequentially generated, and each provides data to train the next

5.7 What to Learn or Compute?

5.7.1 How to Compute?

There are on the order of 35^{80} possible chess configurations, 250^{150} possible Go configurations, and 10^{82} atoms in the universe. So how can we possibly hope to find optimal solutions/strategies (not to mention closed-form solutions) in these high-dimensional settings with extremely large state and/or action spaces? In principle this is possible given that humans learn to play games like Chess and Go reasonably well, and in practice researchers have been able to develop successful algorithms to play these games. How is this possible?

5.7.2 How to Approximate?

Given that we have no closed-form solution to many problems of interest, we must find some way to approximate solutions tractably. This can be done via featurization i.e. reducing the high-dimensionality. Consider the problem of autonomous driving, for example. The inputs to the system at the lowest level are image pixels, which are very high-dimensional. But within these images, only a few pixels are actually useful, for example those which tell you there is or is not an obstacle in your path. So when we train neural networks to obtain useful policies, we can loosely think of this as learning features from data and performing some kind of regression on those features.

5.7.3 How to Learn Low-Dimensional Structures?

Our ability to solve problems in high-dimensional spaces relies crucially on the fact that low-dimensional structures exist in high-dimensional spaces. Featurization really is compression, which is the process by which high-dimensional data is converted to low-dimensional data without losing too much information. The goals and roles of deep neural networks are compression, optimization, and linearization, i.e. neural networks take high-dimensional inputs and compress then to low-dimensional representations to optimize over. For example, when classifying digits using the MNIST image dataset, the inputs are high-dimensional images but neural networks (specifically convolutional neural networks) will learn features from the data like lines and curves at certain angles, and the presence or lack of these features in images allows the network to perform classification.

5.8 Some Representative Algorithms: Value-Based RL

5.8.1 Q-Learning

A basic Q-Learning algorithm was developed by Watkins and Dayan in 1992:

$$Q_{t+1}(s_t, a_t) = (1-\eta)Q_t(s_t, a_t) + \eta T_t(Q_t)(s_t, a_t)$$

where T is the Bellman operator. This is a model-free (no need to learn the dynamics of the system), stochastic approximation to solve the Bellman equation.

5.8.2 Sample Complexity

Yuxin Chen et. al. in 2021 showed that a theoretical lower bound on how many samples are needed to find the optimal Q-function is $|S||A|$, but in most practical settings the size of the sample space alone is enough to make gathering this many samples impossible. For example, for the AlphaGo program, $|S| = 2^{361}$. This is the tightest bound that has been derived, so theoretical bounds do not explain our ability to learn efficiently.

The key insight for explaining why we can solve high-dimensional problems tractably is the low-dimensional structure discussed earlier in the lecture. If we consider a tabular (matrix) Q-function where each entry represents a unique state-action tuple, i.e. $Q_{ij} = Q(s_i, a_j)$, it will be the case in most high-dimensional problems that the optimal Q-function is actually very structured i.e. low-dimensional, even if the magnitude of the state space is very large.

A work by Yuzhe Yang et. al. in 2020 showed that the number of samples required to find an optimal Q-function is actually proportional to the rank of Q (assuming it is represented as a matrix). So the fact that we don't need an impossible amount of data to find optimal Q-functions in high-dimensional settings can be explained by the fact that the Q-functions themselves are highly ordered due to the low-dimensional structures embedded in these settings.

5.8.3 Example: Parallel Parking

A case-study on parallel parking using reinforcement learning was performed by Dwarakanath in 2020 which showed that state-of-the-art RL methods still fall short of more classical model-based control methods i.e. there is a long way to go in refining RL algorithms to produce truly optimal agents/controllers.